	DSoS ***IST-1999-11585*** ***Dependable Systems of Systems***

State of the Art Survey

Report : Deliverable BC2

Report Delivery Date: 30 September 2000

Classification: Public Circulation

Contract Start Date: 1 April 2000 **Duration:** 36m

Project Co-ordinator: University of Newcastle upon Tyne

Partners: DERA, Malvern – UK; INRIA, Rocquencourt – France; LAAS-CNRS, Toulouse – France; TU Wien – Austria; Universität Ulm – Germany; LRI, Orsay - France

Project funded by the European Community under the "Information Society Technology" Programme (1998-2002)

LAAS-CNRS Report No. 00353

Technical Report CS-TR-708, University of Newcastle upon Tyne

List of Authors

Jean Arlat	LAAS-CNRS, Toulouse, F
Jean-Charles Fabre	LAAS-CNRS, Toulouse, F
Valérie Issarny	INRIA, Rocquencourt, F
Mohamed Kaâniche	LAAS-CNRS, Toulouse, F
Karama Kanoun	LAAS-CNRS, Toulouse, F
Christos Kloukinas	INRIA, Rocquencourt, F
Bruno Marre	LRI, Orsay, F
Eric Marsden	LAAS-CNRS, Toulouse, F
David Powell	LAAS-CNRS, Toulouse, F
Alexander Romanovsky	University of Newcastle upon Tyne, UK
Pascale Thévenod-Fosse	LAAS-CNRS, Toulouse, F
Hélène Waeselynck	LAAS-CNRS, Toulouse, F
Ian Welch	University of Newcastle upon Tyne, UK
Irfan Zakkiudin	DERA, Malvern, UK
Apostolos Zarras	INRIA, Rocquencourt, F

Table of Contents

Table of Figures

State of the Art Survey

Jean Arlat[1], Jean-Charles Fabre[1], Valérie Issarny[2], Mohamed Kaâniche[1], Karama Kanoun[1], Christos Kloukinas[2], Bruno Marre[3], Eric Marsden[1], David Powell[1], Alexander Romanovsky[4], Pascale Thévenod-Fosse[1], Hélène Waeselynck[1], Ian Welch[4], Irfan Zakkiudin[5], Apostolos Zarras[2]

[1]LAAS-CNRS (Toulouse, F), [2]INRIA (Rocquencourt, F), [3]LRI (Orsay, F), [4]University of Newcastle upon Tyne (UK), [5]Dera (Malvern, UK)

Introductory Remarks

This report provides a state of the art survey related to the work package on architecture and design, and to the work package on validation. The report is organised into 5 chapters, the first 3 relate to DSoS work on architecture and design, and the last 2 to work on validation. These chapters may be read independently, and corresponding bibliographical references are given separately at the end of the report. The 5 chapters address the following areas:

1. **Architecture and Design:** This chapter focuses on work in the field of architecture-based development of software systems. It discusses proposed notations for the rigorous description of software architectures, together with associated methods and tools for the design, analysis and building (construction) of software systems from their architectural description.

2. **Mechanisms for Enforcing Dependability of Services:** This chapter addresses both the essential mechanisms for enforcing dependability of services, and the architectural concepts for the design and implementation of dependable systems of systems.

3. **Wrapping Technology:** This chapter summarises the work done in the area of wrapping technology with respect to both solving architectural mismatch issues, and protecting components against erroneous interacting components.

4. **Validation Techniques:** This chapter provides an overview of work in the field of validation, addressing related methods based on testing, fault injection, and model checking.

5. **Dependability Evaluation of Large Systems:** This chapter surveys the two complementary approaches that can be undertaken for the dependability evaluation of systems of systems, i.e., analytical modelling and measurement-based assessment.

Chapter 1 – Architecture and Design

Valérie Issarny, Christos Kloukinas, Apostolos Zarras *(INRIA)*

1.1 Introduction

Easing the development of complex software systems such as the ones targeted in the DSoS project calls for addressing their intrinsic complexity that comes from a number of key concerns:

- Means should be provided for reasoning about the system's functional properties and quality (e.g., dependability, extensibility, performance, scalability, ...) throughout the system development process, so as to enable undertaken design and implementation decisions to be checked against the requirements specification, and discovering misconceptions as early as possible.
- The software development process must reduce the costs associated with the development, maintenance, and support of the software system.
- Reducing the aforementioned costs has in particular led to the notion of component-based software systems. This notion enables developing the system out of existing software components should they be either COTS components or legacy systems, and hence should be of great value for developing systems of systems. However, it necessitates means for easing the robust integration of such components, which were not designed for use within the specific system.

Dealing with the above issues requires adequate support for specifying, analysing, simulating, building, documenting, and visualising the software system, in a way that is comprehensible by various stakeholders (e.g., designers, developers, integrators, users, ...). The architecture of the software system is a convenient base vehicle to tackle this need: it prescribes the set of significant decisions regarding the gross organisation of the system by abstractly characterising the system's composing software elements [Perry & Wolf 1992, Shaw & Garlan 1996]. However, for the description of the system's software architecture to actually bring aid in the software development process, it is necessary to provide notations, methods and tools that enable describing the system's software architecture in a way that allows systematic analysis, verification and further elaboration of the developed system. Towards that goal, there has been a number of research works undertaken in the software architecture community since the early 90's, which is outlined in the following sections. Section 1.2 discusses further architecture-based development of software systems giving its base ground and the issues it raises. Section 1.3 focuses on the notations that have been introduced for the precise description of software architectures and their processing. Section 1.4 concentrates on architecture-based development of distributed systems, which now requires accounting for component and middleware technologies. Finally, Section 1.5 gives concluding remarks, highlighting some of the open research issues that remain when concerned with the development of dependable systems of systems.

1.2 Architecture-based Development of Software Systems

The software architecture of a system provides a very abstract description of the system by focusing on its structure and abstracting away implementation detail. Hence, architecture-based software development is concerned with assembling well-defined architectural elements, according to some guiding principles, in order to satisfy the major functional and quality requirements of the target system. The bases of software architecture elaboration then lie in the abstraction of the architectural elements, their composition, and the definition of architectural styles that guide the development process. The following subsection defines the conceptual building blocks of software architectures that resulted from this concern, and is followed by a discussion on how to effectively exploit those building blocks for the thorough development of robust software systems.

1.2.1 Conceptual Building Blocks

It is now accepted by the vast majority of the software architecture community that the description of a system architecture should be based on the following building blocks:

- ***Components*** that abstractly characterise units of computation or data stores. In general, the specification of a component gives the behavioural specification of the component together with the component's interfacing points (i.e., both provided and required interfaces, which are often referred to as ***ports***) with the other architectural elements.

- ***Connectors*** that abstractly characterise composition patterns among components. A connector thus prescribes the interaction protocol that takes place among the components that are composed through it. Hence, a connector specification gives the behavioural specification of the connector together with the connector's interfacing points (often referred to as ***ports*** or ***roles***) with the other architectural elements.

- ***Configurations*** that define the structures of (sub-)systems by composing collections of component instances through bindings *via* connector instances. A system's software architecture is then defined as a configuration together with the component and connector types that are instantiated within the configuration. Notice further that a configuration may be integrated within a system, in which case it corresponds to either a component or a connector for which the architectural design has been elaborated.

While the above notions are rather straightforward to interpret at a first glance, it appears that the distinction between component and connector has been a source of misunderstanding and that the notion of connector is not well established. In particular, this has led some researchers to retain a single kind of architectural element, i.e., the component, and to define configurations in terms of direct bindings among components (e.g., [Magee *et al*. 1995]). However, the notion of connector is of primary interest when concerned with the development of complex distributed systems. It enables abstracting complex distributed system management functions as for instance embodied in middleware infrastructures, while separating them from the application-specific components composing a given software system. Works in the software architecture community that do consider connectors as first-class entities have mainly focused on the specification of connectors with respect to the interaction pattern they enforce among application-specific components (e.g., [Allen & Garlan 1997]). Complementary aspects of software connectors such that the functions they embody towards improving the overall system quality through dedicated mechanisms, has also been examined from the perspective of middleware synthesis out of the application's non-functional requirements (e.g., [Issarny *et al.* 1998a]); however, this approach assumes a single type of interaction pattern, i.e., remote procedure call. Thus, the clear understanding and detailed specification of software connectors is still at an early stage and calls for further investigation. This first requires a survey and a classification of relevant work in the areas of operating, distributed, and communication systems such as the one recently presented in [Mehta *et al.* 2000].

The aforementioned base architectural building blocks serve structuring the software system. However, they are not sufficient per se to guide the elaboration of the system software architecture, which should build upon previous knowledge about how related systems or architectural elements were developed. This issue is captured through the notion of ***architectural style*** that provides means for exploiting commonalities between systems and for leveraging analysis and implementation efforts [Garlan *et al*. 1994a, Allen 1997]. Basically, an architectural style defines a set of properties that are shared by the configurations that are members of the style; such properties may prescribe the kinds of software connectors that may be used (e.g., when the underlying infrastructure is partly fixed), and topology constraints (e.g., connectors may not be directly connected). The notion of architectural style is a significant aid in the development process in several respects. For instance, it helps cope with needed architectural evolution as encountered in the development of software product families (e.g., see [Kuusela 1999] for issues raised in the development of a specific family) by setting the architectural commonalities among the members of a given family. Another area of relevance is the one of exploiting knowledge on how to enforce a given quality property through the adequate structuring of the architecture (e.g., see [Saridakis & Issarny 1999] for the treatment of fault-tolerance). This latter aspect relates to the wider issue of describing solutions for specific problems reoccurring in systems development, which may be addressed by applying corresponding design and architectural patterns [Buschmann *et al.* 1996]. Some promising results have further been achieved in this area from the standpoint of developing patterns assisting in applying different fault tolerance techniques: exception handling [Garcia *et al*. 2000], Coordinated Atomic actions [Beder *et al*. 2000] *etc*. Specifying such patterns using architectural styles should ease further their effective reuse and the analysis of the system architectures based on them.

1.2.2 Issues in Effectively Supporting Architecture-based Development

Effectively supporting architecture-based development requires elaborating and maintaining models of the system architecture under development since models constitute the primary means to help understanding the problems that are addressed and their solutions. Such models are necessary in all the phases of the (possibly iterative) development process, which are typically the requirements, design and analysis, implementation, testing, and deployment phases. It is thus needed to define adequate notations associated with the specification of the architectural building blocks, for the sake of architecture-based system modelling. These notations should further come along with methods and tools for assisting in the elaboration of the model as well as for assessing it during the various phases of the development process.

Given that the elaboration of a system architecture involves various stakeholders and development dimensions, it is advisory to enforce the separation of concerns principle within the development process and to provide different views (or perspectives) of the architecture model. This is in particular highlighted in [Kruchten 1995] where the following "4+1" views are introduced: the logical view that is the functional model of the design; the process view that focuses on the concurrency and synchronisation aspects of the design; the physical view that gives the mapping of the software onto the hardware platform; the development view that describes the static organisation of the software in its development environment; the "+1" view is the one concerned with constraining the overall system architecture with selected use cases (or scenarios).

Work in the software architecture community has been on providing effective solutions to architecture-based system modelling. Proposed solutions may further be divided into two areas at this stage of our presentation, depending on whether they are concerned with the definition of notations for describing software architectures or with the overall architecture-based development process. In the former category, we meet the definition of ***Architecture Description Languages*** (ADL) that mainly originate from academia and research laboratories; these are discussed in the next section. In the latter category, we find development processes that are mainly proposed in industry such as the one used at Siemens [Borrmann & Newberry-Paulish 1999], and the RUP process [Kruchten 1999] from Rational that is based on the "4+1" view model of software architecture sketched above and that is supported by the ROSE development environment[1]. Notice that rather than relying on novel notations for architecture description, these processes rely on a standard notation, namely UML [OMG 1997a, OMG 1997b]. We defer until the next section a discussion about the respective advantages of either using a novel ADL, or not for the specification of architecture-based system models.

In addition to the above research directions, we find studies that focus more specifically on the *a posteriori* architecting of existing software systems. While it is now common concern (if not practice) to provide well-defined architectures for software systems, there exist systems for which the information is not readily available and that is necessary for making the system evolved. To help in this process, methods and tools are proposed for both recovering the system's architecture [Bratthall & Runeson 1999, Guo *et al*. 1999] and for modifying it [Pree & Koskimies 1999]. Close to this concern and in fact addressed in the aforementioned references, comes the issue of dealing with software product families. As previously mentioned, defining an architectural style eases tackling the affordable evolution of a software product. However, this only partly addresses the problem [Perry 1998], since all the future needed evolutions of a software system cannot be anticipated, and may later call for system rearchitecting. Although the development of systems of systems is concerned with the integration of existing systems, we do not intend addressing their internal modification but rather explore how to compose them with additional architectural elements so as to make them fit within the overall system. Hence, we do not further address the recovery of architectures and their evolution in the remainder of this chapter.

