Software Architecture and Dependability

Valérie Issarny, Apostolos Zarras

Software Architecture and Dependability

Valérie Issarny[1] and Apostolos Zarras[2]

[1] INRIA, Domaine de Voluceau, B.P. 105, 78 153 Le Chesnay Cédex, France,
Valerie.Issarny@inria.fr
[2] Computer Science Department, University of Ioannina, Greece,
zarras@cs.uoi.gr

Abstract. Dependable systems are characterized by a number of attributes including: reliability, availability, safety and security. For some attributes (namely for reliability, availability, safety), there exist probability-based theoretic foundations, enabling the application of *dependability analysis techniques*. The goal of dependability analysis is to forecast the values of dependability attributes, based on certain properties (e.g. failure rate, MTBF, etc.) that characterize the system's constituent elements.
Nowadays, architects, designers and developers build systems based on an architecture-driven approach. They specify the system's software architecture using Architecture Description Languages or other standard modeling notations like UML. Given the previous, we examine what we need to specify at the architectural level to enable the automated generation of models for dependability analysis. In this paper, we further present a prototype implementation of the proposed approach, which relies on UML specifications of dependable systems' software architectures. Moreover, we exemplify our approach using a case study system.

1 Introduction

To characterize a system as a dependable one, it must be trustworthy. In other words, the users of the system must be able to rely on the services it provides. The less the system fails in providing correct service the more dependable it is. A system failure is the manifestation of a fault, which leads the system into an erroneous state. Building dependable systems amounts in building systems that do not fail, or building systems whose failure can be tolerated. In order to achieve the previous there are several techniques that have been proposed. These techniques can be classified into the following categories [22]:

- Fault prevention techniques, aiming at the avoidance of fault creation within the system.
- Fault tolerance techniques aiming at the provision of correct service, despite the presence of faults.
- Fault removal techniques, whose main objective is to reduce the presence of faults in the system.

M. Bernardo and P. Inverardi (Eds.): SFM 2003, LNCS 2804, pp. 259–285, 2003.

– Fault forecasting techniques, whose goal is to analyze and estimate the number of faults in the system and their consequences.

Developing dependable systems relies on a software development process that consists of a set of typical engineering work-flows. This set of work-flows is usually performed in an iterative manner. Namely, the work-flows we consider are:

– The requirements elicitation work-flow.
– The analysis and design work-flow.
– The implementation work-flow.
– The test work-flow.
– The deployment work-flow.

The development process further comprises work-flows that aim at managing the execution of the engineering work-flows. The previous consist of several tasks for managing workers (i.e., architects, designers, developers), the activities performed by those workers and the artifacts produced after the execution of the activities. Applying fault prevention, fault removal, fault tolerance and fault forecasting techniques requires introducing corresponding activities in the engineering work-flows of the software development process. Moreover, using the aforementioned techniques has also implications on the management work-flows.

Fault prevention involves applying specific design methodologies and construction rules. Consequently, there are activities to be added in the analysis and design work-flow and in the implementation work-flow. The management work-flows must further contain activities that constraint the workers participating in the aforementioned engineering work-flows to apply the fault prevention activities introduced in the engineering work-flows.

Fault tolerance techniques consist of: error recovery and error compensation techniques. Error recovery aims at taking the system from an erroneous state to an error-free state, while error compensation involves enhancing the system with redundant entities so as to be able to deliver correct service from an erroneous state. Based on the previous, the analysis and design work-flow must include activities that introduce fault detectors, fault notifiers, redundancy management, logging and recovery elements in the architecture of the dependable system. The implementation work-flow must contain activities that deal with the integration of the previous elements with the rest of the system's entities. Finally, the deployment work-flow must contain activities for properly deploying redundant elements on hardware nodes.

Fault removal techniques are composed of three basic steps: verification, diagnosis and correction. The verification step aims at checking whether the system's behavior is coherent with the system's expected behavior. If it is not, the other two steps must be performed. In general, during the verification step a number of constraints are checked against the system's actual behavior. The constraints may be either generic in that they are required for many different families of systems (dead-lock freedom, absence of starvation, absence of memory leaks), or specific to the particular system. System-specific constraints are deduced from

the users' functional requirements on the system (e.g., the system is able to successfully execute specific scenarios). Verification may be either static, or dynamic. In static verification, the constraints on the system behavior are checked against a model of the system (e.g., model checking techniques). Static verification techniques involve introducing specific activities in the analysis and design work-flow for building the system model in terms of a formalism like PROMELA [14], FSP [27], etc. Dynamic verification amounts in testing the runtime behavior of the system using random or deterministic test cases. Naturally, dynamic verification imposes performing specific activities in the testing work-flow.

By definition [22], dependability is a quite wide concept, which is characterized by a number of attributes including *reliability, availability, security* and *safety*. Depending on the system, our interest is usually narrowed into some of those attributes. The goal of fault forecasting is to estimate-predict the values of dependability attributes, based on certain properties (e.g., failure rate, MTBF, service rate, etc.) that characterize the system's constituent elements. From now on, we refer to fault forecasting techniques as dependability analysis techniques. Reliability analysis, for instance, aims at calculating the probability that the system provides correct service for a particular time period. Traditional techniques for dependability analysis rely on specifying constraints describing either what it means for the system to provide error-free service (Block Diagrams), or what it means for the system to provide erroneous service (Fault Trees). More sophisticated analysis techniques require modeling the system's failure and repair behavior using state space models.

Regarding the software development process, dependability analysis requires specifying related properties (e.g., failure rate, MTBF) characterizing the elements that make up the system. Consequently, we need to enhance the analysis and design work-flow to include such activities. Moreover, we have to enhance the deployment work-flow with activities that allow achieving the previous for the nodes used for executing the system's elements. Finally, constraints for error-free or erroneous service delivery and state space models must be specified during the analysis and design work-flow. The values of the properties that characterize the system's constituent elements may be assumed, or they may be based on measures gathered during the testing work-flow of a previous iteration.

In this paper, we present an approach for automating the previous activities. More specifically, in Section 2, we present general concepts related to the specification of software architectures and the dependability analysis of systems at the architectural level. Then, in Section 3, we examine what we need to specify at the architectural level to enable the automated generation of models for dependability analysis and how to generate them from architectural descriptions. Section 4 presents a prototype implementation of the proposed approach, which relies on UML to specify the architecture of dependable systems. In Section 5, we give details related to a case study we use for the assessment of the solution we propose. Finally, in Section 6, we summarize with our contribution and the future perspectives of this work.

2 Software Architecture and Dependability Analysis

As we mentioned in Section 1, our main goal is to facilitate the generation of constraints and state space models for the dependability analysis of systems, from the systems' architectural descriptions. Specifying software architectures involves using a notation. Architecture Description Languages (ADLs) are notations enabling the rigorous specification of the structure and behavior of software systems.

ADLs come along with tools supporting the analysis and the construction of software systems, whose architecture is specified using them. Several ADLs have been proposed so far (e.g., ASTER[17], CONIC [28], C2 [40], DARWIN [26], DCL [4], DURRA [5], RAPIDE [24], SADL [33], UNICON [38], WRIGHT [2]); they are more or less based on the same principles [7, 15, 30]. In particular, the architecture of software systems is specified using *components*, *connectors* and *configurations*.

Before getting into the semantics of components, connectors and configuration, it should be noted that ADLs are widely known and used in academia, but their use in the industry is quite limited. Industrials, nowadays, prefer using object-oriented notations for specifying the architecture of their software systems. UML, in particular, is becoming an industrial standard notation for the definition of a family of languages (i.e., UML profiles) for modeling software systems. However, there is a primary concern regarding the imprecision of the semantics of UML.

