Quality Analysis of Dependable Systems: A Developer Oriented Approach

Apostolos Zarras[1], Christos Kloukinas[2], and Valérie Issarny[3]

[1] Computer Science Department, University of Ioannina, Greece,
zarras@cs.uoi.gr
[2] VERIMAG, Centre Équation, 2 avenue de Vignates, 38610 Gières, France,
Christos.Kloukinas@imag.fr
[3] INRIA, Domaine de Voluceau, B.P. 105, 78 153 Le Chesnay Cédex, France,
Valerie.Issarny@inria.fr

Abstract. The quality of dependable systems (DS) is characterized by a number of non-functional properties (e.g., performance, reliability, availability, etc.). Assessing the DS quality against these properties imposes the application of quality analysis and evaluation. Quality analysis consists of checking, analytically solving, or simulating models of the system, which are specified using formalisms like CSP, CCS, Markov-chains, Petri-nets, Queuing-nets, etc. However, developers are usually not keen on using such formalisms for modeling and evaluating DS quality. On the other hand, they are familiar with using architecture description languages and object-oriented notations for building DS models. Based on the previous and to render the use of traditional quality analysis techniques more tractable, this paper proposes an architecture-based environment that facilitates the specification and quality analysis of DS at the architectural level.

1 Introduction

Nowadays, there exists a clear trend for business, industry, and society to place increasing dependence on systems, consisting of the integration of numerous, disparate and autonomous components. Consequently, users have strong non-functional requirements on the quality of these systems. To satisfy these demands, quality analysis must be performed during the lifetime of the system. The quality of dependable systems (DS) is characterized by a number of attributes (e.g., security, performance, reliability, availability, etc.), whose values are, typically, improved by using certain means (e.g., encryption, load balancing, fault tolerance mechanisms). Two different kinds of quality analysis can be performed:

- *Qualitative analysis*, which aims at facilitating and verifying the correct use of certain means for improving the DS quality.
- *Quantitative analysis*, which aims at predicting the values of the quality attributes characterizing the overall DS quality.

The above kinds of quality analysis are complementary. In particular, the results of quantitative analysis are most probably affected by certain means, whose correct use is verified by the qualitative analysis. On the other hand, the use of certain means is guided by the results of the quantitative analysis at early design stage.

Performing quality analysis is not a new challenge and several techniques have been proposed and used for quite a long time [1–4]. Techniques for qualitative analysis are mainly based on theorem proving and model checking. Typically, models specifying the system's behavior are built using formalisms like CSP, CCS, Pi-Calculus, TLA, etc. Then, these models are checked against properties that must hold for the system to behave correctly. Techniques for quantitative analysis can be analytic, simulation, or measurement-based. Again models specifying the system's behavior are built using formalisms like Markov-chains, Petri-nets, Queuing- nets, etc. Certain model parameters (e.g., failure rates of the system's primitive elements) are obtained using measurement-based techniques. Then, the models are analytically solved, or simulated, to obtain the values of the attributes that characterize the overall system's quality. The main problem today is that building good quality models requires lots of experience and effort. Developers use Architecture Description Languages (ADLs) [5, 6], and object oriented notations (e.g., UML [7]) to design the system architecture. It is a common case that they are not keen on building quality models using CSP, CCS Markov chains, Petri-nets, Queuing-nets, etc. Hence, the ideal would be to provide the developers with an environment, which enables the specification of DS architectures and further provides adequate tool support that facilitates the specification of models suitable for DS quality analysis.

In this paper, we investigate the above issue and we present a developer-oriented, architecture-based environment for the specification and quality analysis of dependable systems. The specification of DS architectures is based on an extensible ADL, which is defined in Section 2. Section 3, then, presents an approach that facilitates the qualitative analysis of DS at the architectural level. Similarly, Section 4 discusses an approach that facilitates the quantitative analysis of DS at the architectural level. Finally, Section 5 concludes this paper with a summary of the contributions to DS quality analysis.