1.3 ADL as the Supporting Notation

Defining notations for the description of software architectures has been one of the most prominent areas of research in the software architecture community since the early papers about the need for a disciplined elaboration of system architectures [Schwanke *et al*. 1989, Shaw 1989, Perry & Wolf 1992]. Regarding the overall development process, ADLs that have been proposed so far are mainly concerned with architecture modelling during the analysis and design phase. Recently, close

[1] http://www.rational.com

coupling with a requirements engineering method has been proposed in [Riemenschneider *et al*. 2000]. Notice further that some existing ADLs enable deriving system implementation and deployment (referred to as system construction in the following) given available implementation for the system's primitive components and connectors. Hence, although all the above efforts are conducted independently, we may envision the provisioning of architecture-based development environments that do ease building robust software systems since, as discussed below, one of the contributions of the software architecture field lies in providing methods and tools for the thorough design and assessment of the system's architecture.

A major objective in the definition of ADLs is to provide associated CASE tools, which enable automating (at least partly) tasks underpinning the development process. In particular, a special emphasis has been put on the usage of formal methods and associated tools, whose application to the analysis of complex software systems is eased due to the focus on the system's architecture that is abstract and concise. In that context, the description of software architectures has first been examined without introducing any new specific notations but rather using as is existing formal notations and exploiting their support for achieving analyses. For instance, we meet the following works in the above category. The CHAM (Chemical Abstract Machine) formalism has been used for describing the structure and abstract behaviour of a specific architecture (i.e., the one of a compiler) in [Inverardi & Wolf 1995]. The Z language has been used to characterise architectural styles and has later led to define a framework for such characterisations so as to enable comparing styles sharing a common semantic model [Abowd *et al*. 1995]. Logic has been used in [Moriconi *et al*. 1995] for supporting correct stepwise refinement of configurations. Graph grammars are exploited in [Le Metayer 1996] for enabling constrained architecture evolution. The advantages of introducing ADLs over the above works are obvious with respect to leveraging the elaboration of software architectures. An overview of existing ADLs is provided hereafter, and is followed by a discussion about the relation of such notations with the UML standard software modelling language that is becoming a major player in industry. We conclude this section by sketching some ongoing research work in the software architecture community, focusing more specifically on work relevant to the development of dependable systems of systems.

1.3.1 Architecture Description Languages

Basically, an ADL offers notations for the characterisation of the architectural building blocks that were introduced in the previous section. Hence, any ADL provides means to describe the structure of a system, in general both in a textual and a graphical form. It is not our intention to provide an exhaustive list of existing ADLs in the following. Instead, we refer the interested reader to existing surveys and in particular the ones presented in: [Allen 97] that further compares ADLs with other development languages; [Issarny 97] that focuses on ADLs based on the use of formal methods; and [Medvidovic & Taylor 2000] that proposes a classification and comparison framework for ADLs. Here, we concentrate specifically on the assistance brought by existing ADLs regarding the thorough elaboration of system architectures, and reference illustrative ADLs from this standpoint.

Existing ADLs differ on the intended exploitation of the elaborated architecture within the development process. To the best of our knowledge, all of the ADLs that have been proposed in the literature target aid in the system's design, and fall into (sometimes both) two categories depending on whether they provide assistance in the analysis or in the construction of the designed system.

Detailing further, existing ADLs may be distinguished according to the support they provide with respect to the three above aspects of system development:

- ***System design***: As already raised, this is the central target of ADLs. However, ADLs differ at this level according to the assistance they offer for stepwise architectural refinement. In general, this takes the form of a classical type system defined over architectural elements where a subtyping relation is often included towards checking the correct refinement of the architectural elements and possibly of architectural styles. For instance, this is the approach undertaken in the definition of the C2 language [Medvidovic *et al*. 1999]. Refinement correctness may be assessed more thoroughly by accounting for the behavior of the architectural elements. Such a capability is offered by the SADL language [Moriconi & Riemenschneider 1997] based on the work presented in [Moriconi *et al*. 1995] that uses logic theories for checking correct architecture refinement with respect to the embedded interaction protocols (i.e., the focus is on the refinement of connectors). Specification of the behaviour of architectural elements in logic is also exploited in [Saridakis & Issarny 1999] for assisting in the refinement of architectures enforcing fault tolerance properties.

- ***System analysis***: Support for behavioural analysis using model checking technology has received a great deal of attention in the definition of ADLs. These ADLs are based on existing formalisms and exploit associated tools for enabling checking liveness and/or safety properties. The Wright language [Allen & Garlan 1997] belongs to this family of ADLs; it is based on CSP and is coupled with the FDR tool for checking deadlock freedom. The use of CSP further enables checking architecture consistency with respect to the correct usage of connectors according to the interaction protocols expected by components. Although the use of the CSP formalism does not prevent the description of dynamic architectures as presented in [Allen *et al*. 1997], it is quite restrictive with this respect. This has in particular led to the definition of ADLs based on the Π-calculus such as the one presented in [Canal *et al*. 1999]. Behavioural analysis is also supported by the Darwin language [Magee *et al*. 1995] through its extension with labelled transition systems that comes along with a tool for compositional reachability analysis. This allows analyzing system models with respect to both safety [Cheung & Kramer 1996] and liveness [Cheung *et al*. 1997] properties. Yet another approach to behavioural analysis has been undertaken in the definition of the Rapide language [Luckham *et al*. 1995]; this ADL enables simulating the system model through the use of partial order sets of events.

  Another key aspect in the analysis of a system architecture lies in assessing the qualities of both the architecture (e.g., evolvability, scalability, ...) and the system itself (e.g., dependability, performance, ...). Quality analysis of system architectures is quite an open area of research as it raises a large number of issues and is made more complex by the subjective nature of quality properties. However, quality assessment of software architectures has already been examined with respect to the quality of the software [Kazman *et al*. 1994], and to performance and reliability properties of architectural styles [Klein et al. 1999]. The latter work focuses on the integration of quality attributes within the description of architectural styles so as to assist in the development of systems offering quality properties. The assessment of the overall system quality relies on existing methods and tools for reliability and performance assessment where the designer translates the architecture-based system model into models understood by the quality assessment tools. A similar approach is undertaken in [Zarras & Issarny 2000] but considering the use of a standard notation (i.e., UML) for modeling architectures, and addressing the systematic translation of these models into models processed by tools for reliability and performance assessment.

- ***System construction***: The third area of extensive contribution in the definition of ADLs to assist in the software development process, is on easing system construction. In this context, the ADL is coupled with tools that generate executable configurations from the description of system architectures whose primitive components and connectors correspond to either source files or executables. Examples of ADLs belonging to this category, to give a few, are Aster [Issarny *et al*. 1998a], C2 [Medvidovic *et al*. 1999], Darwin [Magee *et al*. 1997], and Unicon [Shaw *et al*. 1995].

There are other dimensions in the distinction of ADLs as in particular highlighted in [Medvidovic & Taylor 2000]. The additional criteria that are introduced there relate to further refining the above three areas where ADLs contribute to the development process. However, one significant distinction that we have not made so far relates to whether the ADL enables the definition of any kind of architectures or whether it is domain-specific. While most of the ADLs aimed at system analysis are general-purpose architecture modelling languages, a number of those aimed at system construction targets a particular domain. For instance, the C2 language enables modelling only layered systems.

Although there are ADLs that offer similar capabilities, existing ADLs in general offer a complementary rather than a competing aid. This has led to the definition of the ACME interchange language, which aims at enabling the combination of various ADLs and associated tools in a single development environment [Garlan *et al*. 1997]. In the same spirit, the AML architecture meta-language has been proposed in [Wile 1999]. These efforts together with useful design assistance brought by the various ADLs enable foreseeing the provision of an ADL-based environment that effectively supports the overall software development process. However, the actual usage of such an environment does not depend solely on the benefits it brings regarding the quality of the software products that can be developed. It first requires acceptance from architects, designers and developers who may not be willing to invest in acquiring knowledge about some novel notations, which is most likely to happen when those are based on formal methods. This issue is already highlighted by the actual usage of ADLs that have been around for some time. To the best of our knowledge, such ADLs have been used for the analysis of complex software systems (e.g., Wright was used for analyzing the HLA architecture [Allen *et al*. 1998]) but this was within research projects. Use in industry of ADL-based development environments still requires further evolution of ADLs. In particular, the growing acceptance of the UML standard within industry, for modeling software systems raises the concern of coupling ADL notations with UML rather than considering them as two separate (possibly conflicting) notations serving distinct purposes.

1.3.2 Relation with the UML Standard Modelling Notation

UML is a notation for object-oriented design and analysis, which was standardised by the OMG in 1997 [OMG 1997a, OMG 1997b]. UML consists of a meta-model, the definition of the semantics of concepts identified in the meta-model, and a notation guide that identifies different diagram types that utilise the concepts of the meta-model. A UML model of a system then consists of several partial models where each addresses a certain set of software aspects. Models are specified using diagrams, which fall into the following categories: (*i*) static structure diagrams including those defining object types, (*ii*) use case diagrams to represent the functionalities of a system (or any model element) as manifested to external actors, (*iii*) sequence and collaboration diagrams for describing patterns of interactions among instances, (*iv*) state diagrams to characterise the dynamic behaviour of model elements, (*v*) activity diagrams to represent state machines of actions that are generated internally, and (vi) implementation diagrams to show aspects of the system implementation, i.e., the structure of the source code and of the runtime implementation.

Considering UML as a possible notation for describing software architectures is not widely accepted in the software architecture research community. In particular, it is often argued that UML is specifically aimed at object-oriented design and is thus concerned with a lower level of abstraction than ADL. It is further argued that UML lacks supporting formalisms and is thus not suited for the rigorous analysis of software systems. The latter restriction is not significant since there is a number of ongoing work about coupling UML with formal notations, including the associated OCL language [OMG 97c] that enables specifying semantics constraints in terms of first-order logic predicates within UML-based system models. In general, nothing prevents extending UML so as to enable formal specifications within diagrams for more rigorous analyses, in the same way it has been adopted for the definition of ADLs. Considering the former argument against using UML as a base notation for architecture description, this is a subjective matter as UML classes may correspond to "*coarse-grained*" architectural elements such as components. Alternatively, architectural description using an ADL may well be exploited to detail a low-level architecture where objects correspond to some programming language objects. The distinction is further blurred when considering the architectures of distributed object systems such as CORBA 3[2]. The only pre-requisite in describing architectures using UML is to maintain a clear distinction among the various levels of abstraction that are addressed during the system design so as to always be able to capture the system's gross organization at the right level of detail.

Furthermore, as already raised in the previous section, the use of UML for architecture description has already been successfully adopted in industry for assisting in architecture-based development processes [Borrmann & Newberry-Paulish 1999, Kruchten 1999]. However, these approaches do not enable taking benefit of the various results in the area of ADL definition. Towards that goal, the integration of ADL-based specifications (i.e., specifications written in Wright and C2) within a UML model has been studied in [Robbins *et al.* 1998]. As a result, this enables architecture-based design using an *a priori* well-known standard notation while allowing usage of analysis tools coming along with ADLs. From our point of view, it seems more viable in the long term to consider direct extensions of UML for architecture-based development. In particular, the tools coming along with ADLs, which were sometimes developed prior to the ADL (e.g., the FDR tool exploited for the analysis of Wright-based architectures), can be reused in this context without much effort. The issue is then on providing appropriate guidelines for the description of UML-based architectures, that is, to set the UML notations and their semantics, which are to be used for characterizing architectural building blocks. The actual provisioning of such an environment as opposed to an environment

[2] http://www.omg.org

relying partly on an ADL is still an open issue, which will in particular be considered within the DSoS project.

1.3.3 Research Directions

Ongoing research in the software architecture domain relates to further promote the disciplined elaboration of system architectures by enhancing existing results whose overview was provided in the previous sections. Considering the development of dependable systems of systems architectures, a great deal of attention needs to be put on the thorough assessment of the system's dependability as well as on the exploitation of integration technologies for building up architectures. The former issue will be addressed through extensive work on validation and evaluation techniques within the DSoS project, which shall be combined with the architecture-based design methods that will also be investigated within the project. The latter issue is concerned with the specification and design of connectors relying on existing component-based and middleware technologies, which is further addressed in the next section.

An area of research works in the software architecture domain that is relevant to the DSoS project objectives is the one concerned with novel systems resulting from technological evolutions. In particular, one application domain for provisioning systems of systems relate to systems developed over the Internet, which are typically dynamically formed and coalitions of distributed autonomous resources. In this context, it has been argued that the design of the system's software architecture must account for partial knowledge about the behaviour of the constituent architecture elements and hence requires adapting the corresponding specifications [Shaw 2000]. In addition, Internet-based systems as well as a number of other emerging distributed software systems must be self-adaptive due to their continuously changing environments. An infrastructure supporting such a feature has been proposed in [Oreizy et al. 2000], which is based on architecture-oriented design and implementation. Although at a preliminary stage, this work shows the benefits of using an architecture-based approach for the development of systems of systems, in addition to the ones that have been mentioned so far. As a longer term research initiative in the software architecture domain that relate to developing next-generation systems of systems, we meet studies about enabling invisible (or pervasive) computing where services and information are seamlessly brought to users out of existing systems. In particular, this brings a number of challenges for the definition of systems' software architectures as for instance discussed in [Garlan 2000].