One way to increase the impact of ADLs in the real world and decrease the ambiguity of UML is to define an ADL that provides a set of core extensible UML-based language constructs for the specification of components, connectors and configurations. This core set of extensible constructs shall further facilitate future attempts for mapping existing ADLs into UML.

2.1 Components

A *component* is a unit of data or computation and the basic features that characterize it are its interface, type and properties.

A component *interface* describes a number of interaction points between the component and the rest of the architecture. All of the ADLs mentioned above support this particular feature. However, several syntactic and semantic differences have been observed between them. In ASTER, for instance, an interface defines a set of operations; components export interfaces to the environment and import interfaces from other architectural elements. In CONIC, an interface defines a set of entry and exit ports that are typed. In DARWIN, CONIC's successor, an interface specifies services required from and provided by a component. In DCL, components are called *modules*. A module is a group of actors, i.e., a group of processing elements that communicate through asynchronous point-to-point message passing [1]. A module description comprises request rules, which prescribe the module's interface. A component interface in C2 defines two kinds of

interaction points, named top and bottom ports. Ports are used by a particular component to accept requests from, and issue requests to, components that reside either above, or below it (the architecture is topologically structured). A component interface in UNICON defines a number of interaction points, called players. Players are typed entities. The type of a player can be out of a limited set of predefined types. In WRIGHT, a component interface defines input and output ports. Pretty similar is the way interaction points are defined in DURRA. In RAPIDE, the points of interaction can be either services required from or provided by a component, or events generated by a component. Finally, in SADL, an interface is just a point of interaction.

A component *type* is a template used to instantiate different component instances into a running configuration. All of the ADLs mentioned above distinguish between component types and instances. Types are usually extensible. Sub-typing (e.g., in C2, ASTER) is a typical method used to define type extensions. In DARWIN and RAPIDE, types are extended through parameterization.

Component *properties* characterize the component's observable behavior (which may include erroneous behavior). In WRIGHT, behavior is described in CSP [12, 13]. In RAPIDE, partially ordered sets of events (posets) are used to describe component behavior. In the very first version of DARWIN, properties were described in CCS [32]; in the latest version, properties are described in pi-calculus, which extends the semantics of CCS with means that allow specifying the dynamic instantiation of processes [31]. In DCL, the behavior of a module is deduced by the behaviors of the actors that constitute the module. An extension of the basic ACTORS formalism is used to describe the behavior of actors [3] within a software architecture. Finally, in ASTER, temporal logic is used to describe properties. Similarly, in SADL, the authors propose using Temporal Logic of Actions (TLA) [21] for the specification of component properties.

2.2 Connectors

A connector is an architectural element that models the interaction protocols among components. Its basic features are again its interface, type, and properties.

Some ADLs do not consider connectors as first-order architectural elements (e.g., CONIC, DARWIN, RAPIDE). For the other ADLs, a connector specification is similar to a component specification. In WRIGHT and UNICON, for instance, a connector interface is a set of interaction points, named roles. In DURRA, a connector is called channel and its interface is defined in the very same way as a component interface. In C2 and SADL, connector interfaces are described using the same syntax as the one used to describe component interfaces. In DCL, connectors are again groups of actors, called *protocols*. Protocols define a set of roles describing the way interaction takes place among modules. In all ADLs, except for UNICON, connector types are extensible. The formalism used for the specification of component properties is further used for the specification of connector properties.

2.3 Configurations

A configuration is the assembly of components and connectors. It is described in terms of associations (usually called bindings) between points of interaction. Several ADLs either assume or provide means to describe *constraints* for a particular configuration.

Constraints may simply describe restrictions on the way components are bound. In DARWIN, for instance, only bindings between required and provided services are allowed. In ASTER, the types of the interfaces that are bound should match. Some ADLs allow specifying constraints on the behavior of the overall configuration. In ASTER, for example, we can specify dependability requirements for a particular configuration. RAPIDE also allows specifying constraints on the behavior of a particular configuration. Constraints may also relate to the (dynamic) evolution of a particular configuration. In DURRA and RAPIDE, for example, it is possible to describe conditions under which a configuration changes into another one.

2.4 ADLs and Dependability Analysis

Pioneer work on the dependability analysis of software systems at the architectural level includes Attribute-Based Architectural Styles (ABAS) [25]. In general, an architectural style includes the specification of: types of basic architectural elements (e.g., pipe and filter) that can be used for specifying a software architecture, constraints on the use of these types, and patterns describing the data and control interaction between them.

An ABAS is an architectural style, which additionally provides modeling support for the analysis of a particular quality attribute. Dependability attributes (i.e., reliability, availability, safety) are among the quality attributes for which we can define ABASs. More specifically, an ABAS includes the specification of:

- *Quality attribute measures* characterizing the quality attribute (e.g., the probability that the system correctly provides a service for a given duration).
- *Quality attribute stimuli*, i.e., events affecting the value of the quality attribute measures (e.g., failures).
- *Quality attribute properties*, i.e., architectural properties affecting the value of the quality attribute measures (e.g., faults, redundancy).
- *Quality attribute models*, i.e., traditional models that formally relate the above elements (e.g., a state space model that predicts reliability based on the failure rates and the redundancy used).

In [20], the authors introduce the Architecture Tradeoff Analysis Method (ATAM) where the use of an ABAS is coupled with the specification of a set of scenarios that constitutes a *service profile*. ATAM has been applied for analyzing quality attributes like performance, availability, modifiability, and real-time. In all these cases, quality attribute models (e.g., state-space models, queuing

networks, etc.) are manually built given the specification of a set of scenarios and an ABAS-based architectural description of a system. However, in [20], the authors recognize the complexity of the aforementioned task; the development of quality analysis models requires about 25% of the time spent for applying the whole method. ATAM is a promising approach for doing things right. However, it needs to be enriched for facilitating the specification of quality models.

One solution to the previous lies on the automated generation of quality attribute models from architectural descriptions. Note that there is no unique way to model systems. A model is built based on certain assumptions. Thus, the model generation procedures should be customizable. Customization is done according to the assumptions, made by the developer for the quality stimuli and properties, affecting the value of the particular quality attribute that is assessed. While this paper concentrates on dependability quality attributes, the interested reader may refer to [43] for details regarding the case of performance.

3 ABAS for Automated Dependability Analysis of Software Architectures

As already mentioned in the introduction, dependability is characterized by a number of attributes including reliability, availability, safety, security. For reliability, availability and safety, there exist probability-based theoretic foundations, enabling *dependability analysis*. In this section, we define an ABAS that facilitates dependability analysis regarding these attributes [42].

To perform dependability analysis, we have to specify a service profile, i.e., a set of scenarios, describing how the system provides a particular service. A scenario (e.g., a UML collaboration or sequence diagram) specifies the interactions among a set of component and connector instances, structured as prescribed by the configuration of the system. Scenarios are associated with the values of the dependability measures that the system's users require (these values are gathered during the requirements elicitation). Moreover, the definitions of the base architectural elements are associated with dependability measures, properties, and stimuli, as detailed below.

3.1 Dependability Measures, Stimuli, and Properties

The basic reliability measure we use is the probability that the system provides correct service for a given time period. Similarly, the availability measure we consider is the probability that the system provides correct service at a given moment in time. For safety, a typical measure is the probability that there will be no catastrophic failure for a given time period. Hence, safety analysis is reliability analysis regarding only catastrophic failures.