2 Architecture Description Language

2.1 Background and Related Work

Architecture description languages are notations enabling the rigorous specification of the structure and behavior of systems. ADLs come along with tools that facilitate the analysis and the construction of systems, whose architecture is specified using them. Several ADLs have been proposed in the past years and they are all based on the same base principles [6]. In particular, the structure of systems is specified using *components*, *connectors* and *configurations*. It is worth noticing that existing ADLs have concise semantics and are widely known and used in academia, but their use in the industry is quite limited. Industrials,

nowadays, tend to use 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. However, there is a primary concern regarding the imprecision of the semantics of UML. To increase the impact of ADLs in the real world, and to decrease the ambiguity of UML, we propose an ADL defined in relation to standard UML elements. Our main objective is the definition of a set of core extensible 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. Our effort relates to the definition of architecture meta- languages like ACME [5] and AML [8]. Our work also compares to the recent XML-based, extensible ADL [9]. Our approach can be the basis for the definition of a standard UML profile for ADLs, while [9] can be the basis for a complementary standard DTD used to produce textual specifications from graphical ADL models.

2.2 Basic Concepts

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) [4]; (iii) If the semantics of the identified element(s) match exactly, adopt the element(s) as a part of the core ADL language constructs.

A ***component*** abstracts a unit of computation or a data store. As discussed in the literature [10, 11], 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. Moreover, the UML Class element is often considered as the basis for defining architectural 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,

[4] 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.

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, which may be instantiated multiple times, and associated with other subsystems. Precisely, we define an ADL component as a UML Subsystem, that may provide and require standard UML interfaces. The ADL component is further 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.

A ***connector*** is an association representing the protocols through which components may interact. Hence, the natural choice for specifying it 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. 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"* [7]. 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.

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"* [7]. 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 [6]. Constraints that are specific to a style are introduced through the definition of a corresponding extension of the ADL configuration element, possibly combined with extension of the UML elements for component and connector definition.

2.3 Tools

The basic ideas described so far for the specification of software architectures are realized into a prototype implementation of the architecture-based development environment, which makes use of an existing UML modeling tool. More specifically, we use the Rational Rose tool [5] for the graphical specification of software architectures. The Rational Rose tool allows the definition of user specific add-ins that facilitate the definition and use of stereotyped elements. Given the aforementioned facility, we implemented an add-in that eases the specification of architectural descriptions using the elements defined in the previous subsection. 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 shall serve as input to the tools we use for qualitative and quantitative analyses of architectures. We further developed an OCL verifier that can be used to verify architectural constraints expressed in OCL. Note that we could have used an already existing verifier implemented in Java [6]. However, given that the expected complexity of our models is high, we preferred developing a more efficient implementation based on OCAML [7], which has been successfully used to efficiently develop large applications like the COQ theorem prover [8].

2.4 Example

To illustrate the use of our environment, we employ examples taken from a case study we are investigating in the context of the DSoS IST project [9]. 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 1 gives a screen shot of the actual architecture of the TA as specified using the UML modeling tool, which we customized. The TA comprises the *TravelAgent-FrontEnd* 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

[5] 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.

[6] http://www.db.informatik.uni-bremen.de/projects/USE

[7] http://www.caml.inria.fr/ocaml/

[8] http://www.coq.inria.fr

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

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 the specification relating to HTTP in Figure 1) 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.