1.4 Architecture-based Development of Distributed Systems

The development of distributed software systems is recognised as a complex task: in addition to the development of the system-specific parts, issues raised by distribution management should be addressed. However, since the early 90s, the development of such systems has been made simpler through the emergence of standardised software infrastructures that offer solutions to problems frequently met in application families. Such infrastructures lie in component and middleware technologies. Briefly stated, components correspond to the building blocks of distributed systems, and may be easily composed for interaction. Thus, this corresponds to the notion of component used in architectural descriptions, except it is closely coupled with some middleware technology that is a middleware layer lying between the application and the network operating system, and providing reusable solutions to problems like heterogeneity, interoperability, security, transactions, fault tolerance *etc*. Middleware and component technologies are now exploited for the development of most distributed systems and shortly discussed in the following subsection. This leads us to refine the notions of architectural building blocks as well as to examine the architecture-based design of middleware underpinning the development of distributed software systems.

1.4.1 Component and Middleware Technologies

The main constituents of any distributed system are the system components, which offer services to other components that can request service execution. The system's component model then defines the way services are defined and accessible, and the way components are identified. Considering distributed systems that are developed nowadays, these rely on some distribution middleware, which sets the system's model [Lewandowski 1998]. Available middleware can be classified into three gross categories: (*i*) transaction-oriented middleware that mainly aim at system architectures whose components are database applications; (*ii*) message-oriented middleware that target system architectures whose component interactions rely on publish/subscribe communication schemes; and (*iii*) object-oriented middleware that are originally based on the remote procedure call paradigm and that enable the development of system architectures complying with the object paradigm (e.g., inheritance, state encapsulation) and hence enforce an object model for the system (i.e., the architectural components are objects and connectors abstract at least the interaction protocols offered by the middleware). Notice further that although originally based on the remote procedure call interaction paradigm, object-oriented middleware may be extended with features enabling the development of system architectures similar to the ones targeted by the two other categories of middleware.

We refer the interested reader to [Emmerich 2000] for a detailed presentation of distributed object engineering, including supporting object-oriented middleware. Basically, we find the three following current major object-oriented middleware for building distributed applications:

- The standard Common Object Request Broker Architecture (CORBA) [OMG 1995] from the Object Management Group (OMG) defines an object model for building CORBA applications. In this context, an application is a collection of objects where each object is an identifiable encapsulated entity that may provide an interface defined in the CORBA Interface Definition Language (IDL). An interface is a set of operations that can be requested by objects. Requests are then issued through a CORBA proxy, which combines functionalities provided by the CORBA Object Request Broker (ORB) that mediates the interactions among objects. A CORBA proxy subdivides into the client-side proxy (called stub) and the server-side proxy (called skeleton). In addition to the above, CORBA-compliant middleware infrastructures may provide a set of standard Common Object Services (COSs) for distribution management [OMG 1998] (e.g., COSs for the management of concurrency control, objects, security, transactions). Recently, the definition of a new CORBA version, called CORBA 3, has been undertaken for further promoting component-based development. Basically, the resulting extension lies in enabling the specification of components out of objects, which are closer to the component notion of software architectures.

- The proprietary Distributed Common Object Model (DCOM) middleware infrastructure [Microsoft 1998] from Microsoft offers functionalities similar to the CORBA standard, introducing an object model and a broker mediating object interactions. Interfaces are here defined using the DCOM IDL, called MIDL. The infrastructure also comes along with services for enhanced distribution management.

- The proprietary Enterprise Java Beans (EJB) infrastructure [Sun 1998] from Sun enables the development of applications built out of a collection of objects, called beans, which may be either persistent or not, and are both hosted and managed by entities called containers. Object interfaces are defined using an IDL that is a subset of Java. The Java Remote Method Invocation (RMI) broker is used for managing interactions among objects and additional services are offered for implementing enhanced functionalities within containers.

The above middleware need to be accounted for in the development of any distributed system. However, although they have distinct features and rely on slightly different object models, a development environment prototype that exploits a single kind of object-oriented middleware should be easy to adapt to support the others. In particular, this is substantiated by the existing support for interoperation among components belonging to the three above types of middleware and by the fact that they can be considered as prescribing a model that is a subset of the Open Distributed Processing Reference Model (RM-ODP) [ISOIEC 1995]. The recent Simple Object Access Protocol (SOAP) from the W3C is also likely to play a prominent role in the development of Internet-based distributed systems; it is an XML-based lightweight protocol for exchange of information using remote procedure calls. However, it is still at an early design stage and will be examined in the course of the DSoS project when getting more elaborated.

1.4.2 Matching Architectural and Middleware Building Blocks

The building blocks of distributed software systems relying on some middleware infrastructure fit quite naturally with the ones of software architectures. Hence, the development of such systems can be assisted with an architecture-based development process in a straightforward way. This is already supported by a number of ADL-based development environments targeting system construction (see Section 1.2.1) such as Darwin and Aster. In this context, the architectural components correspond to the application components managed by the middleware, and the architectural connectors correspond to the supporting middleware. However, as noticed previously, most of the work on the specification of connectors have focused on the characterisation of the interaction protocols among components whilst connectors abstracting middleware embed additional complex functionalities (e.g., support for the management of fault tolerance, security, transactions).

The above concern has led the software architecture community to examine the specification of the non-functional properties offered by connectors. For instance, these are specified in terms of logic formulae in [Issarny *et al*. 1998b], which further enables synthesising middleware customised to the application's requirements as supported by the Aster ADL [Issarny *et al*. 1998a]. Dually, domain-specific ADLs such as C2 target the description of architectures based on connectors enforcing specific interaction patterns, which may not be directly supported by middleware infrastructures. This issue has been investigated in [Dashofy *et al*. 1999], which explores the applicability of middleware infrastructures to the construction of domain-specific software architectures.

Another issue that arises when integrating existing components, as promoted by middleware infrastructures, is that of assembling components that rely on distinct interaction patterns. This aspect is known as architectural mismatch [Garlan *et al*. 1994b] and is one of the criteria substantiating the need for connectors as first-class entities in architecture description. The abstract specification of connector behavior as for instance supported by the Wright ADL enables reasoning about the correctness of component and connector composition with respect to the interaction protocols that are used. However, from a more pragmatic standpoint, software development is greatly eased when provided with means of solving architectural mismatches, which further promotes software reuse. Such a systematic aid is presented in [DeLine 1999], which introduces a number of notations and associated tools that resolve mismatches during the integration of reused software.

Connectors implemented using middleware infrastructures actually abstract complex software systems comprising a broker, proxies but also services for enhanced distribution management. Hence, middleware design deserves as much attention as the overall system design and must not be treated as a minor task given reliance on some middleware infrastructure. Architecture-based design is again of significant assistance here. In particular, existing ADLs enable describing conveniently middleware architectures as addressed in [DiNitto & Rosenblum 1999]. In addition, the fact that middleware architectures build upon well known solutions regarding the enforcement of non-functional properties, the synthesis of middleware architectures that comply with the requirements of a given application may be partly automated through a repository of known middleware architectures [Zarras 2000]. In the same way, this *a priori* knowledge about middleware architectures enables dealing with the safe dynamic evolution of the middleware architectures according to environmental changes, by exploiting both the support for adaptation offered by novel reflexive middleware infrastructures and the rigorous specification of software architectures as enabled by ADLs [Blair *et al*. 2000].

1.5 Concluding Remarks

This chapter has given an overview of past and ongoing work in the software architecture domain for effectively enabling architecture-based development of robust software systems. Results in the area primarily lie in the definition of ADLs that allow the rigorous specification of the elements composing a system architecture, which may be exploited for aiding in the system design and in particular in the assessment and construction of software systems.

Ongoing research work focuses on closer coupling with solutions that are used in practice for the development of software systems. This includes integration of ADLs with the now widely accepted UML standard for system modeling. From this perspective, one issue that remains is whether architecture description should only be given in terms of UML diagrams with a possible extension of the UML language, or be a combination of ADL-based specifications and UML diagrams. Practically, the former approach should be encouraged so as to ensure the actual usage of solutions aimed at easing architecture-based development of software systems. However, it is not yet obvious that this is achievable. Another area of concern when considering the development of actual distributed systems is the one of exploiting middleware infrastructures and in particular the CORBA standard for the systems' development. This issue has already deserved a great deal of attention and there exist architecture-based development environments that do ease the design and construction of middleware underlying the system execution out of middleware infrastructures. However, addressing all the features enabled by middleware within the architecture design is not yet fully covered. For instance, this requires capturing the composition of, possibly interfering, middleware services enforcing distinct non-functional properties. Another area of ongoing research work from the standpoint of architecture specification relates to handle needed architectural evolution as required by emerging applications, including those based on the Internet. In this context, it is mandatory to enable the design of system architectures that can adapt to the environment.

The above research issues are of prime interest for the development of any distributed system, and in particular for the one of dependable systems of systems. Hence, solutions that are to be proposed shall be examined in the course of the project. Concentrating more specifically on the development of dependable systems of systems, their intrinsic features raise two issues for a thorough architecture-based development. First, although the abstract notion of architectural component does enable considering an autonomous system as a component instance, most of the results from the software architecture domain target components that correspond (possibly implicitly) to pieces of software. Hence, the adequate specification of components abstracting autonomous systems should be investigated. This distinctive feature also impacts upon the system construction from architectural description. For instance, wrapping technology for integrating systems need be devised. Dealing further with dependability of the overall system adds on to the above. In particular, architectural solutions to the enforcement of dependability properties must be precisely characterised and easy to integrate within systems. Also, it is crucial to be able to assess the system's dependability, which will rely on the work done within the DSoS project in the area of validation techniques and dependability evaluation.

Chapter 2 – Mechanisms for Enforcing Dependability of Services

Jean-Charles Fabre, Eric Marsden, David Powell *(LAAS-CNRS)*;

Alexander Romanovsky *(University of Newcastle upon Tyne)*

2.1 Introduction

The objective of this chapter is not only to address the essential mechanisms for enforcing dependability of services but also to discuss architectural concepts for the design and the implementation of dependable systems of systems. In Section 2.2, we discuss modern approaches, which have proven to be useful for structuring complex applications and for providing their fault tolerance: exception handling mechanisms, transaction models, advanced workflow systems, various atomic actions based strategies. In Section 2.3, we summarise the conventional distributed fault tolerance techniques together with an example and talk about middleware-based fault tolerant architectures, in particular, ones based on CORBA. Open middleware and reflective architectures are briefly addressed as a promising field of investigation within DSoS. Section 2.2 is in fact devoted to application-specific fault tolerance techniques whereas Section 2.3 is directed at application-transparent fault tolerance mechanisms and architectural solutions. Section 2.4 summarises the work we intend to do in the DSoS project concerning both topics.

2.2 Structuring and Fault Tolerance Mechanisms for Complex Systems

2.2.1 System Structuring, Fault Tolerance and Exception Handling

Providing fault tolerance is an immanent part of all steps of developing modern complex applications. The choice of the ways the system is structured should depend on the ways the system will tolerate faults. There are many reasons supporting the fundamental principle that such system design should be accompanied by designing features providing its fault tolerance, so that at each phase of system development (starting from the first ones) fault tolerance issues are addressed. The best practice uses approaches combining in a natural way system development and structuring techniques to be applied with the fault tolerance techniques and forcing system developers to apply appropriate measures for tolerating faults.

The importance of *structuring* for developing complex systems is well recognised. It allows us to develop system recursively, reasoning in terms of higher level abstractions by hiding several steps of data and behaviour modification. Action nesting, multi-layer system structuring, nested method calls and classes are some of the examples of structuring.

Errors of different types can occur in complex systems of systems and very often the responsibility of tolerating them is laid on or transferred to the application level (e.g., environmental faults, residual software faults, transient faults, faults which the underlying software or hardware is not able to handle transparently, etc.). The most general and beneficial approach to providing *application-specific fault tolerance* is to associate it with the system structuring while developing such applications. It is clearly much more complicated to deal with faults if the units of system structure are not units of error containment, error detection or error recovery. Many researchers and system developers believe that the *atomicity* property, understood here as indivisibility of a unit execution with respect to errors (all-or-nothing semantics), is vital for proper system structuring and providing fault tolerance. Systems are easier to develop, to understand and to analyse if their execution is built out of atomic units encapsulating several components and/or operations provided no information crosses the border of such units. The ability to nest such units is essential for dealing with system complexity in a scalable way (a unit is called *nested* if it contains a subset of components or/and operations from the *containing* one).

Exception handling [Cristian 1995] is accepted to be the most general technique for dealing with any faults which can hit an application (including application-specific faults and, in particular, environmental faults [Rubira 1994]) because it offers several ways of separating normal and abnormal behaviour. To define the rules of exception handling and, in particular, exception propagation, one has to relate exception handling to a structuring technique. A set of exceptions and exception handlers is associated with an *exception context*. If one cannot handle an exception raised within the context or if there is no handler for the exception raised, then an exception is propagated to the containing context. Each context is associated with a structuring unit. We will say that, for a given component, the *containing context* is associated with the component which uses it (or that the former component is associated with the *nested context* with respect to the latter). “Uses” means here that the component refers to the interface of another component.