A scenario may fail if instances of components, nodes[3], and connectors used in it, fail because of faults causing errors in their state. The manifestations of

[3] An architectural component is assumed to be associated with a set of nodes on top of which it executes. For primitive components the associated set contains one node, while for composite components, it may contain more than one node

errors are failures. Hence, faults are the basic *properties*, associated with components/connectors/nodes, which affect the dependability measures. Failures are the *stimuli*, associated with components/connectors/nodes, causing changes in the value of the dependability measures. According to [22], faults and failures are further characterized by the features given in Tables 1 and 2. Different combinations of the values of these features can be used to customize properly the generation of dependability models, which is detailed in Section 3.2.

Features	**Range**	**Associated Architectural Element**
domain	time/value	Component/Connector/Node
perception	consistent/inconsistent	

Table 1. Dependability Stimuli: Specification of Failures

Features	**Range**	**Associated Architectural Element**
nature	intention/accident	Component/Connector/Node
phase	design/operational	
causes	physical/human	
boundaries	internal/external	
persistence	permanent/temporary	
arrival-rate	Real	
active-to-benign	Real	
benign-to-active	Real	
disappearance	Real	

Table 2. Dependability Properties : Specification of Faults

Another property of the base architectural elements that affects dependability measures is redundancy. Redundancy schemas can be specified using the base architectural constructs defined in Section 2. More specifically, a redundancy schema is a composite component that encapsulates a configuration of redundant architectural elements, which behave as a single fault tolerant unit. According to [23], a redundant schema is characterized by the following features: the kind of mechanism used to detect errors, the way the constituent elements execute towards serving incoming requests, the confidence that can be placed on the results of the error detection mechanism and the number of component and node faults that can be tolerated. The features characterizing a redundancy schema are summarized in Table 3. A repairable redundancy schema is characterized by additional features (e.g., repair-rate).

3.2 Dependability Models

The dependability properties, stimuli and measures can be formally related using Block Diagrams (BDs), Fault Trees (FTs) and state space models [34, 11, 35].

Features	Range	Associated Architectural Element
error-detection	vote/comp./acceptance	Component
execution	parallel/sequential	
confidence	absolute/relative	
service-delivery	continuous/suspended	
no-comp-faults	Integer	
no-node-faults	Integer	

Table 3. Redundancy Property

A BD represents graphically a constraint for providing a service S. Hereafter, we call such a constraint, *constraint-to-succeed*. The BD consists of a set of system components that need to be operational to provide S (i.e., the components participating in a scenario that describe how the system provides S). Every component C in the BD is characterized by certain dependability measures. The reliability (resp. availability) measure for C is the probability that C provides correct service for a time period T (resp. time instance t). The safety measure for C is the probability that there is no catastrophic failure of C during a time period T. Components are connected using serial or M-out-of-N parallel connections. If we connect N components using serial connections, all of them must be operational to provide S. On the other hand, if we connect them using an M-out-of-N parallel connection, at least M components out of the set must be operational to provide S. The overall system reliability (resp. availability, safety) is obtained through simple combinatorial calculations involving the reliability (resp. availability, safety) measures of the individual components that belong to the BD.

Taking an example, suppose that providing a service for a time period T requires using components C1, C2 and C3. The corresponding constraint-to-succeed can be specified as a logical formula, $C1 \wedge C2 \wedge C3$, consisting of the conjunction of three predicates. Predicates $C1$, $C2$, $C3$ are true if components C1, C2, C3 are operational and false otherwise. The BD that graphically represents the constraint-to-succeed is shown in Figure 1(a). According to that BD, C1 is connected in serial with C2, which is further connected in serial with C3. The overall reliability is the probability that the $C1 \wedge C2 \wedge C3$ constraint holds:

$$\begin{aligned} BD.reliability &= P(C1 \wedge C2 \wedge C3) \\ P(C1 \wedge C2 \wedge C3) &= C1.reliability * C2.reliability * C3.reliability \end{aligned}$$

Suppose now that providing a service S for a time period T requires using either components C1, C2 or C1, C3. Again, the constraint-to-succeed can be described as a logical formula, $C1 \wedge (C2 \vee C3)$. The corresponding BD is given in Figure 1(b). C2 and C3 are connected with a 1-out-of-2 parallel connection forming a new block, which is connected in serial with C1. The overall reliability is the probability that the $C1 \wedge (C2 \vee C3)$ constraint holds:

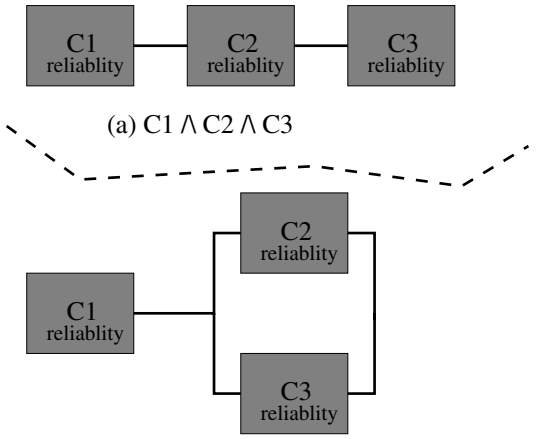


Fig. 1. Example of a Block Diagram.

$$
\begin{aligned}
BD.reliability &= P(C1 \wedge (C2 \vee C3)) \\
P(C1 \wedge (C2 \vee C3)) &= C1.reliability * C2.reliability + \\
&\quad C1.reliability * C3.reliability - \\
&\quad\quad C1.reliability * C2.reliability * C3.reliability
\end{aligned}
$$

So far, we calculate the dependability measures of a particular system as a function of the dependability measures that characterize the components of this system. However, we can further think of dependability measures as a function of the probability that the system fails. To calculate the probability of system failure we have to identify and model what should happen for the system to fail. The previous can be achieved using FTs [34, 11, 35]. FTs and BDs are equivalent in the sense that the values of the dependability measures obtained are the same. Moreover, having a BD, we can easily generate automatically an equivalent FT, and conversely. However, BDs and FTs enable modeling the system from different perspectives, depending on which one is more convenient for the worker in charge of the dependability analysis.

An FT visualizes a constraint, which describes undesired stimuli (i.e., failures) that lead to system failure. Hereafter, we call such a constraint, *constraint-to-fail*. The overall system failure is called the top-event. Undesired events are connected with AND and OR gates. AND gates connect events whose subsequent or concurrent occurrence triggers the top-event. OR gates connect events whose alternative occurrence triggers the top-event. Every event is characterized by the probability of its occurrence (P_{occur}).