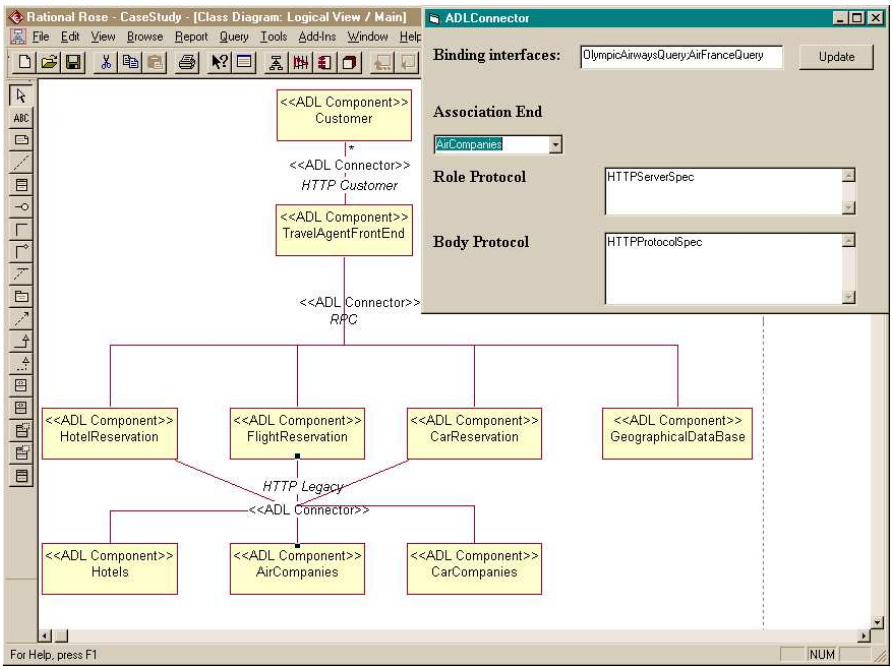

Fig. 1. The Architecture of the Travel Agent DS

3 Qualitative Analysis

3.1 Background and Related Work

Since the early work on ADL definition, there has been a significant effort for defining ADLs that ease the qualitative analysis of software architectures. Specifically, a number of existing ADLs come along with tools like theorem provers and model-checkers, allowing the specification of the functional behavior of components and connectors making up a system, and the verification of properties that must hold for the system against the system's functional behavior [1, 4]. In existing ADLs, the specification of the system's behavior is, typically, done using formalisms like logic or process algebras. Hence, to perform qualitative analysis, developers have to learn these formalisms and tools. They further have to derive mappings between the basic architectural concepts (e.g., components, connectors, ports, roles, etc.) they use to specify software architectures and the basic constructs provided by the formalism that is to be used for specifying the system's functional behavior (e.g., processes, channels, etc.), if these mappings are not already provided by the ADL itself. Neither of the previous tasks is straightforward for everyday developers who are very experienced and educated on the use of object-oriented modeling methods (e.g., UML methods), and several programming languages like C, C++, Java, but are not experts in logic and process algebras. An evidence of this is provided in the Web site of PVS [10], a well-known theorem-prover, where the following warning is given: *"...PVS is a large and complex system and it takes a long while to learn to use it effectively. You should be prepared to invest six months to become a moderately skilled user (less if you already know other verification systems, more if you need to learn logic or unlearn Z)..."*

A number of approaches have recently been proposed to try to alleviate the previous complexities towards rendering the use of qualitative analysis more tractable to nowadays developers. For instance, in [12], the authors propose a tool for model checking UML models. Developers have to specify these using state-chart diagrams, which are then used for generating models that serve as input to the SPIN model checker [13]. However, state-chart specifications of system behavior are quite low level and certainly not easy to produce. Take for instance the usual case where developers need to specify loops, procedure calls, synchronization and communication using state-charts. In this case, developers would prefer using a modeling language, which resembles a real programming language instead of using automata such as state-charts. We thus consider that the approach proposed in [12] is not a solution. Another approach is proposed in [14], which introduces stereotypes that are formally specified using Finite State Processes (FSP). FSP models may then be generated from UML models annotated with the stereotypes, for analysis using the Labelled Transition System Analyzer (LTSA) tool. This solution alleviates the limitations of the previous but it requires specifying formal models in FSP, which is not known by the vast majority of developers.

[10] http://pvs.csl.sri.com/whatispvs.html

From the above discussion and from a pragmatic point of view, it is not possible to completely avoid using a tool-specific formalism for performing qualitative analysis in the general case. Hence, our basic requirement becomes to integrate into our environment an existing tool for qualitative analysis, whose formalism for behavioral modeling is as natural as possible for the developers. This has led us to exclude, in the first place, theorem provers. In consequence, we are left with the option of integrating into our framework a model-checking tool. The second requirement, for rendering the qualitative analysis simpler, comprises providing an automated procedure that maps basic architectural concepts into basic constructs of the behavioral modeling formalism assumed by the selected model-checking tool. This allows the automated generation of formal behavioral models from DS architectural descriptions.

3.2 Basic Concepts

Support for the specification of the functional behavior of the basic architectural elements that constitute a DS, is provided by our environment as follows:

- ADL components are characterized by a property, called *"Body Behavior"*, whose value can be assigned to a textual specification, given in any behavioral modeling formalism, describing the components' behavior.
- UML interfaces provided/required by ADL components are characterized by a property, called *"Port Behavior"*, whose value describes in some textual specification, the particular protocol used at that point of interaction.
- ADL connectors are characterized by:
  - A property, named *"Body Protocol"* (see Figure 1), whose value specifies the role-independent part of the interaction protocol.
  - A set of properties (see Figure 1), named *"Role Protocol"*. Each one of these corresponds to an association end, i.e., a role. The value of each property specifies the role-dependant part of the interaction protocol represented by the connector.