The considerations above clearly show that system structuring, fault tolerance and exception handling are concerns which should be addressed together while both choosing the approaches to system design and developing complex applications. Good system development should use appropriate techniques for dynamic and static system structuring out of units incorporating general fault tolerance features based on exception handling. It is extremely beneficial to associate different meanings with the same concept of a structuring unit and by doing this to minimise the number of concepts used in system development.

Complex systems of systems are distributed systems with intensive parallelism and concurrency. There are many complex scenarios and many subsystems involved in SoS execution. Many abnormal events can happen and should be dealt with in a disciplined fashion. This requires special techniques for structuring, fault tolerance and exception handling. We will briefly outline several main trends in research that are relevant, from our point of view, to the specific characteristics of complex systems of systems.

2.2.2 ACID Transactions and Advanced Transaction Models

Atomic transactions were one of the first structuring techniques developed for concurrent and distributed systems. Preserving and guaranteeing important properties of data (or, objects, resources) affected by several operations are in the focus of this approach, which takes care of atomicity, consistency, isolation and durability (the ACID properties) of these structuring units [Gray & Reuter 1993]. Consistency means that the execution of any transaction on its own is a correct transformation of the data states and does not violate their integrity. Isolation plays a major role in providing inter-transaction concurrency: when it holds, the designer of a transaction which uses some data does not have to know about other transactions using them. It is guaranteed that, even when several transactions are executed simultaneously, they do not affect each other, and the recovery of any of them is separated from the execution of the others. Durability is understood as the ability of data to survive any assumed hardware faults which can happen after the transaction has been successfully completed (committed). The atomic transaction scheme relies on three standard operations: start, abort and commit, which mark the boundaries of a transaction. Each transaction encompasses several operations on data and, in this sense, is a concept of a higher level than any individual operation.

Although atomic transactions have been successfully applied in many applications, developing new and extended transaction models has always been an area of a very active research, mainly because the original model is either too general or too restrictive in many respects.

To provide more flexible features for programming complex systems and their recovery, the concept of nested transactions was developed [Moss 1981] in which a transaction can start subtransactions, thus creating a tree of sibling transactions. A subtransaction can either commit or abort; but its commit does not take effect (is not visible to the outside world) until the parent transaction commits. The advantages of nested transactions are: they can be aborted independently without causing the abortion of the whole transaction, sibling transactions are executed concurrently.

Recently the concept of multithreaded transactions (MTT) has been developed to allow several active components (threads, processes) to take part in the same transaction and to operate together on the same set of data. One of them starts a transaction, then others learn its identity, using this identity they can access data within such transaction. Very typical examples of MTT are the CORBA transaction service and Arjuna [Parrington *et al.* 1995]. The object-based language Argus [Liskov 1988] is a very interesting example of MTT enriched by powerful exception handling: applications are composed out of guardians, each of which provides an interface consisting of callable procedures, called handlers. Handlers can fork concurrent threads which are joined when the handler is completed. Handler execution forms an atomic transaction, nested handler calls form nested transactions. Argus provides a very powerful extension of sequential exception handling: handlers can have exceptions declared in their interface which are propagated to a single-threaded caller when any thread inside the transaction signals it. Any thread may decide to signal an exception with or without transaction abort. The Argus model proved to be very influential: several systems have been developed which rely on similar computational models.

A number of generalised transaction models have been developed recently in order to overcome some of the limitations of traditional (flat or nested) transactions, such as lack of support for long-lived actions, cooperative activities and multidatabase systems. Much of this work is surveyed comprehensively in [Elmagarmid 1993]. Long-lived activities, for example, can keep data locked for a very long period of time and considerable computation can be lost if they are aborted. Usually, these models extend the canonical transactional model by breaking or softening some of the ACID properties. In the Sagas model [Garcia-Molina & Salem 1987], for example, each saga consists of several ACID transactions T1, …, Tn; the support guarantees that either all of them are successfully completed or compensation transactions are run to eliminate partial results. It is required that each Ti has Ci - the compensation ACID transaction, so that if an execution of a saga is interrupted/aborted when Tj is executed then the support aborts Tj and runs Cj-1, ..., C1. The Dynamic Action model [Nett & Mock 1995] allows the isolation property to be relaxed on the grounds that it neither matches the general-purpose character of distributed systems, which should support communication and cooperation nor does it allow for flexible concurrency. In this model, data uncommitted within one transaction can be accessed by another transaction, but the support transparently traces all transactions which have used or might have used these uncommitted data and aborts them if the initial transaction is aborted.

2.2.3 Advanced Workflow Systems

Another important research area relevant to the DSoS project is developing modern workflow systems, which are used for designing and controlling complex business processes [Leymann & Roller 1999]. Workflow systems are concerned with both activity control and data integrity of such applications; they usually describe all possible paths of the process execution, structure the execution of these systems as sequences of activities (some of which can have nested activities in their turn). Workflow management systems are useful mainly for bottom-up design. They fit well to the object model: activities can be implemented as invocations of objects. Usually, such systems use languages of two types: scripting (or, modelling) languages for describing the sequences of activities and conventional programming languages for manipulation of data. Workflow management systems deal with long-lived activities or with activities involving human beings, this is why they typically provide support for tracing dependencies which are created when data from the completed activity are used by another activities: appropriate actions involving all dependant activities can be taken if necessary. This feature can be used, for example, if data inconsistency is created or an error in the completed activity is found.

It has been recognised for many years that workflows could benefit from using the transactional paradigm and in many respects developing modern workflow systems became the main driving force and application area for research on transactions [Hsu 1993]. Systems are much easier to design if their parts have the ACID properties. Transactional workflows allow system designers to view these parts as atomic transactions. For example, the OPENflow system [Wheater *et al.* 2000] is based on Arjuna. It uses a scripting dataflow language (with many elaborate features for component coordination) for describing the schema and Java for programming data manipulation. This approach promotes a recursive view of workflow execution. In this system, component activities can be programmed to have the ACID properties but if they do not, it is the responsibility of the system designer to trace dependencies and undertake the required actions (e.g., to guarantee the consistency or to perform the recovery). The OPENflow system is CORBA compliant and it has robust distributed support implemented using service ACID transactions.

Many workflow systems use extended transactional models to deal with the peculiarities of this application area: weakening the ACID properties for some activities (e.g., for long-lived components) is often required. In this case, the support systems often trace the dependencies and perform some special actions on all dependent activities (informing or aborting them, executing separate compensation or replacement activities, etc.).

Modern business processes have to deal with numerous abnormal situations, which cannot be treated simply either in an ad hoc way or by transactional abort, this is why introducing exception handling models into workflows is now a topic of active research. It is clear that abnormal situations and handling them are very much specific for these applications and that exception handling models should fit well into the workflow development process. Analysis of existing systems clearly shows that general exception handling features always support disciplined, structured and unified ways for dealing with exceptions of different types. For example, the OPERA process support system [Hagen & Alonco 1998] offers flexible features combining exception handling and an advanced transaction model. The scripting language incorporates special features for error detection and handling, which are conceptually similar to exception handling features found in programming languages. In effect, this approach offers an advanced fault tolerance mechanism for incorporating both transactions and exception handling into workflow systems. This model allows parent activity to define handlers for each nested activity whose execution can be either aborted or resumed after handling. The choice of the way exceptions are handled depends on the type of activities; the OPERA system allows system designers to develop activities of 5 types: non-atomic, semi-atomic (activities, which do not provide automatic rollback but keep enough information to allow undo), atomic (activities, which have no side effect if they fail), restartable (activities, which can be restarted after failure), compensatable (activities, which can be rolled back after they have successfully completed).

2.2.4 Atomic Actions and Coordinated Atomic Actions

Concurrent systems can be classified into three categories [Horning & Randell 1973, Hoare 1976]: independent (disjoint), competing and cooperating systems. Transactions are intended for competitive systems as they guarantee the consistency and atomicity of data (objects) accessed by multiple clients but they do not deal with structuring multiple clients, or, to put in a more general context, they are not suitable for developing cooperative systems. In reality, many complex systems are cooperative. The general concept of *atomic actions*, proposed in [Campbell & Randell 1986] is intended for designing such systems and for providing their fault tolerance by a disciplined exception handling mechanism. Several participants enter an action and cooperate inside it to achieve joint goals. To guarantee atomicity no information is allowed to cross the action border. Participants leave the action together when all of them have completed their job. If an error is detected inside an action, all participants take part in a cooperative recovery. Atomic actions can be nested, so that when a nested action fails, a containing action is responsible for recovery. Many implementations of atomic actions have been developed (e.g., using CSP, Ada, OCCAM, C and C++ extended by concurrency features, multicast libraries, specialised operating systems).

A more general approach, which deals with both competitive and cooperative systems is *Coordinated Atomic* (CA) *actions* [Xu *et al.* 1995] allowing designers to choose the relationship between components, i.e., whether they compete or cooperate. CA actions are a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components in distributed systems: they integrate and extend atomic actions and transactions, the former are used to control cooperative concurrency and to implement coordinated error recovery, the latter are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency. This allows various types of faults to be tolerated, as well as combinations of faults occurring in different components involved in the CA action execution (using an extended resolution mechanism [Campbell & Randell 1986]). Fault tolerance is provided by a safe and simple exception handling model where:

- Internal and interface (external) exceptions are clearly separated.
- All action participants are involved in coordinated exception handling when any of them raises an exception.
- Each of the participants has to have a handler for each internal exception.
- Concurrent exceptions are resolved before handling.

- Only interface exceptions can be propagated from an action.

A number of distributed Java and Ada schemes have been developed and various case studies have been used to check the applicability of this approach (e.g., [Romanovsky *et al*. 1998, Zorzo *et al.* 1999, Xu *et al.*1999]: a series of Production Cell case studies, including a fault tolerant one and a real time one; a distributed internet Gamma computation. An auction system, and a subsystem of a railway control system, which deals with train control and coordination in the vicinity of a station are under development now.

2.2.5 Spheres of Control

C.T. Davies pioneered in developing a general conceptual view on both the atomic transactions and atomic actions [Davies 1979]. He addressed many concepts concerned with concurrent systems, recovery and integrity within an overall scheme that he called *data processing spheres of control*. Spheres of control are intended to deal with various problems including coordinating multiple processes within recovery regions, sharing partial (uncommitted) data between processes, and controlling concurrency across machine boundaries. However, the descriptions of spheres of control provided little implementation advice for general applications, and early work on transactions, though influenced by Davies, was much less ambitious in its goals. Spheres of control define process bounding for various purposes and allow for both dynamic and static system structuring with a clear definition of properties of this structuring. Many kinds of control are considered in the spheres to make it possible for system designers to have a flexible choice: they include atomicity, consistency, recovery, auditing, commitment, resource control, etc. All phases of fault tolerance are addressed using spheres of control. The intention is to be as general and as flexible as possible to allow for a flexible choice of the amount of processing one wishes to consider as a unit of action, for maximum process independence while maintaining consistency and for a choice of the level of atomicity which suits the application best.

2.3 Middleware-based Systems and Fault Tolerance Mechanisms

Independently from their basic principles, fault tolerance is tightly connected to architectural design choices as far as application-transparent mechanisms are concerned. From typical hardware-based solutions in the seventies, the development of large distributed systems played in favor of software-based solutions. Their objective is to provide distributed fault tolerance strategies on networks of standard computers, mainly based on the replication of software components. Key ingredients are group communication protocols that are the basis for the development of distributed error recovery strategies.

In this section, we summarise the most recent works concerning fault tolerance architectures for distributed systems and analyse the most promising solutions for large systems. We focus in particular on middleware technology and CORBA because of their widespread usage in large industrial computer systems. The combination of reflection and middleware technology is certainly the most promising field of investigation for the development of adaptable fault tolerance. The key features of reflective middleware seem of high interest for introducing dependability mechanisms into systems of systems.

2.3.1 Distributed Fault Tolerance Mechanisms

The definition of fault assumptions to be considered is a major activity in the design of fault tolerant systems. Faults can be accidental or intentional. They may be introduced during the design process or occur in operation. The choice of appropriate fault tolerance techniques depends very much on the nature of the faults and other parameters such as their persistence. Design faults can be handled by diverse design techniques [Randell 1975, Chen & Avizienis 1978, Laprie *et al.* 1995]. Intentional faults are addressed by security techniques (authentication, ciphering techniques, FRS[3] techniques). FRS [Deswarte *et al.* 1991] addresses both intentional and accidental faults. Physical faults in operation are often handled by replication techniques, in particular in distributed systems.