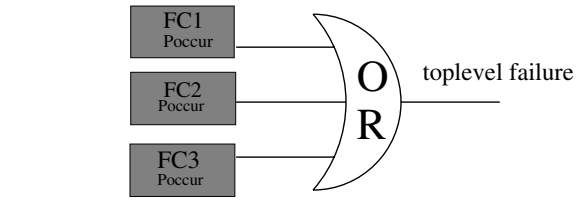
Taking an example, suppose that providing a service S requires using components C1, C2 and C3, then a failure of any of them leads to system failure. The aforementioned constraint can be described as a logical formula, $FC1 \vee FC2 \vee FC3$. Predicates $FC1$, $FC2$, $FC3$ are true if components C1, C2, C3, respectively, have failed and false otherwise. The resulting FT, shown in Figure 2(a),

depicts an OR gate that takes as input the failure events of C1, C2, C3 and has as output the failure of the overall system. The reliability in this case is:

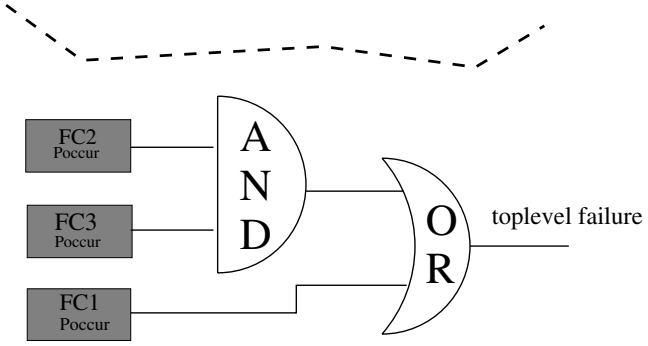
$$\begin{aligned} FT.reliability &= 1 - P(FC1 \vee FC2 \vee FC3) \\ P(FC1 \vee FC2 \vee FC3) &= FC1.P_{occur} + FC2.P_{occur} + FC3.P_{occur} - \\ &\quad FC1.P_{occur} * FC2.P_{occur} - \\ &\quad FC1.P_{occur} * FC3.P_{occur} - \\ &\quad FC2.P_{occur} * FC3.P_{occur} + \\ &\quad FC1.P_{occur} * FC2.P_{occur} * FC3.P_{occur} \end{aligned}$$

Suppose now that S requires using component C1 and either component C2, or component C3. Then, a failure of both C2 and C3 leads to system failure. Alternatively, a failure of C1 leads to system failure. The previous can be specified as a logical formula, $FC1 \vee (FC2 \wedge FC3)$. Figure 2(b) gives the corresponding FT. The reliability in this case is:

$$\begin{aligned} FT.reliability &= 1 - P(FC1 \vee (FC2 \wedge FC3)) \\ P(FC1 \vee (FC2 \wedge FC3)) &= FC2.P_{occur} * FC3.P_{occur} + FC1.P_{occur} - \\ &\quad FC2.P_{occur} * FC3.P_{occur} * FC1.P_{occur} \end{aligned}$$



(a) C1 ∧ C2 ∧ C3



(b) C1 ∧ (C2 ∨ C3)

Fig. 2. Example of a Fault Tree.

The techniques we presented until now, rely on static descriptions of either the components we need for correct service provisioning (i.e., BDs), or the fail-

ures that lead to an overall system failure (i.e., FTs). Although those techniques are quite easy to apply, they do not cover cases where we have to model dynamic aspects of the system that affect the values of the dependability measures. For example, the dependability analysis of systems with transient faults involves modeling that those faults disappear with a certain rate. Similarly, the dependability analysis of systems with intermittent faults requires modeling the way those faults activate (if an intermittent fault is active the service is not correctly provided) and passivate (if an intermittent fault is passive the service is correctly provided despite the presence of the fault), during the lifetime of the system. In other words, we have to model the failure behavior of the components and connectors that make up the system. In the case of repairable systems, we have to further model how faulty architectural elements eventually become operational, and conversely. Another issue that we can not model with BDs and FTs is the occurrence of dependent failures.

Modeling and analyzing the failure and repair behavior of systems relies on state space models [34, 11, 6, 10]. A state space model consists of a set of transitions between states of the system. A state describes a situation where either the system operates correctly, or not. In the latter case, the system is said to be in a *death state*. The state of the system depends on the states of the architectural elements that constitute it. Hence, a state can be seen as a composition of sub-states, each one representing the situation of an architectural element. A state is constrained by the range of all possible situations that may occur. A transition is characterized by the rate by which the source situation changes into the target situation. If, for instance, the difference between the source and the target situation is the failure of a component, the transition rate is the faulty component's failure rate. If, on the other hand, the difference between the source and the target situation is the repair of a component, the transition rate is the component's repair rate. The mathematical model that is employed for calculating reliability and availability based on a state space model, involves solving a system of first order differential equations.

Taking an example, suppose that in order to provide a service S we have to use components C1, C2 and C3. Moreover, suppose that C1, C2 and C3 have permanent faults. The state space model that specifies the failure behavior of the system is given in Figure 3; it consists of four states representing the following situations:

State 1 C1, C2, C3 are operational.
State 2 C1 failed, C2, C3 are operational (death state).
State 3 C2 failed, C1, C3 are operational (death state).
State 4 C3 failed, C1, C2 are operational (death State).

The state space model comprises transitions from state 1 to states 2, 3, 4 characterized by the failure rates of C1, C2, C3, respectively.

Let $P(t) = [p_1(t), p_2(t), p_3(t), p_4(t)]$ be a vector that gives the probabilities that the system is in states 1, 2, 3, 4, respectively. The system of differential equations that can be used to calculate those probabilities is the following: $P'(t) = P(t) * A$ where A is a matrix that can be easily calculated from the state

space model as follows: For every transition from state i to state j, set A(i, j) equal to the transition rate. The value of every diagonal element A(i, i) is set to the negated sum of the non-diagonal i row elements of the matrix.

$$\mathrm{A} = \begin{vmatrix} -\sum(A(0,i))_{i=1\ldots3} & C1.failure\_rate & C2.failure\_rate & C3.failure\_rate \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Assuming that $P(0) = [1, 0, 0, 0]$, and that the failure rates of C1, C2, C3 are constant we have the following solution for $P(t)$:

$$P(t) = P(0) * e^{A*t}$$

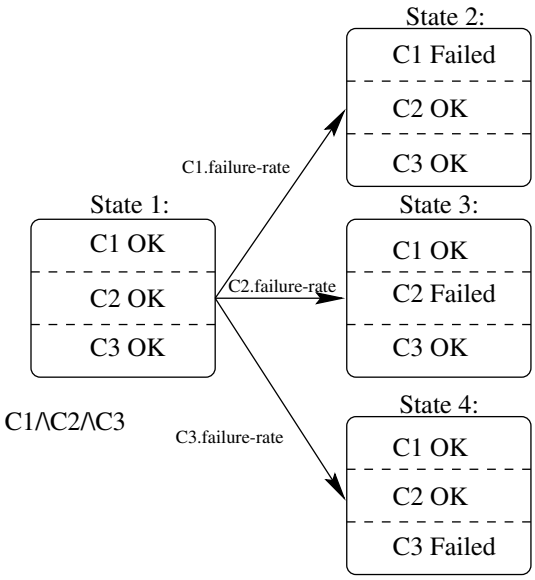


Fig. 3. Example of a state space model.

3.3 Automated Generation of State Space Models from Architectural Descriptions

The specification of large state-space models is often too complex and error-prone. The approach proposed in [19] alleviates this problem. In particular, instead of specifying all possible state transitions, the authors propose specifying the state range of the system, a death-state constraint, and transition rules between sets of states of the system.

The state range consists of a set of variables whose values describe a possible state situation. For example, a system that consists of a redundancy schema of three redundant components may be in 4 states. In each state $i : [0..3]$, $3 - i$ redundant components are operational. Then, the state range is defined as a single variable $numOfOperational : [0..3]$ whose value specifies the number of operational components.

A transition rule may state that: if the system is in a state where more than 1 component are operational (e.g., $numOfOperational > 1$), then the system may get into a state where the number of operational components is reduced by one (e.g., $numOfOperational = numOfOperational - 1$). Given the previous information, a complete state space model can be generated using the algorithm described in [19]. Briefly, the algorithm takes as input an initial state (e.g., the state 0 where $numOfOperational = 3$) and recursively applies the transition rules. During a recursive step and for a particular transition rule, the algorithm produces a transition to a state derived from the initial one. If the death-state (e.g., $numOfOperational <= 1$) constraint holds for the resulting state, the recursion stops.