3.3 Tools

We identified 3 widely used model checking tools that could be integrated into our environment, i.e., FDR2 [11], SMV [12] and SPIN [13]. Among them, we have chosen SPIN because: (i) it is based on a C-like language for modeling system behavior, which is more familiar to DS developers compared to other modeling languages, and (ii) it has built-in channels, i.e., constructs used for modeling message-passing, with which we can easily model parts of the ADL connectors. A model in the SPIN modeling language, i.e., PROMELA, consists of a number of independent processes, i.e., each one has its own thread of execution, which communicate either through global variables or through special communication channels by message-passing, as is done in CSP, at least in its machine

[11] http://www.formal.demon.co.uk/fdr2manual
[12] http://www.cs.cmu.edu/~modelcheck

readable version. Therefore, the mapping of our basic architectural elements to the constructs of PROMELA can be done in a way analogous to the mapping used by the Wright ADL for CSP [1]. In particular in [1], for each component, connector, port/interface and role, a corresponding process is generated. Each generated process shall communicate with the rest through channels generated as prescribed by the configuration of the DS. However, such a mapping results in the generation of a large number of processes and requires a substantial amount of resources for model checking.

Table 1. Generating PROMELA Models

Component	For each component *c*: – Create a PROMELA process type, *"proctype"*, named after the component, whose behavior is given by the value of *"Body Behavior"* – For each port *p* of *c*, create an *"inline"* procedure whose name is the catenation of the component's and the port's name, i.e., *c_p*. This procedure contains the *Port Behavior* of the respective port *p*. For interacting with its environment, *c_p* uses a channel named after the port's name, i.e., *p*.
Connector	For each connector *c*: – Create a *"proctype"*, named after the connector, whose behavior is given by the value of *"Body Protocol"*. Unlike the processes corresponding to ADL components that take no arguments, these processes receive as arguments at initiation time the channels they will be using for their respective roles. These channels are named after the roles themselves.
Configuration	Create a special process called *"init"* in PROMELA, which will be responsible for instantiating the rest of the architecture. More specifically: – The *"init"* process creates as many instances of the processes corresponding to particular ADL components, as there are instances of these components in the configuration. – Afterwards, it does the same for each instance of an ADL connector but it uses the attachments of component ports to connector roles to deduce the specific channels that should be passed as arguments to the processes corresponding to the connector.

To alleviate the above problem, we have chosen to generate independent processes for each component and connector specified in a DS architectural description, while for each port and role we generate PROMELA inline procedures. This inline procedure construct of PROMELA allows us to define new functions that can be used by processes, but do not introduce their own threads of execution. In this manner, we keep to a minimum the number of different processes that the model-checker will be asked to verify, thus enabling the verification of

large architectures. Then, for each port of an ADL component we declare in the PROMELA description of the component, a communication channel named after that port. This channel will be used by the process related to the ADL component for communicating through that specific port. Since ports of ADL components are bound to specific roles of ADL connectors, their channels are passed as arguments to the processes created for these connectors, at the time of their initiation. Thus, messages sent from a process of an ADL component at a channel corresponding to a port of it, will be received by a process of an ADL connector. Similarly, messages sent from a process of an ADL connector to a channel it has received as argument at initiation time, will be in fact received by a process of an ADL component, whose port was mapped to that channel. Even though the proposed mapping may seem as depriving the architect from the possibility to describe complex cases, e.g., multi-threaded components, it is not so. Indeed, it is always possible to describe a component as a composite one, i.e., one that consists of a number of simpler components and connectors, which will be subsequently modeled as a number of independent processes. The steps that are followed for generating a complete PROMELA model from an architectural description are given in Table 1.

3.4 Example

To exemplify the qualitative analysis of DS, we get back to the TA case study. A typical property that is often required over RPC and HTTP connectors is for reply messages to be received by the client in the order it sent the corresponding request messages. Meeting the previous is usually under the responsibility of the connector realization, possibly in association with the server. For instance, the HTTP/1.0 and HTTP/1.1 realizations of the HTTP connector differ in that the latter supports persistent connections and allows pipelining of request messages, which leads to explicitly require for the server to ensure that it sends back reply messages in the order it received the corresponding request messages. In the TA case study, we consider both realizations of HTTP. Moreover, we consider two realizations of the *TravelAgentFronEnd* component and the rest of the Web servers supporting the hotel, car and flight reservations. In the first case, the components process HTTP requests sequentially, while in the second case they use multi-threading to process multiple HTTP requests in parallel.