In the context of distributed systems, the fault model is defined at the level of communicating nodes or processes. The most common fault model is that of simple crash faults. The underlying assumption is that computing elements are fail-silent. This fault model is adopted not because systems necessarily employ extensive self-checking (the inverse is usually the case) but because there are many reasons for which the type of process-level fault can occur, over and above just underlying hardware faults: power failures, unscheduled maintenance, network disconnections, process abort, process blocking due to lack of resources, etc. Dealing with this type of fault is thus often the main focus as far as availability is concerned. In very critical systems, such as safety-critical systems, more general faults such Byzantine faults might need to be taken into account. However, these require complex agreement protocols, described in many papers, such as [Dolev *et al.* 1997, Guerraoui & Shiper 1997].

For crash faults, possible solutions are based on either stable storage or replication strategies. The various protocols suppose that detection of a crash is based on the absence of some sort of *I'm alive* or *heartbeat* message between interacting processes. However in asynchronous distributed systems, it is impossible to distinguish a failed process from a very slow process [Fisher *et al.* 1985]. In many practical systems, time-outs are used to empirically detect whether remote processes have crashed, even if assumptions of synchrony are not really substantiated by an appropriate design. In effect, a synchronous model is assumed, but it is admitted that there is some probability of this assumption being violated [Powell 1992]. It may just be the case that the distributed application is not very critical, so the occasional lack of fault-tolerance has no dire consequences. Alternatively, time-outs are over-dimensioned to the extent that the probability of false detection is considered negligible.

Much recent research has been devoted to defining models that are intermediate between asynchronous and synchronous models. Among them, we can cite the asynchronous model augmented with the notion of unreliable failure detectors [Chandra & Toueg 1996], the timed asynchronous model [Cristian & Fetzer 1998], and the quasi-synchronous model [Veríssimo & Almeida 1995].

Fault tolerance strategies based on stable storage consider that a memory storage device is designed and implemented as a fault tolerant unit. A snapshot of the state of the running entities is regularly stored in this unit. Recovery consists in this case in creating a new copy of the object and in performing its initialisation from the last state saved on stable storage. When the fault tolerance

[3] FRS: Fragmentation-Redundancy-Scattering.

strategy is based on replication, a set of replicas executes the same computation on different sites. We can identify passive, semi-active and active replication strategies [Chérèque *et al.* 1992]. In the former, only one replica is active, the primary, and returns the results to the calling entity; the other replicas, the backups, are passive and update their state according to the last snapshot transmitted by the primary (in checkpoint messages). In the semi-active strategy, all replicas are active but there is one, the leader that dictates some decisions (e.g. message acceptance or process preemption) to the other replicas, namely the followers [Barrett *et al.* 1990]. This means that, in this case, a consensus protocol is not mandatory. In addition, the leader replica may take sole responsibility for sending output messages. The other object replicas are called followers. The current state of the computation is acquired by the followers if (i) they process the same input messages in the same order and if (ii) the same input messages produce the same results. The first assumption can be fulfilled (but not required) by a group communication service (as in ISIS [Birman 1985]) ensuring total order of input messages to all correct recipients. The second assumption relates to the determinism of the replicas computation and relies on a design discipline: identical input data must lead to the same output results at all replicas. For instance, local values (time, random numbers, etc.) must be avoided or agreed among the replicas (in fact, imposed by the leader). Intra-object concurrency can also lead to non-deterministic behaviour. These assumptions are also mandatory for the last strategy, the active replication strategy; in this case all the replicas are active, process the request autonomously and may be able to send the results to the calling object. Agreement on the final results is performed by majority voting either (i) between replicas before the results are sent to the calling object (source validation) or (ii) at the calling object receiving a copy of the results from all active replicas (destination validation). In this solution, not just omission faults are tolerated but also value faults. Recovery after a crash may differ slightly from one strategy to another. In the passive and semi-active replication strategies, a backup must be elected to replace the primary or the leader respectively. Reconfiguration leads to the creation of a new object replica (cloning). In the active replication strategy, there is no recovery (it is a masking strategy) and just reconfiguration is achieved (a new object replica is created and inserted into the group of object replicas) in order to restore the initial fault tolerance level. More details about these strategies can be found in [Speirs & Barrett 1989, Chérèque *et al.* 1992, Barrett *et al.* 1990, Powell 1991].

2.3.2 Middleware-based Fault Tolerant Systems: An Example

Middleware-based systems provide facilities to compose systems together, whatever the hardware and the basic software layers are (e.g., the executive). This is in particular the case for CORBA, which also provides interoperability between interacting components developed in different programming languages.

An early example of middleware-based fault tolerant system is certainly the DELTA-4[4] system [Powell 1991], a complete object-based fault tolerant system. The various steps of its development included the definition, design and implementation of the architecture, and its validation by fault injection and formal verification. One of the initial objectives of this system was to consider distributed systems as networks of (as far as possible) off-the-shelf computers (Unix workstations). Its concepts are based on ODP (*Open Distributed Processing concepts*) and OSI (*Open Systems*

4 DELTA-4 Esprit Projects n°818/2252: Definition and Design of a Dependable Distributed Architecture

Interconnection). Distributed fault tolerance is based on objects and message passing. All mechanisms are based on a *Multicast Communication System* (MCS) providing group membership and atomic multicast protocols running on a fail-silent network attachment controller (NAC). The NAC implementation is based on hardware-implemented self-checking and memory protection. The group membership and multicast communication protocols are based on basic LAN protocols: 802.4 token bus and 802.5 token ring.

The DELTA-4 system was a reference for object-based distributed fault tolerant systems in which several conventional replication mechanisms were developed. Many similarities exist between DELTA-4 and CORBA, at least from a development viewpoint using an IDL (*Interface Definition Language*).

2.3.3 CORBA-based Fault Tolerant Systems

Given that middleware-based applications are generally distributed over multiple machines, they are more likely to experience faults than traditional centralised systems. However, standard middleware platforms such as CORBA have (until recently) provided little support for dependability. In this section, we describe why traditional process-based fault tolerance techniques are insufficient, discuss research work on dependability in CORBA-based systems, and present the recent FT-CORBA standardisation work.

Traditional process-based fault tolerance techniques are inadequate for CORBA-based systems. A method based on detecting process failure is insufficient, since the failure of a single CORBA object, or of a thread in a broker, may not cause the crash of the container process. Furthermore, standard process-based state recovery techniques are insufficient to restore the interconnected object relationships typical in distributed object systems.

The standard CORBA specification provides little support for fault tolerance. Clients will normally be informed of communication failures, since IIOP (*Internet Inter-ORB Protocol*, used for remote method invocations) provides error detection inherited from the underlying internet protocols. Detection of server crashes is more dependent on the ORB implementation: the POA (*Portable Object Adapter*, which mediates between the ORB and server objects) may detect the failure of a server and restart it automatically (possibly restoring state from a checkpoint), transparently redirecting invocations against that object to the freshly started instance. However, developers often resort to implementing a heartbeat mechanism manually to detect failures.

More sophisticated approaches to fault tolerance in CORBA-based systems are based on replicating servers, using some form of group communication service. There are a number of approaches [Felber 1998]:

- The **integration approach** incorporates an existing group communication system within an ORB. This is the approach taken by Electra [Maffeis & Schmidt 1997] and Orbix+Isis [Landis & Maffeis 1997]. Fault tolerance is transparent to clients and servers (though an extended (Application Programming Interface) is available to clients that wish to implement specialised mechanisms), but requires a customised ORB to be used.
- The **interception approach** captures messages issued by an ORB and funnels them through a group communication toolkit. This is the approach taken by Eternal [Moser &

Melliar-Smith 1997], which intercepts all IIOP traffic before it reaches the network stack and transfers it to a group communication system. Either active or passive replication strategies can be used. While the interception approach is transparent for clients and servers, it provides only a low level view of the activity of a server process, limiting the nature of the faults which can be tolerated.

- The **service approach** provides group communication as a CORBA service alongside the ORB. This is the approach taken by the *Object Group Service* and by DOORS [Natarajan *et al.* 2000]. This approach is less transparent for clients, which must make explicit use of the group service API.

More recently, fault tolerance mechanisms were provided to CORBA applications using a **reflective approach** [Killijian & Fabre 1998, Killijian & Fabre 2000]. This reflective solution relies on open compilers, namely OpenC++ [Chiba 1995] and OpenJava [Tatsubori 1999].

Recent standardisation efforts by the OMG (*Object Management Group*, an industry consortium) have led to the FT-CORBA specification, which incorporates the interception and service approaches to replication. The main points of the specification are:

- The notion of object group reference, which allows clients to invoke operations on a group of servers. Upon failure, the client ORB falls back on alternate object references contained in the group reference. The service context of invocations is augmented to allow the server ORB to detect duplicate requests and maintain at-most-once semantics (the server returns the cached result of the request).
- Mechanisms for replica management, using active, semi-active or passive replication strategies.
- Standardised interfaces for fault detection and notification by specialised CORBA objects called *Fault Detectors*, using either a push- or pull-based strategy.
- Mechanisms for logging and recovery: Network traffic related to remote invocations is recorded to be played back to the new primary upon recovery (when a passive replication strategy is used).
- Standardised interfaces which allow fault tolerance strategies to be modified dynamically (setting the maximum number of replicated instances for example).

Limitations: the FT-CORBA standard requires deterministic behaviour of application objects and of ORBs to ensure strong replica consistency. This is very difficult to achieve in practice. There are no mechanisms for handling correlated faults, or for coping with network partitioning.

2.3.4 Reflective Middleware Technology and Fault Tolerance

One aspect of research in the dependability community is the investigation of new techniques for introducing fault tolerance mechanisms in a flexible and non-intrusive manner. Indeed, given their long lifetimes, large software systems are often required to satisfy evolutions in their functional and non-functional requirements. It is important that the architectural choices made when designing or integrating such systems allow a sufficient degree of adaptation. The application of reflective techniques in middleware is a promising approach to developing adaptable and dependable distributed systems from pre-existing components, without incurring excessive integration costs.

Reflection is a useful mechanism for introducing flexibility and openness in system design. The standard approach to hiding the complexity of a software component is to view it as a black box, with implementation details being hidden from the user. In the reflective approach, certain details of the internal functioning of the component can be accessed via its *meta-interface*. This provides the *meta-level* of the system with a causally connected view of the activity of the *base-level*.

Reflection is a useful technique for providing a clean separation between an application's functional requirements (which are implemented in the base-level), and non-functional requirements such as security and fault tolerance (which are implemented in the meta-level). Reflective techniques have been used for the implementation of fault tolerance mechanisms both at the programming language level [Fabre & Perennou 1998] and in operating system kernels [Garbinato *et al.* 1995].

Current research is investigating the use of reflective notions at the middleware level. Benefits expected from this approach include:

- Separation of concerns between the application and the implementation of the dependability mechanisms (using introspection to provide transparent support for checkpointing, for example).
- The possibility of implementing more efficient dependability mechanisms, by giving them access to information from low level system layers. For instance, access to details of the scheduling mechanisms could be used for fine grained synchronisation of replication strategies, and information about the state of the network stack could provide improved error detection.
- Simple and well defined mechanisms for parameterising the system's non functional mechanisms, in particular in response to dynamic changes in the runtime environment.
- Improved support for the reuse of components implementing fault tolerance and security mechanisms.

The most advanced work to date on reflective middleware includes DynamicTAO [Roman *et al.* 1999], a reflective ORB implementation which concentrates on the need to maintain consistency when reconfiguring a system in response to changes in environmental conditions (network bandwidth, processor load for example), and the investigation of agent-based approaches to reconfiguration of large-scale systems. Another approach is taken by Jonathan [Dumant *et al.* 1998], a modular framework for the construction of flexible ORBs based around customisable binding mechanisms.

2.4 Fault tolerance and DSoS

The application-specific fault tolerance approach to be developed within the project will make use of the respective strengths of Davies' spheres of control (as the conceptual framework), CA actions and workflows. CA actions can serve as a solid starting point in developing structuring techniques suitable for complex systems of systems as they:

- Allow system developers to design, structure and provide fault tolerance of complex concurrent systems in which components cooperate and compete.
- Provide support for exception handling, which is vital for components that are not capable of rolling back and which allows actions to have multiple outcomes.
- Provide some of the generality of spheres of control and within this limited area offer a full support for maintaining consistency and achieving fault tolerance.
- Allow the tolerance of environmental faults, software design faults and crashes of nodes with transactional objects.
- Are built on a much richer concept of atomicity than traditional workflow systems and allow for a more general way for achieving fault tolerance.

Workflows, on the other hand, offer powerful features for invoking flows of activities and for constructing complex activities, which might be very useful in the context of the project. Other important features which transactional workflows provide allow one to deal with long-lived activities and with activities involving people. This is achieved by weakening some of the ACID properties. A thorough comparative study will be carried out to find in which extent these features are more suitable for developing systems of systems than general features which CA actions offer. The “action” concept at the SoS level will provide support for: multiple participating components, recursive system structuring, autonomous component behavior (e.g., asynchronous action entry, degradation), safe exception handling inside and outside actions, robust distributed support, error containment at the action level, action atomicity.