State range definitions, transitions rules and death constraints can be automatically generated from architectural descriptions embedding the specification of dependability stimuli and properties, by following the steps below.

First, a state range definition for each scenario *scen* belonging to a given service profile is generated. The state of a scenario is composed of the states of the component and connector instances used within the scenario and the state of nodes on top of which the component instances execute. If a component is composite, its state is composed of the states of the constituent elements. The state range for the scenario consists of a set of variables, each one of which corresponds to a component/connector/node. The values of a variable depend on the kind of faults that may cause failures. At this point, the generation procedure is customized accordingly. In the case of permanent faults for instance, a component/connector/node may be either in an operational, or in a failed state. Hence, the corresponding state range variable may take two possible values OPERATIONAL or FAILED. In the case of intermittent faults, a component/connector/node may be in an operational state, or it may be in a failed-active or in a failed-benign state. Consequently, the corresponding state range variable may take three values OPERATIONAL, FAILED-ACTIVE, FAILED-BENIGN. The values of a state range variable further depend on the kind of redundancy used (take for instance the example we gave above with the 3 redundant components). Again, the generation procedure is customized accordingly.

After generating the state range definition for a scenario *scen*, the step that follows comprises the generation of transition rules for components/connectors/-nodes used in the scenario. These rules depend on the kinds of faults of the corresponding architectural element. For instance, for permanent faults, the rules follow the pattern given in Table 4. What is left at this point is to generate the definition of the initial state of the scenario, and the definition of the death state constraint. The initial state is a state where all of the elements used in the

Architectural Element	**Rule**
Component	For all instances of primitive components, c: – If *scen* is in a state where c is in an OPERATIONAL state st, then *scen* may get into a state st' where c is FAILED. The rate of these transitions is equal to the arrival rates of the faults that cause the failure of c, $c.Faults.arrival\text{-}rate$ (see Table 2). For all instances of composite components, c: – If *scen* is in a state st where c is OPERATIONAL, then *scen* may get into a state st' where c is FAILED due to a failure of a constituent element c'. The rate of these transitions is equal to the arrival rates of the faults that cause the failure of c', $c'.Faults.arrival\text{-}rate$. For all instances of composite components rc, representing a redundancy schema of k components: – If *scen* is in a state st where rc is OPERATIONAL, and the number of failed redundant component instances is fc, then *scen* may get into a state st' where the number of failed components of rc is $fc + l$. The difference between st and st' is l redundant component instances of the same type t, which in st were OPERATIONAL and in st' are FAILED. This rule captures failure dependencies among redundant component instances of the same type. These components are used in the same conditions and with the same input. Hence, if one of them fails due to a design or an operational fault, all of them will fail.
Connector	For all instances of primitive connectors, see the case of primitive components. For all instances of composite connectors, see the case of composite components.
Node	We assume that nodes fail independently from each other. Hence, for all nodes in *scen*: – If *scen* is in a state st where a node n is in an OPERATIONAL state, then *scen* may get into a state st' where n is in a FAILED state. – Moreover, in st', all instances of components c deployed on n are in a FAILED state. – Finally, in st' all instances of redundancy schemas rc, built out of m components deployed on n, have $fc + m$ failed components and $fn + 1$ failed nodes. The rate of these transitions is equal to the arrival rate of the faults that caused the failure of n, $n.Faults.arrival\text{-}rate$.

Table 4. Transition Rules for Permanent Faults

scenario are operational. A scenario is in death state if any of the architectural elements used in it is not operational. Hence, the death state constraint consists of the disjunction of base predicates, each one of which defines the death state constraint for an individual element used in the scenario. More specifically, the base predicate for a component, connector, or a node, states that the value of the corresponding state range variable is FAILED. The base predicate for a redundancy schema is the disjunction of two predicates. The first one states that the number of failed redundant component instances is greater than the number of component faults that can be tolerated. Similarly, the second one states that the number of failed redundant nodes is greater than the number of node faults that can be tolerated.

4 A Developer-Oriented Environment for Dependability Analysis

The ideas proposed so far for dependability analysis at the architectural level are realized in the prototype implementation of a developer-oriented environment [43, 36]. As we already discussed in Section 2, UML is an emerging industrial standard for modeling the architecture of software systems. Consequently, our environment relies on an already existing UML modeling tool. More specifically, we use the Rational Rose tool[4] for the specification of software architectures.

However, we further mentioned the fact that the semantics of UML are imprecise compared to the ones of the ADLs we examined in Section 2. Consequently, we proposed defining an ADL that extends the standard UML semantics towards dealing with this lack of precision. To define ADL components, connectors, and configurations in relation to standard UML model elements, we undertook the following steps: (i) identify standard UML element(s), whose semantics are close to the ones needed for the specification of ADL components, connectors and configurations; (ii) if the semantics of the identified element(s) do not exactly match the ones needed for the specification of components, connectors, and configurations, extend them properly and define a corresponding UML stereotype(s)[5]; (iii) If the semantics of the identified element(s) match exactly, adopt the element(s) as a part of the core ADL language constructs.

As discussed in the literature [9, 29], various UML modeling elements may be used to specify an ADL component. The most popular ones are the Class, Component, Package, and Subsystem elements. From our point of view, the UML Component element is semantically far more concrete compared to an ADL component, as it specifically corresponds to an executable software module. The UML Class element is often considered as the basis for defining architectural

[4] `http://www.rational.com`. Notice that the use of the Rational Rose tool was mainly motivated by pragmatic consideration that is the ownership of a license and former experience with this tool. However, our specific developments may be integrated within any extensible, UML-based tool that processes XMI files.

[5] A UML stereotype is a UML element whose base class is a standard UML element. Moreover, a stereotype is associated with additional constraints and semantics.

components. However, a UML class does not directly support the hierarchical composition of systems. It is true that the definition of a UML Class may be composite, consisting of a number of constituent classes. However, the class specification can not contain the interrelationships among the constituent classes. Consequently, if an ADL composite component is mapped into a UML class, its definition may comprise a set of constituent components for which we have no means to describe the way they are connected through connectors. Technically, to achieve the previous we would need to define a Package containing the UML class definitions and a static structure diagram showing how they are connected. However, packages cannot be instantiated or associated with other packages. Hence, they are not adequate for specifying ADL components. This leads us to use the UML Subsystem element to model ADL components. A UML Subsystem is a subtype of both the UML Package and Classifier element. Hence, it may be instantiated multiple times, and associated with other subsystems. Precisely, we define an ADL component as a stereotyped UML Subsystem, that may provide and require standard UML interfaces. The ADL component stereotype is characterized by a property, named *"composite"*, which may be true or false, depending on whether or not a component is built out of other components and connectors. Moreover, the ADL component stereotype is associated with the dependability features identified in Tables 1, 2 and 3.

The natural choice for specifying ADL connectors in UML is by stereotyping the standard UML Association element. A connector role corresponds to an association end. Moreover, the distinctive feature of a connector is a non-empty set of interfaces, named *"Interfaces"*, representing the specific parts of components' functionality playing the roles. Each interface out of the set must be provided by at least one associated component. Equally, each interface out of the set must be required by at least one associated component. The ADL Connector stereotype is further characterized by the dependability features identified in Tables 1 and 2.

So far, we considered connectors as associations representing communication protocols. However, we must not ignore the fact that, in practice, connectors are built from architectural elements, including components and more primitive connectors. Taking CORBA for example, a CORBA connector can be seen as a combination of functionalities of the ORB and of CORBA services (i.e., COSs). Hence, it is necessary to support hierarchical composition of connectors. At this point, we face a technical problem: UML Associations can not be composed of other model elements. However, there exists a standard UML element called Refinement defined as *"a dependency where the clients are derived by the suppliers"* [37]. The refinement element is characterized by a property called mapping. The values of this property describe how the client is derived by the supplier. Hence, to support the hierarchical composition of connectors, we define a stereotype, whose base class is the standard UML Refinement element and is used to define the mapping between a connector and a composite component that realizes the connector.