The processes corresponding to the RPC and the HTTP/1.0 connectors are similar in functionality; they iterate constantly, doing the following: (i) they receive a request from the component assuming the role of the RPC caller/Web client, (ii) deliver it to the component assuming the role of RPC callee/Web server, (iii) receive the reply from the callee/server, and (iv) forward it to the caller/client. The HTTP/1.1 connector works differently; it can receive multiple requests and forward them to the callee/server, or decide to read one (or more) replies and deliver them to the caller/client. For instance, we get the following specification provided for the RPC connector:

Role Protocol: *caller(RPC_channel, request, reply)*
{*RPC_channel ! request; RPC_channel ? reply* }
Role Protocol: *callee(RPC_channel, request, reply)*
{*RPC_channel ? request; RPC_channel ! reply* }
Body Protocol:
{*Msg request, reply;*
do::
caller ? request; callee ! request;
callee ? reply; caller ! reply
od }

The *Customer* component initiates requests to the *TravelAgentFrontEnd* and waits for responses. The reservation components get requests from the RPC connector and diffuse them, through the HTTP connector, to the existing Web servers supporting the hotel, car and flight reservations. In the sequential versions of the *TravelAgentFrontEnd* component and of the Web servers supporting reservations, the corresponding PROMELA processes process each request and send the corresponding reply before serving a new request. Their concurrent versions are based on a pool of threads. For illustration, we get the following specification provided for the Web servers that handle requests sequentially:

Port Behavior: *HTTP_Request(HTTP_channel, request, reply)*
{ *HTTP_channel ? request; HTTP_channel ! reply* }
Body Behavior:
{*chan HTTP_channel ; Msg request, reply;*
do::
HTTP_Request(HTTP_channel, request, reply)
od }

Four different PROMELA models were generated. These models result from the combination of the different HTTP and Web server versions. More specifically, for all components and connectors, corresponding processes were generated. The realizations of the generated processes consist of the *Body Behavior* and *Body Protocol*, for components and connectors respectively. These processes were connected via channels generated for each port of the various components, according to the configuration given in Figure 1. SPIN was then used to assess the TA against ordered delivery of reply messages to the customers for all the 4 cases resulting from the combination of the different HTTP and Web server versions. Checking of the models resulted in identifying an erroneous architecture for the TA that is the case where Web components interact via HTTP/1.1 and the Web servers handle concurrently the request messages. The full source code of the TA PROMELA model used in this case study can be found in [15].

4 Quantitative Analysis

4.1 Background and Related Work

Pioneer work on the quantitative analysis of software systems at the architectural level includes Attribute-Based Architectural Styles (ABAS) [16]. 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 using these types of architectural elements, and patterns describing the data and control interaction among them. An ABAS is an architectural style, which additionally provides modeling support for the quantitative analysis of a particular quality attribute (e.g., performance, reliability, availability). 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, mean response time).
- *Quality attribute stimuli*, i.e., events affecting the quality attribute of the system (e.g., failures, service requests).
- *Quality attribute parameters*, i.e., architectural properties affecting the quality attribute of the system (e.g., faults, redundancy, thread policy).
- *Quality attribute models*, i.e., traditional models that formally relate the above elements (e.g., a Markov model that predicts reliability based on the failure rates and the redundancy used, a Queuing network that enables predicting the system's response time given the rate of service requests and performance parameters like the request scheduling and the thread policies of the various system elements).

In [17], 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, which roughly constitutes the specification of a service profile. ATAM has been tested for analyzing quality attributes like performance, availability, modifiability, and real-time. In all these cases, quality attribute models (e.g., Markov models, queuing networks) are manually built given the specification of a set of scenarios and the ABAS- based architectural description. However, in [17], 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, nowadays, there is a constant additional requirement for doing things fast and easy.

Our environment supports the automated generation of quality attribute models from architectural descriptions embedding quality attributes. In particular, the environment currently supports the generation of performance and reliability models aimed at analysis tools that have been recognized successful for handling complex models associated with real systems. Note that there is no unique way to model systems. A model is built based on certain assumptions. Thus, the model generation procedures supported by our environment are customizable. Customization is done according to certain assumptions that can be made by the developer for the quality stimuli and parameters affecting the value of the particular quality attribute that is assessed. Due to the lack of space, we provide hereafter details regarding only the case of reliability. The interested reader is referred to [18] and [15] for details regarding the case of performance,

where the former concentrates on performance analysis of workflow-based systems.