Our further intention is to investigate the advantages offered by applying reflective concepts to industry-standard middleware platforms such as CORBA brokers, in order to provide advanced application-transparent fault tolerance mechanisms. The DSoS work packages concerned with the characterisation of middleware robustness by fault injection will provide essential inputs for this work.

We expect that this research will also illustrate innovative ways of reifying to the system integrator the dependability attributes which can be customised at the middleware level. For instance, the approach could be useful in providing details of the non functional properties of an IFS, and in customising its behaviour.

In summary, the approach should provide principled mechanisms for enabling the dynamic reconfiguration of systems of systems while maintaining consistency.

Chapter 3 – Wrapping Technology

Jean-Charles Fabre, Eric Marsden *(LAAS-CNRS)*;

Alexander Romanovsky, Ian Welch *(University of Newcastle upon Tyne)*

3.1 Introduction

The use of wrapping techniques in the context of DSoS is mainly motivated by two different objectives; roughly speaking, the first one relates to the problem of interface mismatch, the second relates to protection. A wrapper is a software layer that sits around a given target component. This software layer is responsible for intercepting all service calls to the target component from other components. Ideally, this software layer cannot be bypassed for both safety and security reasons. The notion of wrapper was initially defined by the ISAT working group of DARPA (*Information Science And Technology*) as a software entity that is composed of essentially two parts: an *adaptor* providing additional services to applications, an *encapsulation mechanism* responsible for linking components.

This general definition is more oriented towards adaptation to a given operational environment rather than for the procurement of defense mechanisms. The notion of protection was raised as soon as the use of COTS (*Commercial Off-the-Shelf*) components was considered in the design and the implementation of dependable systems.

This chapter summarises the work done on both of the above aspects of wrapping technology. In the former, the wrapper provides an API (*Application Programming Interface*) to its environment that is implemented on the functions provided by the wrapped component. In the latter, the role of the wrapper is to protect the target component from erroneous interacting components and also prevent error propagation from the target component to the outside world.

3.2 Wrapping in Component-based Systems

3.2.1 Why Component Wrapping

Component-based development has been a focus of research for industry and universities because it promises decreased system complexity and reduced cost of development. Szyperski [Szyperski 1998] gives the following definition of a component that reflects the idea that components are reused and composed together to create a system :

- *A component is a unit of independent deployment*. A component encapsulates its constituent features, and is well separated from its environment and other components.
- *A component is a unit of third-party composition.* It must encapsulate its implementation and interact with its environment through well-defined interfaces.
- *A component has no persistent state*. The only state a component might have is state that does not contribute to its functionality. This implies that in most cases there will never be a need for multiple copies of components.

There is considerable confusion between objects and components. The term *component* has been used in different ways in different contexts. Current implementations of components (COM, Java Beans, etc.) exhibit only some of the properties described above. In fact components and objects are distinct from each other. Components may be implemented using objects but also may be implemented using procedures and static or global variables, or a functional language or directly using assembly language. Objects have the properties of unique identity, state and encapsulation of state and behaviour. There is no guarantee that these properties are exhibited by components.

Developing complex systems by component integration is often complicated by the following factors: introducing new or extended functional and non-functional requirements (e.g., adding functionality, improving dependability), using components in a different (wider or narrower) context or environment, heterogeneity of components. There are many reasons why components may not fit well into the integration process or match each other. For example, components are designed with a set of assumptions in mind that may not match the assumptions in the environment where they are deployed. Component wrapping is used to overcome such problems as it addresses them without having to modify the components themselves. Wrapping in component-based systems has many specific characteristics. First of all, general wrapping techniques can be developed for a particular implementation of components (e.g. for CORBA) because in this case all components follow the same interface agreements. Secondly, component wrappers can be easily re-used; this can improve productivity and give a possibility of separate validation of component wrappers therefore reducing the validation effort for the composed system.

It is important, from our point of view, to separate clearly *aims* (of using wrapping) and *means* (structured approaches to wrapper implementation) while discussing component-oriented wrapping techniques. Traditionally wrappers are used in the security domain [OMG 2000, Erlingsson & Schneider 1999]. A new emerging area is error confinement (discussed in detail in Section 3.3). There are some other applications, such as caching, management, testing.

3.2.2 How to Wrap Components

Developing structuring techniques for component wrapping is an area of active research. The choice of techniques depends mainly on the system architecture, the level on which the component functions, the way it is plugged into environment, trade-offs and, in a less degree, on the aims of wrapping. We will survey some of the existing approaches using a general view on system architecture shown in Figure 1.

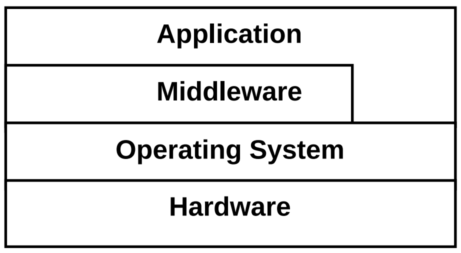

Figure 1 – Different Levels of Wrapping

A virtual machine can be used to isolate the hardware from the operating system or to isolate applications from the hardware (Figure 2). For example, various authors (e.g., [de Oliveira

Guimarães 1998]) have used a modified Java Virtual Machine to implement the traps necessary to develop wrappers. The advantages of this approach is that it is highly transparent as it requires no changes to code in the upper layers, and is very powerful. The disadvantage of the approach is the potential complexity of developing a virtual machine implementation, the performance costs, the level of abstraction may be too low for some applications because working at the instruction level may lead to mapping problems when dealing with application level abstractions, compatibility problems can arise if hardware differs from the target hardware for which the middleware/application has been compiled.

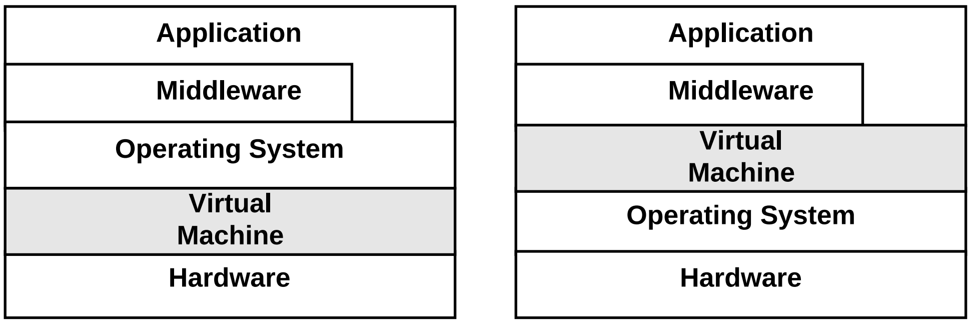

Figure 2 – Virtual Machine as a Wrapper

Another possible solution is to wrap the operating system (Figure 3). The usual way to do this is to replace the system libraries with proxies that intercept system calls. A well-developed example of this approach is *generic wrappers*. The wrappers can be used, for example, to bring components under the control of a security policy. The wrappers act as localised reference monitors for the wrapped components. An interesting example of this approach is found in [Fraser *et al.* 1999]. Here the emphasis is on binary components and their interaction with an operating system via system calls. Wrappers are defined using a Wrapper Definition Language (WDL) and are instantiated as components are activated. The wrappers monitor and modify the interactions between the components and the operating system. Generic policies (in this case for access control, auditing, intrusion detection) can be specified using the WDL.

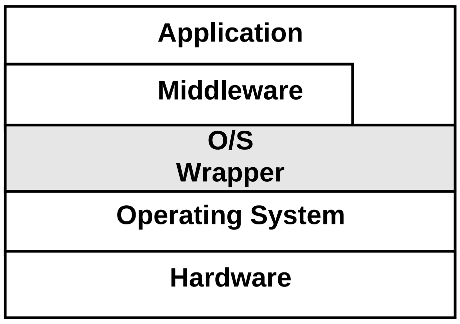

Figure 3 – Operating System Wrapper

The use of *generic wrappers* and a wrapper definition language is an attractive approach as it is flexible and is generalisable to many platforms. However, there are some drawbacks. Wrappers can only control flows across component interfaces and cannot control internal operations such as access to state or flows across outgoing interfaces (outgoing interfaces [Szyperski 1998] are defined by the set of method calls the component can make upon other components). Also the level of abstraction does not easily map to the application level, the wrappers are better suited to system level policies.

Middleware provides transparent distribution and access to services, including, for example, persistence and transactional ones, to components as it mediates interaction between components. Examples of middleware are CORBA2, CORBA3, DCOM+ and Enterprise Java Beans (EJB). Most middleware use the ideas of wrappers (Figure 4) in order to transparently insert calls to services. These are termed interceptors in CORBA and DCOM+, and proxies in CORBA3 and Enterprise Java Beans.

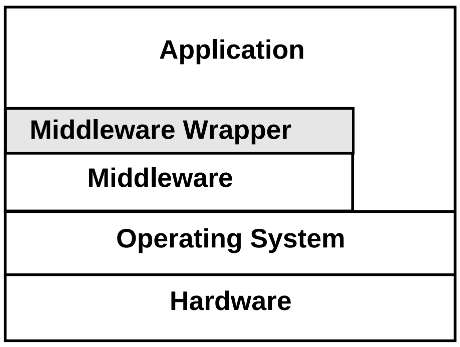

Figure 4 – Middleware Wrappers

The CORBA2 specification allows for interceptor services. These are services that can be inserted into the normal invocation path for CORBA objects. The interceptor service is registered with the ORB that then ensures that when the client sends a request to an object the request is passed through the interceptor service and on return the result also passes through the interceptor service. The interceptor can then implement the non-functional requirements such as security [OMG 2000]. DCOM+ interceptors are generated automatically by component containers and intercept cross process calls.

EJB and CORBA3 generate proxies that stand in place of the target component and allow interception of method invocations sent to the component. The degree to which these interception services and proxies are open varies. For example, ORBIX has a feature called *filtering* that is in effect a CORBA interception service. However, in general they are not open and are tuned to support particular services. This requires users to generate proxies manually or using particular tools. The other drawback is that it only applies to distributed interactions, and ignores internal events such as field access and exception raising that may be useful to have under wrapper control.

The most common way to implement application level wrappers (Figure 5) is to generate proxies for components or objects. Unfortunately this suffers from a number of problems which Holzle gives a good overview of in [Holzle 1993]: needs to create a new wrapper every time new object needs to be adapted; reusable wrappers can cause problems in languages where identity checks are performed via pointer equality checks; wrappers have a tendency to spread “infecting” everything; in languages without delegation we cannot override methods of the wrapped object; indirection can be expensive. The main point Holzle makes is that complications ensue because we have added another object to the system that has the related invariant that all interactions must be via that object, this invariant is expensive and difficult to maintain.

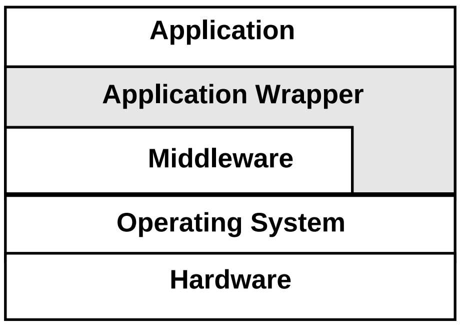

Figure 5 – Application Wrapping

These problems can be overcome by performing the wrapping at the binary level (rather than the language one) thus allowing a unification of the wrapper and the object. This is implemented in the system Binary Component Adaptation [Keller & Holze 1998]. Here, Java compiled class files are rewritten under the control of a *delta* file that specifies what changes to make to the code. Example of changes would be the addition of new methods, changes to the class hierarchy, etc. This is quite a low-level approach working at the level of the implementation and very specific to particular component implementations. Other examples of this approach are the Java bytecode rewriting toolkits (e.g., [Chiba 2000]). These toolkits provide object-oriented frameworks for writing programs that manipulate the structure of class files and load time representations of elements of class files such as methods, types, instructions etc. Java programs can then be written that describe how class files can be rewritten as late as load time. The main drawback with this approach is that the programmer has to have a detailed understanding of both the structure of class files and Java virtual machine programming. A more abstract approach is to allow programmers to work at the level of the protocols governing execution of the application and to implement changes to these protocols through the application of bytecode rewriting toolkits. *Kava* [Welch & Stroud 2000] is an example of this meta-object protocol [Kiczales *et al.* 1991] approach. *Kava* is a portable implementation for Java that allows control over behaviours such as method invocation sending, method execution, field access etc. Using *Kava* wrappers can be implemented as meta-objects using the Java language that take control the behaviour of applications at runtime. These meta-objects are bound to the application objects at the time at application load time.

There are approaches that abstract away from particular component implementations and use policy files to specify required transformations in order for components to meet new or modified requirements. For example, SASI [Erlingsson & Schneider 1999] uses a security automaton to specify security policies and enforces policies through software fault-isolation techniques. The security automaton is merged into application code by a language specific rewriter that adds code implementing the automaton before each instruction. Partial evaluation techniques are used to remove unnecessary checks. One of the problems the authors found when applying SASI was the lack of high level abstractions. SASI is very powerful and can place controls on low level operation such as push and pop allowing rich security policies to be described. However, the security policy language is low level with the events being used to construct the policies almost at the individual machine language instruction level.