By definition, a configuration specifies the assembly of components and connectors. In UML, the assembly of model elements is specified by a model. The corresponding semantic element of a model is the standard UML Model element, defined as *"an abstraction of a modeled system specifying the system from a certain point of view and at a certain level of abstraction...the UML Model consists of a containment hierarchy where the top most package represents the boundary of the modeled system"* [37]. Hence, a configuration is actually a UML model, consisting of a containment hierarchy where the top-most package is a composite ADL component. The given definition of configuration is weak in that it enables the description of any architectural configuration provided it complies with the well-formedness rules associated with the component and connector elements. This results from our concern of supporting the description of various architectural styles, which possibly come along with specific ADLs as is the case with the C2 style [30]. Constraints that are specific to a style are introduced through the definition of a corresponding extension of the ADL configuration element, possibly combined with extensions of the ADL component and connector stereotypes.

The Rational Rose tool allows the definition of user specific add-ins that facilitate the specification and use of stereotyped elements and their associated features. Given the aforementioned facility, we implemented an add-in that eases the specification of architectural descriptions using the stereotypes mentioned above. Moreover, we use an already existing add-in, which enables generating XMI textual specifications of architectures specified graphically using the Rational Rose tool; these textual specifications serve as input to tools for dependability and performance analysis [43].

The generation of the XMI textual specifications for dependability analysis relies on the automated procedure we described in Section 3 (the procedure and the tools we use for the case of performance are detailed in [43]). The specific tool we use for dependability analysis is SURE-ASSIST [6]. The tool is properly customized to accept as input the textual specifications we generate. Then, it calculates reliability bounds. The tool was selected because it is highly rated among other reliability tools [10] and because it is available for free. However, the automated support provided by our environment for dependability analysis can be coupled with any other tool that accepts as input state space models.

5 The Developer-Oriented Environment in Action

To illustrate the use of our environment for dependability analysis, we employ an example taken from a case study we investigated in the context of the DSoS IST project[6]. The case study is a travel agent system (TA). TA offers services for flight, hotel, and car reservations. It consists of the integration of different kinds of existing systems supporting air companies, hotel chains, and car rental companies. Figure 4 gives a screen shot of the actual architecture of the TA as

[6] `http://www.newcastle.research.ec.org/dsos`

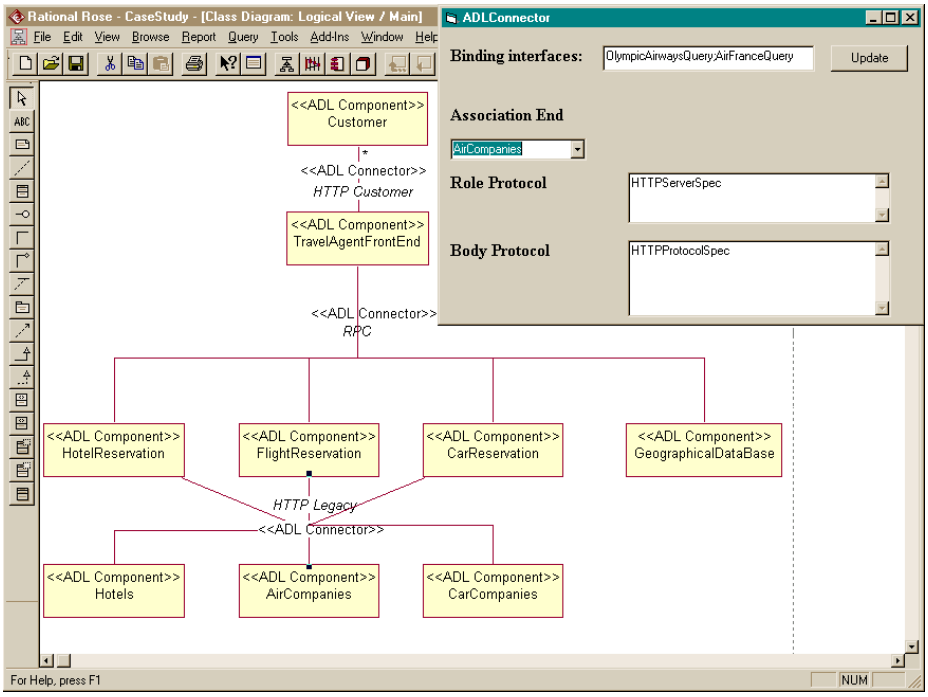


Fig. 4. The Architecture of the Travel Agent system.

specified using the UML modeling tool, which we customized. The TA comprises the *TravelAgentFrontEnd* component, which serves as a GUI for potential customers wanting to reserve tickets, rooms, and cars. The TA further includes the *HotelReservation*, *FlightReservation*, *CarReservation* components, which accept as input individual parts of a customer request for hotel, ticket and car reservation, and translate them into requests for services provided by specific hotel, air company and car company components. The set of the hotel components is represented by the Hotels composite component. Similarly, the sets of air company and car company components are represented by the *AirCompanies* and *CarCompanies* composite components. Two different kinds of connectors are used in our architecture. The HTTP connectors (e.g., see Figure 4) represent the interaction protocol among customers and the TA front end component, and among components translating requests and existing component systems implementing Web servers. The RPC connector represents the protocol used among the front end component and the components that translate requests. Note that multi-party connectors abstract complex connector realizations, which may actually be refined into various protocols, depending on the intended behavior. For instance, the RPC connector may be refined into a number of bi-party connectors as well as into a complex transactional connector.

The dependability measure we are interested in is reliability. However, the goal of the analysis is not to obtain precise values of the reliability measure since this would require to precisely model the Internet. The previous is considered, in general, as rather unrealistic [8]. For that reason, we concentrate on comparing different scenarios towards improving the design of our system, while assuming certain invariants for modeling issues related to the Web. Our objective is to try to improve the reliability of TA while keeping the cost of the required changes in the TA system low.

The scenario shown in Figure 5 as a UML collaboration diagram, is a typical use case of TA. This scenario constitutes the basic service profile used for the reliability analysis, i.e., the provided scenario is processed for the automatic generation of the state space model analyzed by the SURE-ASSIST tool. According to the scenario, one or more customers use an instance, *ta*, of the *TravelAgent-FrontEnd* to request the reservation of a flight ticket, a hotel room and a car. The *ta* component instance breaks down such a request into 3 separate requests. The first one relates to the flight ticket reservation and is sent to an instance, *fr*, of the *FlightReservation* component. The *fr* component instance uses this request to generate a new set of requests, each one of which is specific to an air company that collaborates with the TA system. The set of specific requests is finally sent to an instance, *ac*, of the *AirCompanies* composite component, which represents the current set of collaborating air companies. Similarly, the second and the third requests are related to the hotel and the car reservations, respectively. These requests are sent to instances of the *HotelReservation* and *CarReservation* components, which reproduce them properly and send them to the current sets of collaborating hotels and car companies.