4.2 Basic Concepts

To perform quantitative analysis, we have to specify a service profile, i.e., a set of scenarios, describing how the inspected system is used. In our environment, scenarios are specified using UML collaboration diagrams. A scenario then specifies the interactions among a set of component and connector instances, structured as prescribed by the configuration of the inspected system. Moreover, the definitions of the base ADL elements have been extended to support the specification of reliability measures, parameters, and stimuli, as defined below.

The basic *reliability measure* is the probability that a scenario successfully completes within a given time duration. A scenario may fail if instances of components, nodes [13], and connectors used in it, fail because of faults causing errors in their state. The manifestations of errors are failures. Hence, faults are the basic *parameters*, associated with components/connectors/nodes, which affect the reliability of an inspected system. Failures are the *stimuli*, associated with components/connectors/nodes, causing changes in the value of the reliability measure. According to [19], faults and failures can be further characterized by the properties given in Tables 2 and 3. Different combinations of the values of these properties can be used to customize properly the generation procedure of quality attribute models, which is detailed in Subsection 4.3.

Except for faults and failures, another parameter affecting reliability is redundancy. Redundancy schemas can be defined using the base ADL constructs defined in Section 2. More specifically, a redundancy schema is a configuration of redundant architectural elements, which behave as a single fault tolerant unit. According to [20], a redundant schema is characterized by 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 properties characterizing a redundancy schema are summarized in Table 4. A re-configurable/repairable redundancy schema may be characterized by additional properties (e.g. repair rate, number of spares, state of the spares), whose values reflect the particular re-configuration/repair policy used.

Table 2. Properties of Failures

Failure Properties	**Range**	**Associated ADL Element**
domain	time/value	Component/Connector/Node
perception	consistent/inconsistent	

[13] Since an ADL component is by definition an extension of the standard UML Subsystem element, it is associated with a set of UML nodes on top of which it executes.

Table 3. Properties of Faults

Fault Properties	Range	Associated ADL 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 4. Properties of Redundancy Schemas

Redundancy Properties	Range	Associated ADL 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	

4.3 Tools

The quantitative analysis of DS is supported by our environment with automated procedures, which take as input, architectural specifications defined using the basic concepts discussed so far, and generate traditional quality attribute models. The specific tool integrated into our environment for reliability analysis is called SURE-ASSIST [21]. The tool calculates reliability bounds given a state space model describing the failure and repair behavior of the inspected system. The tool was selected because it is very highly rated compared to other reliability tools [2] and because it is available for free. However, the automated support provided by our environment for reliability analysis can be coupled with any other tool that accepts as input state space models.

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 state of its constituent elements. Hence, it can be seen as a composition of sub states, each one representing the situation of a constituent element. A state is constrained by the range of all possible situations that may occur. A state range can be modeled as a composition of sub state ranges, constraining the state of the elements that constitute the system. 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 equals to

the failure rate of the component. The specification of large state-space models is often too complex and error-prone. The approach proposed in [22] alleviates this problem. In particular, instead of specifying all possible state transitions, the authors propose specifying the following: (i) the state range of the system, (ii) transition rules between sets of states of the system, (iii) the initial state of the system, and (iv) a death state constraint. In a transition rule, the source and the target set of states are identified by constraints on the state range (e.g., if the system is in a state where more than 2 subsystems are operational, then the system may get into a state where the number of subsystems is reduced by one). A complete state space model can then be generated using the algorithm described in [22]. Briefly, the algorithm takes as input an initial system state. Then, the algorithm applies recursively the set of transition rules. During a recursive step, the algorithm produces a transition to a state derived from the initial one. Depending on the rule that is applied, in the resulting state, one or more elements are modeled as being failed, or operational, while in the initial state they were modeled as being operational or failed, respectively. If the resulting state is a death state, the recursion ends.

Complete state space models are automatically generated from DS architectural descriptions embedding the specification of reliability stimuli and parameters, by following the steps below.

First, a state range definition for each collaboration belonging to a given service profile is generated. The state of a collaboration is composed of the states of the component and connector instances used within the collaboration 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 range of states for a component/connector/node depends 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. 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. The range of states for a component further depends on the kind of redundancy used. Again, the generation procedure is customized accordingly.

After generating the state range definition for a collaboration *collab*, the step that follows comprises the generation of transition rules for components/ connectors/nodes used in the collaboration. 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 5. What is left at this point is to generate the definition of the initial state of the collaboration, and the definition of the death state constraint. The initial state is a state where all of the elements used in the collaboration are operational. A collaboration is in death state if any of the architectural elements used within 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 collaboration. More specifically, the base predicate for a component,

connector, or a node states that the element is in a FAILED state. 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 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.