3.3 Wrapping for Error-confinement

3.1.1 Motivations and Principles

The main worry of designers of complex and large systems relates to the faulty behaviour of existing component systems, including COTS components. This is particularly true when connecting legacy systems together, in the context of DSoS.

To tackle this problem two types of activities must be undertaken. The first one is to characterise the failure modes of the target system or component, the second one is to provide strong encapsulation using wrappers, providing complementary protection and error detection mechanisms. These activities must be done in sequence, since results obtained in the former step are inputs for the second one, i.e., the design and the implementation of wrappers. Because they are core components of system architecture, works concerned with wrapping for error confinement focus on executive components, namely off-the-shelf operating systems and real-time kernels. This is why most of the ideas reported in this section address the wrapping of COTS operating systems. We believe however that the various approaches discussed here can be of high interest for reusing other existing components, including legacy systems.

When weak features of the executive support have been clearly identified, wrapping leads to an encapsulated version of the executive that is the basis for the development of architectural solutions that include fault tolerance strategies at upper layers. It is worth noting that the fault tolerance strategies rely on assumptions, the coverage of which is a very important factor for dependability figures [Powell 1992].

Fault injection is often used to tackle the first aspect of the problem, recently using SWIFI techniques (*SoftWare Implemented Fault Injection*) [Kao *et al.* 1993, Hsueh *et al.* 1997, Carreira *et al.* 1998, Kroop *et al.* 1998, Fabre *et al.* 1999, Shelton *et al.* 2000] (see Section 4.2 on fault injection techniques). However, as far as legacy systems are concerned, a detailed analysis of their structure and behaviour can give enough information for the design of protection and fault containment wrappers.

Wrapping is often based on filtering inputs and outputs [Voas & Miller 1997, Voas 1998]. The use of some services can be restricted or even forbidden because of their a priori well-known weak behaviour regarding dependability. This notion of filtering for protection (a wrapper is a filter) was initially used for security purposes [Cheswick & Bellovin 1994]. The idea here is to provide access control to services in both directions, from the outside world to the wrapped component and vice-versa.

3.3.2 Filtering and Verification of Properties

As far as the confinement of errors due to accidental hardware and software faults is concerned, the efficiency of the wrapping depends very much on the colour of the box, namely the openness of the target component. Filtering can only be applied to the inputs and the outputs of the component when it is considered as a *black box*. Although this assumption is often made for existing components, having no observation and control over a component is really a very limiting factor. This is particularly true for executive components because of their complexity but also because their behaviour cannot be verified only by filtering external information. At the other extreme, a *white box* provides, beyond the source code more documents about its development process. This is of course very positive for the overall system integrator since it provides more information, in particular for entering the standardisation processes. However, even in this case, the component's behaviour in the presence of faults is something that is often ignored in the standard bodies. Improving this behaviour by means of wrappers is certainly of high interest for the system integrator in order to put in operation the system with a better level of confidence. The white box approach can thus play an important role in the development of wrappers. However, this approach is sometimes not possible from an economic and industrial viewpoint.

Regarding the weakness of current wrapper technology, recent ideas rely on the development of formal logic expressions describing the expected behaviour of executive services. These expressions are the basis for the verification of the correct behaviour at runtime either using executable assertions [Salles *et al.* 1999] or using a model checker that interprets on-line the formulae [Rodriguez *et al.* 2000]. When these formulae are expressed in temporal logic, both value and timing faults can be considered. In any case, the formulae need access to some internal information within the target component. Regarding the colour of the box, the implementation of the proposed wrappers relies on a *grey box* approach based on the notion of *reflective component*. This approach provides an intermediate approach between the full black and white approaches and is based on the concept of behavioural reflection [Maes 1987]. It enables the system integrator to consider the COTS executive as a black box, provided the COTS supplier provides an additional interface for the implementation of efficient wrappers.

The encapsulation consists in the verification of predicates that specify the correct functional and temporal behaviour of the executive component with respect to their main services (e.g. scheduling, time, synchronisation, and clock interrupts). When a predicate is violated, two types of action can be triggered depending on the severity of the diagnosed situation.

- The wrapper can be passive and only delivers an extended error status to the upper layers. In this case, the recovery action is left open to the surrounding environment.
- The wrapper can put the executive in a safety state (or shutdown of the executive) and have a corrective role.

When the failure situation can be clearly analysed as the consequence of a given combination of input parameters, the wrapper can be able to correct the activation profile or deliver an error status to the invoking component. The correction of the component's behaviour is something that is limited by the knowledge of the activation context and internal state at the wrapper level.

3.3.3 Implementation Issues

Various implementation techniques can be envisaged. The most conventional ones rely on the use of notion of front-end software package or on the use of libraries. The former is similar to the notion of *proxy* (as in the implementation of firewalls). Depending on the type of faults that must be handled, the implementation strategy must be selected carefully. When intentional faults are considered, then the wrapper must be implemented in such way that it cannot be bypassed. The library approach has a limited efficiency in this case. The library approach can cope with accidental faults but a more efficient implementation for executive software is to include the wrappers into the executive address space. When possible, this option has many benefits, since the wrapper cannot be bypassed and also because it may have access to internal information for both detection and recovery.

As mentioned earlier, a reflective approach provides a clean way of separating the executive software and the wrapper from a conceptual viewpoint. This implementation framework (see. Fig. 6) provides separation of concerns between the wrapper itself and its related target component. The meta-level of the system is the wrapper of the target component, the base-level of the reflective component being the target component. The reflective implementation framework enables the verification of predicates that need internal information of the target component (some internal events and function calls, some data structures can be observed and controlled). Beyond conventional solutions only filtering calls and input/output parameters, a reflective implementation of formal wrappers seems very promising, at least for the use of COTS real-time executives in dependable systems. The above approach can certainly be applied to many software components, executive packages being prime candidates.

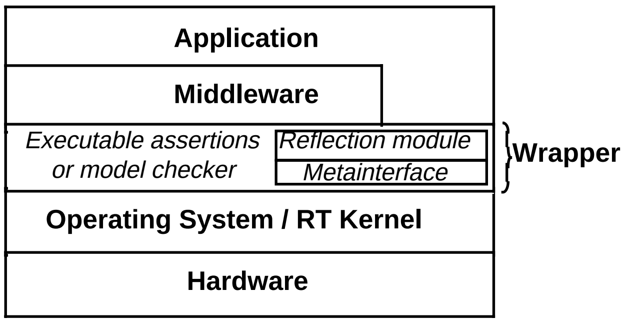
Application
Middleware
Executable assertions or model checker
Reflection module
Metainterface
Wrapper
Operating System / RT Kernel
Hardware

Figure 6 – Reflective Implementation of Wrappers

Figure 6 illustrates the basic reflective implementation framework. The wrapper implements the verification of properties by means of executable assertions or using a model checker. The reflection module provides the access to the necessary information for this verification by *reification* and *introspection* facilities. This module also provides *intercession* facilities, i.e., means to change the internal state and behavior of the wrapped component.

As far as wrapping applications is concerned, an interesting approach is the Sandboxing approach. Sandboxing is a wrapping technique used to protect against malicious faults in untrusted code. It is widely used in Internet applications, in particular to protect host machines when executing applets (mobile code executed on Java virtual machine embedded in Web browsers). The sandbox executes the untrusted code, controlling its access to resources on the local machine. Depending on the policies applied by the sandbox, it may disallow write access to the file system, limit access to the network, or refuse to allocate more than a certain amount of memory. In this way, the wrapped software cannot compromise the integrity of the host machine.

3.4 Wrapping and DSoS

The importance of subsystem wrapping is even higher in the context of the DSoS project than in component-based systems because of the following characteristics of systems of systems. Subsystems have to be always treated as ready-made items whose code is not known. It is often the case that they are not developed for integration and that their main responsibly/requirement is to function correctly in isolation and to provide local services to their users. These subsystems are quite autonomous and they should function properly even if other subsystems are down. They are developed meeting different standards, fault assumptions, agreements, types of time behaviour, etc. The reasons above can cause mismatches of several types [ConceptualModel 2000] that can be successfully dealt with by applying wrappers. Because of all these reasons there always will be a need in a unification and standardisation of both the subsystem behaviour and interfaces to apply general structuring, fault tolerance, etc. techniques at the SoS level. Wrapping software can be used in the DSoS to transform heterogeneous systems into components meeting the same interoperability “standards” (e.g., CORBA objects or EJB beans); to provide unified interfaces, including a unified propagation of the component system interface exceptions; to guarantee known and consistent fault assumptions; to protect subsystems from malfunctioning environment and vice versa; to guarantee the consistency of several interfaces of a component system.

Some of the considerations about pros and cons of different approaches to component wrapping discussed in Section 3.2.2, can be less applicable in the context of the DSoS project. Moreover it may be the case that some approaches will be difficult to apply given the nature of systems of systems, in which case new approaches will be needed for subsystem wrapping.

In practice, the error confinement mechanisms implemented by the wrappers depend very much on the nature of the target component: its type, size, and failure modes. Information on the dependability characterisation of components obtained from techniques described in Section 4 will be of great importance, both in determining the required behaviour of the wrappers, and later in testing the efficiency of the dependability mechanisms they implement.

We believe that the reflective framework presented in Section 3.3.3 is of high interest when stringent dependability requirements are needed. While it has to date only been applied to executive components, the basic principles are applicable to other off-the-shelf components. The main benefits of this approach are the separation of concerns between the wrapper and the target component, but also the limited intrusiveness within the legacy component. Nevertheless, for large and legacy systems this approach has not been envisaged yet. This is something that has to be done within DSoS.

Clearly, as far as large systems are concerned in DSoS (e.g., ATC systems or enterprise systems), wrapping is closer to the notion of firewalling as it is done for security. The firewall in this case is a linking interface (LIF) between systems that includes software packages for error confinement.

Chapter 4 – Validation Techniques

4.1 Testing of Component Systems and their Composition

Pascale Thévenod-Fosse, Hélène Waeselynck *(LAAS-CNRS)*;

Bruno Marre *(LRI)*

Testing is a widespread verification technique [Beizer 1990, Harrold 2000] that consists in executing a program by supplying it with valued inputs in the form of test inputs. Test outputs are observed according to a decision procedure (the oracle) which determines their acceptance or rejection.

Basic testing concepts are introduced in Section 4.1.1. Then, Section 4.1.2 briefly presents current functional test techniques. Section 4.1.3 is concentrated on a particular family of functional test techniques, called property-based testing, that should be of interest in the framework of DSoS. Finally, the place of testing within DSoS is discussed in Section 4.1.4.

4.1.1 Basic Testing Concepts

Save under exceptional circumstances, exhaustive testing cannot be carried out, and, therefore, a (small) subset of the input domain must be selected. Current methods for the determination of test inputs can be considered from two points of view: criteria for selecting the test inputs, and generation of the test inputs. The combination of these two points of view allows the test techniques to be pooled in four groups depicted in Figure 7.

Selection / Generation	STRUCTURAL MODEL	FUNCTIONAL MODEL
SELECTIVE CHOICE	deterministic structural	deterministic functional
RANDOM	statistical structural	statistical functional

Figure 7 – Classification of Test Techniques

The selection of the test inputs may be related to a model of either the structure of the program which defines structural (or white box) testing, or the function of the software which defines functional (or black box) testing. As regards *structural testing*, classical criteria are linked to the program control graph providing a compact view of the algorithm implemented in the program. Examples of such criteria demand testing of all instructions, all branches, or all paths. Unlike structural testing for which the reference model is the program control graph, there currently exists no standard model to describe the expected software behaviour. *Functional testing*, therefore, covers a range of techniques according to the mode of representation retained. For example, transition testing can be associated to finite state machine models; criteria associated to algebraic specifications extract input cases from the specification axioms.

Whatever the selection criterion, input generation can be deterministic or probabilistic. In the first case, which defines *deterministic testing*, inputs are determined by a selective choice so as to satisfy the test criterion retained. In the second case, which defines *statistical testing*, inputs are randomly generated according to a probabilistic distribution over the input domain; the distribution and number of test data being determined by the criterion retained [Thévenod-Fosse *et al*. 1995]. Statistical testing is intended to compensate for the imperfect connection of criteria with software faults.

4.1.2 Functional Test Techniques

Within DsoS, test suites and their decision procedures (oracles) should be designed from specifications following a functional approach rather than from coding details such as in structural approaches. The functional approach allows the component test to be grasped almost independently of component size. Indeed, functional modelling can be more or less detailed and/or based on a hierarchical break-down of the expected functions, so that the models taken individually remain of a tractable complexity. Note, however, that testing raises an implementation difficulty, which must not be overlooked: an instrumentation for test submission (drivers) and test diagnosis (oracles) must be developed. While it is easier to derive abstract test suites when the level of abstraction hides implementation details, it becomes more difficult to submit and diagnose concrete test experiments in the test execution environment.