The component instances used in the scenario may fail to give answers to customers. Component failures are manifestations of design faults. We assume that these faults are accidental, created by the component developers. Moreover, component faults are all permanent and their arrival rates vary depending on the type of the components. More specifically, the fault arrival rates for the components that represent component systems supporting hotels, air companies and car companies are much smaller compared to the faults arrival rates of the rest of the components that make up the TA system. The reason behind this is that the component systems supporting hotels, air companies and car companies have already been in use and their implementations are quite stable. On the other hand, the TA front end and reservation components are still under development. The nodes used in our scenario may fail because of permanent faults. HTTP and RPC connectors may also fail, however, in this case it is more pragmatic to assume that we deal with temporary faults, which may disappear with a certain rate. The arrival rates of node faults are much smaller than the arrival rates of component faults. This holds similarly for the RPC connector. On the contrary, the HTTP connector is expected to be quite unreliable, with a failure rate greater than that of the components used in the TA. For illustration, Figure 5 shows the detailed specification of the reliability stimuli and properties that are given for the *FlightReservation* component.

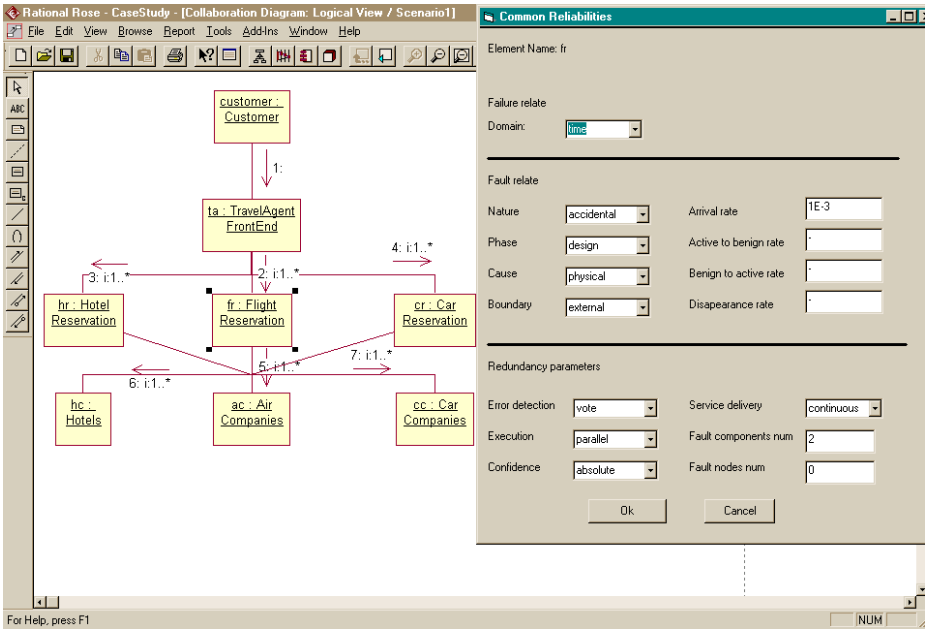


Fig. 5. A generic scenario for TA

By taking a closer look at the architecture of the TA system, we can deduce that some sort of redundancy is used. In particular, the *Hotels*, *AirCompanies* and *CarCompanies* components are composite, consisting of k components that represent the dependable systems supporting hotels, air companies and car companies. The reservation components request from them, room, ticket and car reservations. For the scenario to be successful, we need answers from at least one hotel, one air company, and one car company. Hence, *Hotels*, *AirCompanies*, and *CarCompanies* can be seen as ad hoc redundancy schemas with the following properties: the execution of redundant elements is parallel ($Redundancy.execution = parallel$), the number of component and node faults that can be tolerated is $k - 1$ ($Redundancy.no\text{-}comp\text{-}faults$ and $Redundancy.no\text{-}node\text{-}faults = k - 1$).

To further improve the architecture regarding the provided reliability, we designed three additional redundancy schemas. The first one contains n different versions of the *HotelReservation* component. Upon the instantiation of the schema, n component instances are created, one of each version. These instances execute in parallel and are deployed on n different nodes. The second schema contains n versions of the *FlightReservation* component, the instances of which are also deployed on the n nodes, on top of which the instances of the *HotelReservation* component execute. Finally, the last schema contains n versions of the *CarReservation* component, the instances of which are also deployed on the nodes used to execute the instances of the *HotelReservation* component. At

runtime, a customer request is broken down by the instance of the *TravelAgentFrontEnd* component into individual requests for flight ticket, hotel room and car reservation. Each one of these requests is replicated and sent to all the redundant instances of the corresponding reservation component. Each instance of the reservation component translates the request into specific requests for the corresponding available component systems and sends them. When the instance of the *TravelAgentFrontEnd* starts receiving offers for flight tickets, hotel rooms and cars, it removes identical reply messages and combines them into replies that are returned to the customer. We tried our scenario for $n = 1, 2, 3$ redundant versions. Given the aforementioned scenario, three complete state space models were generated and analytically solved. The results obtained are summarized in Figure 6. For further detail about the scenario, including complexity of the generated state space models, the interested reader is referred to [43].

The main observation we make is that the reliability of TA does increase. However, the improvement when we use redundant versions is certainly not spectacular. The explanation for this is simple. In our scenario, the most unreliable element used is the HTTP connector. This is the main source causing the reliability measure to have small values. Any improvement in the rest of the architectural elements used shall not cover this problem, which unfortunately can not be easily alleviated. Hence, using multiple versions does not bring much gain. However, the good news are that regarding the cost of using multiple versions, we do not lose much. The elements for which we produced multiple versions just translate TA specific requests into component systems' specific requests. Since the functionality of these components is quite simple, re-implementing them differently (e.g., using different developers) is not a complex, neither a time-consuming task. Note here that the fact that the functionality of the redundant components is simple does not mean that there can be no bugs in their implementation. Actually, mistakes in the mapping of TA requests into component systems' specific requests can be quite often. Furthermore, the cost of developing multiple versions is low since we did not really use any strong synchronization among the different versions.

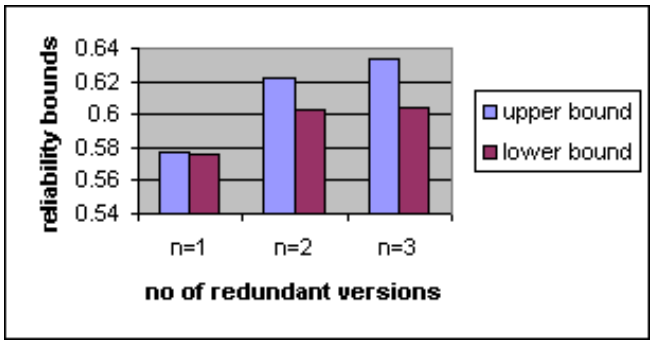


Fig. 6. Results produced by the reliability analysis of TA

6 Conclusion

Work in the software architecture domain primarily focuses on the standard (as opposed to exceptional) behavior of the software system. However, it is crucial from the perspective of system dependability to also account for failure occurrences, which impacts all the phases of the software development process, from the requirements elicitation phase to the deployment phase. In the context of the research activities of the INRIA ARLES research group[7], we have more specifically concentrated on solutions assisting the design and analysis of dependable distributed software systems.

6.1 Assisting the Analysis of Dependable Systems

In this paper, we presented automated support for the dependability analysis of software systems at the architectural level. The overall design and realization of the resulting development environment is guided by the needs of its current and potential users, imposing the simplification of certain important and inevitable development activities related to the quality analysis and assurance of dependable systems. The quality analysis of systems is traditionally based on methods and tools that have a strong formal basis. We believe that the proposed environment brings everyday developers closer to such methods and tools. The environment relies on an architecture description language for the specification of dependable systems architectures, which is defined based on UML, a standard and widely accepted notation for modeling software. Our environment further provides a certain level of automation that eases the development of traditional quality models from architectural descriptions. The associated prototype has been used in the context of the DSoS IST project for the quality analysis of the Travel Agent system. Part of the analysis was presented here in the form of demonstrating examples. We further used the basic ideas of our environment in the context of the C3DS IST[8] project for the performance and reliability analysis of workflow-based dependable systems [41].