Table 5. Transition Rules for Permanent Faults

ADL Element	**Rule**
Component	For all instances of primitive components, c: – If *collab* is in a state where c is in an OPERATIONAL state st, then *collab* 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 3). For all instances of composite components, c: – If *collab* is in a state st where c is OPERATIONAL, then *collab* 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 *collab* is in a state st where rc is OPERATIONAL, and the number of failed redundant component instances if fc, then *collab* 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. The rate of these transitions is equal to the fault arrival rate specified for t. 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 *collab*: – If *collab* is in a state st where a node n is in an OPERATIONAL state, then *collab* 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, $n.Faults.arrival\text{-}rate$.

4.4 Example

To demonstrate the automated quantitative analysis detailed in the previous subsection, we use the TA case study. The goal of our analysis is not to obtain precise values of the reliability measure since this would require to precisely model the Internet, which in general is considered as rather unrealistic [23]. 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 2 gives 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 *TravelAgentFrontEnd* 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 are 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 2 shows the detailed specification of the reliability stimuli and parameters that are given for the *FlightReservation* component.

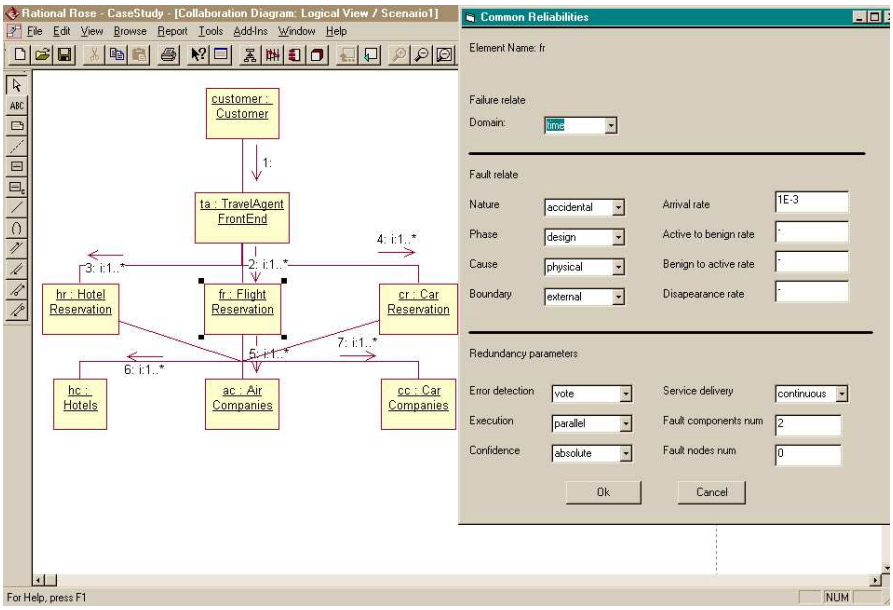

Fig. 2. A generic scenario for TA

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 k different versions of the *HotelReservation* component. Upon the instantiation of the schema, k component instances are created, one of each version. These instances

execute in parallel and are deployed on k different nodes. The second schema contains k versions of the *FlightReservation* component, the instances of which are also deployed on the k nodes, on top of which the instances of the *HotelReservation* component execute. Finally, the last schema contains k 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 and reliability parameters and, three complete state space models were generated and analytically solved. The results obtained are summarized in Figure 3. For further detail about the scenario, including complexity of the generated state space models, the interested reader is referred to [15].

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.

5 Conclusion

In this paper, we presented an environment for the quality analysis of DS. The overall design and realization of our environment is guided by the needs of its current and potential users, imposing the simplification of certain extremely important and inevitable development activities related to the quality analysis and