6.2 Assisting the Design of Dependable Systems

From the perspective of the system's design, failures may be handled through the integration within the system architecture of components and connectors that provide fault tolerant capabilities. Practically, this means that failures are handled by an underlying fault-tolerant mechanism (e.g., transparent replication management) at the middleware level. Such fault tolerance support must further be coupled with application-specific fault tolerance that relies at least on an exception handling mechanism, which enables the software developer to specify the actions to be undertaken under the occurrence of application-specific and underlying runtime exceptions. We have then carried out research in the

[7] http://www-rocq.inria.fr/arles
[8] `http://www.newcastle.research.ec.org/c3ds/`

two following complementary directions towards assisting the architecting of dependable systems:

(i) Systematic aid in the development of middleware architectures for dependable systems;
(ii) Architecture-based exception handling.

The use of middleware is the current practice for developing distributed systems. Developers compose reusable services provided by proprietary or standard middleware infrastructures to deal with non-functional requirements. However, developers still have to design and implement middleware architectures combining available services in a way that best fits the application's requirements. In order to ease this task, we have customized the environment discussed in this paper with the following features [18]: (i) an ADL for modeling middleware architectures, (ii) a repository populated with architectural descriptions of middleware services, and (iii) automated support for composing middleware architectures out of available services according to target non-functional properties whose quality may be assessed both qualitatively and quantitatively.

As previously raised, it is necessary to complement fault tolerance support provided by the underlying middleware architecture, with support for exception handling. We have, thus, proposed a solution to architecture-based exception handling [16], which enhances exception handling implemented within components and connectors. Our solution lies in: (i) extending the ADL so as to enable the specification of required changes to the architecture in the presence of failures, and (ii) associated runtime support for enabling resulting dynamic reconfigurations.

6.3 Perspectives

The above results have been proven successful for assisting the architecting of dependable distributed systems that are closed, i.e., systems whose components depend on a single administrative domain and are known at design time. However, future distributed systems will increasingly be open, which raises new issues for making them dependable. In this context, we are undertaking research in the following directions:

(i) Architecting open distributed systems in a way that accounts for mobility, which requires support for the dynamic composition and quality assessment of architecture instances;
(ii) Design of fault tolerance mechanisms for open distributed systems considering that the systems span multiple administrative domains and hence cannot accommodate locking-based solutions as, e.g., enforced by transactional processing [39].

In general, the above calls for new solutions that allow the development of dependable systems that are highly dynamic and hence requires the integration of adaptive runtime support aimed at enforcing dependability.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE, 1994.

[3] M. C. Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures*. PhD thesis, University of Illinois, 1999.

[4] M. C. Astley and G. Agha. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering*, pages 1–9. ACM-SIGSOFT, November 1998.

[5] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. DURRA: A Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*, pages 83–94, March 1993.

[6] R. Butler and W. Ricky. The SURE Approach to Reliability Analysis. *IEEE Transactions on Reliability*, 41(2):210–218, June 1992.

[7] P. C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.

[8] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *ACM/IEEE Transactions on Networking*, 2001.

[9] D. Garlan, J. Kompanec, and P. Pinto. Reconciling the Needs of Architectural Description with Object-Modeling Notations. In *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML-00)*, 2000.

[10] R. Geist and K. Trivedi. Reliability Estimation of Fault Tolerant Systems : Tools and Techniques. *IEEE Computer*, 23(7):52–61, July 1990.

[11] R. Glass. *Software Reliability Guidebook*. Prentice-Hall, 1979.

[12] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10):576–583, October 1969.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[14] G. J. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[15] V. Issarny. Configuration-Based Programming Systems. In *Proceedings of SOFSEM'97: Theory and Practice of Informatics*, pages 183–200. Springer-Verlag, November 1997.

[16] V. Issarny and J-P. Banâtre. Architecture-based Exception Handling. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.

[17] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 207–214. IEEE, 1998.

[18] V. Issarny, C. Kloukinas, and A. Zarras. Systematic Aid for Developing Middleware Architectures. *Communications of the ACM (CACM)*, 45(6):53–58, 2002.

[19] S. C. Johnson. Reliability Analysis of Large Complex Systems Using ASSIST. In *Proceedings of the 8th Digital Avionics Systems Conference*, pages 227–234. AIAA/IEEE, 1988.

[20] R. Kazman, S. J. Carriere, and S. G. Woods. Toward a Discipline of Scenario-Based Architectural Engineering. *Annals of Software Engineering*, 9:5–33, 2000.

[21] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[22] J-C. Laprie. Dependable Computing and Fault Tolerance : Concepts and Terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, 1985.

[23] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and Analysis of Hardware and Software Fault-Tolerant Architectures. *IEEE Computer*, 23(7):39–51, 1990.

[24] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept 1995.

[25] M. Klein and R. Kazman and L. Bass and Carriere, S. J. and M. Barbacci and H. Lipson. Attribute-based architectural styles. In *Proceedings of the 1st IFIP Working Conference on Software Architecture (WICSA-1)*, pages 225–243, 1999.

[26] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. In *Proceedings of the 1st International Conference on Configurable Distributed Systems*, March 1992.

[27] J. Magee, J. Kramer, and D. Giannakopoulou. Behavior Analysis of Software Architectures. In *Proceedings of the 1st IFIP Working Conference on Software Architectures (WICSA-1)*, pages 35–49, 1999.

[28] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in CONIC. *IEEE Transactions on Software Engineering*, 16(5):663–675, June 1989.

[29] N. Medvidovic, D. S. Rosenblum, J. E. Robbins, and D. F. Redmiles. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, (to appear).

[30] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[31] R. Milner. *A Calculus of Communicating Systems*. Cambridge University Press, 1980.

[32] R. Milner. *Communicating and Mobile Systems: the pi-calculus*. Springer-Verlag, 1999.

[33] M. Moriconi, X. Qian, and A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[34] G. Myers. *Software Reliability - Principles and Practices*. John Wiley and Sons, 1976.

[35] NASA. Reliability Block Diagrams and Reliability Modeling. Technical report, NASA Glenn Research Center, May 1995. http://www-osma.lerc.nasa.gov/rbd/rbdtut.html.

[36] K. Nguyen and V. Issarny. Demonstration of Support for Architectural Design for Dependable SoS. CSDA2 report. Available at URL: `http://www.newcastle.research.ec.org/dsos/deliverables`.

[37] OMG. UML Semantics 1.3, 1997.

[38] M. Shaw, R. Deline, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[39] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. *Architecting Dependable Systems*, volume 2677 of *LNCS*, chapter Dependability in the Web Services Architecture. Springer-Verlag, 2003.

[40] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component and Message Based Ar-

chitectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, July 1996.

[41] A. Zarras and V. Issarny. Automating the Performance and Reliability Analysis of Enterprise Information Systems. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, 2000.

[42] A. Zarras, C. Kloukinas, and V. Issarny. *Architecting Dependable Systems*, volume 2677 of *LNCS*, chapter Quality Analysis of Dependable Systems: A Developer Oriented Approach. Springer-Verlag, 2003.

[43] A. Zarras, C. Kloukinas, V. Issarny, and K. Nguyen. *Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs*, IC2 report An Architecture-based Environment for the Development of DSoS. Available at URL: `http://www.newcastle.research.ec.org/dsos/deliverables`.