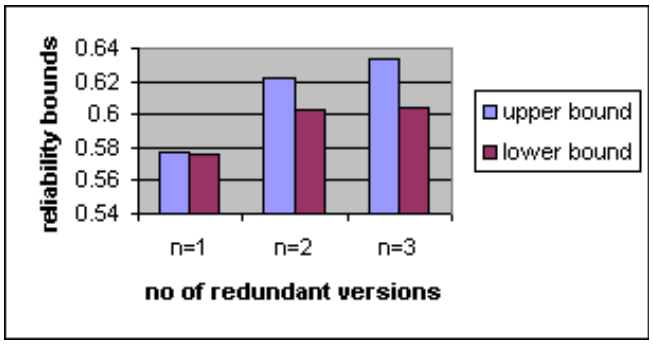

Fig. 3. Results produced by the reliability analysis of TA

assurance of the DS. 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 proposed environment relies on an architecture description language for the specification of DS 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. In the case of quantitative analysis, a high level of automation is achieved since the developer is requested only to specify quality attributes within the architecture, from which formal models processed by existing analysis tools are automatically generated. On the other hand, in the case of qualitative analysis, the automation that can be achieved is more limited since the developer is requested to formally specify the behavior of architectural elements. This led us to carefully select an existing qualitative analysis method and associated tool whose use is natural to the developers.

Having reached into a stable prototype implementation of our environment, we now concentrate on testing it on real world case studies. So far, we used it successfully in the context of the DSoS DST project for the quality analysis case of the Travel Agent system. Parts 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 [14] project for the performance and reliability analysis of workflow based dependable systems [18].

Acknowledgments

This work has been partially funded by the IST DSoS project.

[14] http://www.newcastle.research.ec.org/c3ds/

References

1. Allen, R., Garlan, D.: Formalizing architectural connection. In: Proceedings of the 16th ACM-SIGSOFT-IEEE International Conference on Software Engineering (ICSE'94). (1994) 71–80
2. Geist, R., Trivedi, K.: Reliability estimation of fault tolerant systems: Tools and techniques. IEEE Computer **23** (1990) 52–61
3. Kobayashi, H.: Modeling and Analysis: An Introduction to System Performance Evaluation Methodology. Addison-Wesley (1978)
4. Magee, J., Kramer, J., Giannakopoulou, D.: Behavior analysis of software architectures. In: Proceedings of the 1st IFIP Working Conference on Software Architectures (WICSA-1). (1999) 35–49
5. Garlan, D., Monroe, R., Wile, D.: ACME: An architectural description interchange language. In: Proceedings od CASCON'97. (1997)
6. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
7. OMG: UML semantics 1.3 (1997)
8. Wile, D.: AML: An architecture meta-language. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE-99). (1999)
9. Dashofy, E., Van Der Hoek, A., Taylor, R.: An infrastructure for the rapid development of XML-based architecture description languages. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02). (2002) 266–276
10. Garlan, D., Kompanec, J., Pinto, P.: 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)
11. Medvidovic, N., Rosenblum, D.S., Robbins, J.E., Redmiles, D.F.: Modeling software architectures in the unified modeling language. ACM Transactions on Software Engineering and Methodology ((to appear))
12. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99). (1999) 255–258
13. Holzmann, G.J.: The SPIN model checker. IEEE Transactions on Software Engineering **23** (1997) 279–295
14. Kaveh, N., Emmerich, W.: Deadlock detection in distributed object systems. In: Proceedings of the 8th European Software Engineering Conference (ESEC) / 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). (2001)
15. Zarras, A., Kloukinas, C., Issarny, V., Nguyen, V.K.: An Architecture-based Environment for the Development of DSoS. In: (IC2 Report: Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs) Available at URL: http://www.newcastle.research.ec.org/dsos/deliverables.
16. Klein, M., Kazman, R., Bass, L., Carriere, S.J., Barbacci, M., Lipson, H.: Attribute-based architectural styles. In: Proceedings of the 1st IFIP Working Conference on Software Architecture (WICSA-1). (1999) 225–243
17. Kazman, R., Carriere, S.J., Woods, S.G.: Toward a discipline of scenario-based architectural engineering. Annals of Software Engineering **9** (2000) 5–33
18. Zarras, A., Issarny, V.: 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)

19. Laprie, J.C.: Dependable computing and fault tolerance: Concepts and terminology. In: Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15). (1985)
20. Laprie, J.C., Arlat, J., Beounes, C., Kanoun, K.: Definition and analysis of hardware and software fault-tolerant architectures. IEEE Computer **23** (1990) 39–51
21. Butler, R.W.: The SURE approach to reliability analysis. IEEE Transactions on Reliability **41** (1992) 210–218
22. Johnson, S.C.: Reliability analysis of large complex systems using ASSIST. In: Proceedings of the 8th AIAA/IEEE Digital Avionics Systems Conference. (1988) 227–234
23. Floyd, S., Paxson, V.: Difficulties in simulating the internet. ACM/IEEE Transactions on Networking (2001)