The automation of the test selection process strongly relies on the formalisms used for the description of the available specifications. Current state of the art concerning testing from formal descriptions shows that the research community is divided according to two main aspects: one part of the community focuses on data aspects, while the other focuses on communication or synchronisation aspects.

Data aspects (and sometimes some synchronisation aspects) are generally addressed through symbolic techniques (symbolic evaluation, rewriting, constraint solving techniques) and the underlying formalisms have semantics close to first order logic. Some representatives of these formal test approaches are based on algebraic specifications [Bernot *et al*. 1991], [Doong & Frankl 1994], VDM specifications [Dick & Faivre 1993], B abstract machines [Van Aertryck *et al.* 1997], [Behnia & Waeselynck 1999], or LUSTRE descriptions [Marre & Arnould 2000].

Communication and synchronisation aspects were initially addressed in the protocol testing community. Theoretical bases strongly rely on the theory of process algebra [De Nicola & Hennessy 1984], [Brinksma 1989]. Test suites are selected from finite state machines [Chow 1978], [Fujiwara *et al*. 1991], [Lee & Yannakakis 1994], [Waeselynck & Thévenod-Fosse 1999] or from labelled transition systems [Pitt & Freestone 1990], [Tretmans 1992].

Some test methods address both data and synchronisation aspects. For example, [Dick & Faivre 1993] uses a finite state automaton derived from a VDM specification, [Thévenod-Fosse & Waeselynck 1993] uses statecharts developed with the aid of the STATEMATE tool; other methods use the control automaton derived from a LUSTRE description [Thévenod-Fosse *et al*. 1994], [Raymond *et al*. 1998], [Du Bousquet *et al*. 1999], or address data types and communications aspects from full LOTOS specifications [Gaudel & James 1998].

4.1.3 Property-based Testing

Property-based testing is a special case for functional testing that should be of great interest within DsoS. It uses the specification of a property to drive the testing process. The aim is to validate a program with respect to a target property, that is, to exercise the program and observe whether or not a property is violated. With the emphasis put on properties and not on full specification, test oracles confine themselves to the verification of the property conditions. Previous work on property-based testing is focused on safety properties (essentially invariant properties for [Jagadeesan *et al.* 1997] and [Raymond *et al.* 1998], but also reachability properties for [Marre & Arnould 2000]), and security properties [Fink & Bishop 1997].

Work on safety properties applies to programs that conform to the synchrony hypothesis. The target safety property is translated into either an ESTEREL program [Jagadeesan *et al.* 1997], or a LUSTRE program [Raymond *et al*. 1998], [Marre & Arnould 2000]. In any case, the property is used as the oracle procedure, but its impact on the selection of test inputs may be weak depending on the approach. In [Jagadeesan *et al.* 1997] and [Raymond *et al.* 1998], the selection is performed according to hypotheses on the behaviour of the external environment, in order to avoid generating scenarios that would be impossible in operation. Within these environmental constraints, how to select *a priori* test inputs that correspond to dangerous scenarios with respect to the target property, and how to define when to stop testing, remain open issues. The work of [Marre & Arnould 2000] generates test sequences from LUSTRE “testing descriptions” involving a (possibly partial) model of the object under test, some environmental constraints and the target property. An interactive decomposition mechanism applied to the testing description allows them to tune coverage of any relevant part of the testing description, including the property description.

The approach presented in [Fink & Bishop 1997] consists of a new specification language called TASPEC (Tester’s Assistant SPECification language), static slicing, path coverage criteria, and execution monitoring. TASPEC can serve as an intermediary language between a Z specification and the testing process. It has primitive constructs which enables it to be translated automatically into slicing criteria (to facilitate the extraction of all code affecting conformance to a target property) and test oracles. But the process of selecting input test data is left to the human tester.

As regards the issue of selecting input test data that correspond to dangerous scenarios with respect to a target property, a pioneer work based on the use of optimisation techniques, specifically simulated annealing, is reported in [Tracey *et al.* 1998]. First results are promising although limited to small combinatorial programs and local properties.

4.1.4 Place of Testing within DSoS

Testing activities in the DSoS project are focused on the validation of properties expected from component systems and of emergent properties coming from their composition. Both safety and fault tolerance properties will be considered. Their expression must be strongly connected to the notations and specification formalisms coming from the CM and the AD themes, and the description of the involved components should be precise enough to allow test instrumentation (drivers and oracles) to be implemented. Based on these models, our contribution will put emphasis on improving the combination of functional and non-functional test inputs, the combination being guided by the target property to be validated.

We adopt a more liberal definition of safety properties than [Jagadeesan *et al.* 1997], [Raymond *et al*. 1998] and [Marre & Arnould 2000]. Safety properties are any high level requirements related to the most critical failure modes of one component system. For improved composability, it must be verified that the required property cannot be destroyed by the result of interactions that are engaged with the other component systems. Testing will be used as a means to determine whether or not the constraints (e.g., pre- and postconditions) placed at the component interface are sufficient to ensure the absence of some unacceptable failure behaviour in the integration environment. As explained above, few authors have addressed the issue of selecting input test data that correspond to dangerous scenarios. We will investigate whether the optimisation approach proposed by [Tracey *et al.* 1998] can be merged with another sampling approach, namely statistical testing, to offer a tractable solution to this issue. The dangerous scenarios that “stress” the target property are expected to combine both functional and non-functional inputs.

Fault tolerance properties are emergent properties whose implementation is tightly connected to architectural design choices. As will be explained in the next section, fault injection is a privileged approach for validating such properties. Since this approach raises issues that are close to the testing problem (see Section 4.2.3), it will be investigated whether integration testing policies and test selection criteria can be used to improve the definition of fault injection experiments.

4.2 Fault Injection

Jean Arlat *(LAAS-CNRS)*

Fault injection corresponds to the artificial insertion of faults into a real or simulated target computer system [Carreira *et al.* 1999]. Actually, fault injection experiments are intended to yield three benefits:

- An understanding of the effects of real faults and thus of the related behaviour of the target system.
- An assessment of the efficacy of the fault tolerance mechanisms included into the target system and thus a feedback for their enhancement and correction (e.g., for removing designs faults in the fault tolerance mechanisms).
- A forecasting of the faulty behaviour of the target system, in particular encompassing a measurement of the efficiency (coverage) provided by the fault tolerance mechanisms.

As such, in spite of the continuous progress made both in verification, especially using formal methods, and in modelling, using stochastic processes (encompassing imperfect coverage models as well) — see Section 5, fault injection provides a pragmatic complementary approach towards the dependability validation of fault-tolerant systems.

Initially applied to centralised systems (especially dedicated fault-tolerant computer architectures) [Avizienis & Rennels 1972], fault injection has then addressed distributed systems [Kao & Iyer 1995], and also, more recently, the Internet [Labovitz *et al.* 1999]. Also, the various layers of a computer system (ranging from hardware [Martínez *et al.* 1999] to software: executive [Kao *et al.* 1993], middleware [Pan 2000], application [Tsai & Singh 2000]) can be targeted by fault injection.

It is important to note that such an approach, based on controlled experiments, is very much suitable when non-development items (e.g., COTS components) are integrated and interact into the system being considered, which is particularly the case for the kind of systems addressed by the DSoS project. Indeed, COTS components exhibit usually little information on their development process, on their architecture and on their actual failure behaviour, to allow for a detailed analysis to be carried out. Fault injection thus allows for objective data to be obtained in particular on their failure modes so as to be able to elaborate suitable architectural solutions — such as fault containment and error processing mechanisms (e.g., wrappers) — to better detect/tolerate their errors (see Section 3.3). Such insights and measurements are also useful to feed any higher-level models that may be developed for dependability verification or evaluation purpose.

This section further exemplifies the role of fault injection into the dependability validation process. First, the types of faults and the related injection techniques are discussed, then after a clear identification of the scope of relevance of fault injection, examples of results are given that show the type of insights that one can derive from fault injection experiments and their impact on the validation process. Finally, the place of fault injection within DSoS is briefly addressed.

4.2.1 Injecting Faults

Two main questions arise when dealing with fault injection:

- What kind of faults are to be injected?
- How to inject faults?

To date, two main kinds of faults have been targeted by fault injection activities: physical (hardware) faults and design (software) faults. Indeed, little work has been devoted to devising controlled experiments addressing human-related interaction faults (being they accidental or intentional).

As is also the case for other efforts made in dependable and fault-tolerant computing, most mature advances and techniques concerning fault injection deal with physical fault injection. Moreover, supported by the fault statistics available, transient faults have dominated this effort (over permanent faults). Dedicated tools are available to flip bits at the pins of an IC, alter the power supply or even bomb the system/chips with electromagnetic interferences (EMI) or heavy-ion radiations [Karlsson *et al.* 1998]. Although significant work has been carried out dealing with software mutation [Voas & McGraw 1998], software bug injection gave rise to much less investigations and is still largely an open research issue. Injecting software faults is a more difficult task. Also, although they are the result of a (permanent or hard) bad design, their perceived faulty behaviour is often transient (or soft) [Gray 1986].

Nevertheless, it is important to point out that what really matters when probing the behaviour of a target system in presence of faults is much less the faults themselves than the consequences provoked/observed. Indeed, in practice, similar error patterns often originate from various distinct causes (faults). Besides it allows to overcome the high costs and practical restrictions (e.g., intrusiveness, repeatability, controllability, etc.) attached to the use of physical injection techniques with recent hardware technologies (e.g., high density chip packing and high clock speed), such a matter of fact is one main driver for the emergence of the so-called software implemented fault-injection (SWIFI, in short) technique. SWIFI flips bits in processor register cells or in memory, thus simulating the consequences of either transient hardware faults and to some extent software faults as well, [Madeira *et al.* 2000]. In particular, several studies have shown that such simple bit flips allows for generating errors similar to those provoked by physical injection (e.g., see [Kanawati *et al.* 1995]). Recently several powerful and generic tools have been developed that support this technique; Xception [Carreira *et al.* 1998] and MAFALDA [Rodríguez *et al.* 1999] are two examples of such tools. Moreover, SWIFI is generic, as it can be applied at the various layers of the target system (processor, microkernel, OS, middleware, application).

Both physical and software techniques assume that at least a prototype of the target system is available. An alternative, usable as early as the design phase, is to apply fault injection on a simulation model. This way, the main design choices and architectural solutions, including fault tolerance mechanisms, can be assessed in the early phases of the development process. Simulation-based fault injection is also generic as it can be applied at various levels (device, gate, RTL (Register Transfer Level), Processor Memory Switch, Network, etc.) [Kaâniche *et al.* 1998]. Of course, there is an inevitable trade-off between i) the level of detail/abstraction of the simulation model and ii) the length (duration) of the simulation runs that can be carried out. As VHDL (Very high speed integrated circuits Hardware Description Language) is of widespread use, significant efforts were carried out on fault injection into VHDL simulation models both worldwide (e.g., see [DeLong *et al.* 1996]) and also within the framework of the ESPRIT PDCS and DeVa projects, both from the evaluation and verification viewpoints (e.g., see [Jenn *et al.* 1995] and [Arlat *et al.* 1999], respectively). In addition, simulation also provides useful support for the objective characterisation of fault models to be applied with the SWIFI technique (e.g., see [Yount & Siewiorek 1996]).

4.2.2 Relevance and Impact of Fault Injection

While it allows for a detailed analysis of the behaviour of a target system in presence of faults, fault injection can only contribute partially to the overall dependability assessment of a fault-tolerant system or a fault tolerance mechanism. For testing purpose, in principle, devising fault injection experiments is relatively straightforward: one has simply to select the kind of faults (the fault model) foreseen during the design of the system or of the specific mechanism; however developing both effective and high coverage experiments is not always easy in practice. Statistical techniques (as those developed for software testing [Thévenod-Fosse *et al.* 1995] — see also Section 4.1) can be used to guide the fault selection process (e.g., see [Arlat *et al.* 1999]). When fault tolerance efficiency (such as the error detection coverage) has to be evaluated, things become more complex: fault injection experiments alone cannot provide a proper answer. Indeed, besides the selection of a representative[5] fault model, one must consider with which relative frequency each fault category within the fault model is likely to occur to obtain an unbiased estimation of the coverage. Such an issue, that was identified as early as in [Rennels & Avizienis 1973], was formally analysed in [Powell *et al.* 1995] and then refined in [Cukier *et al.* 1999] by considering both frequentist and Bayesian estimates.

Moreover, when dependability measures for a specific target system need to be evaluated, one has to account also for the frequency of occurrence of the faults, and fault injection alone cannot help on

[5] What kind of faults/errors will actually occur in operation?

that neither. For example, stochastic models can be developed that feature parameters accounting not only for component failure and repair distributions, but also for fault handling and error processing (e.g., coverage and latency) [Arlat *et al.* 1993]. The measurements obtained during fault injection experiments can then be used to value the fault tolerance parameters of the models. Putting it in another way, dependability evaluation is thus a compound approach that must closely associate analytical and experimental evaluations. By providing objective results on the faulty behaviour of the target system, fault injection has thus a key role in this process.

Similar problems (as the one of having access to trustworthy information on the fault occurrence profile), arise in many fields. For example, one is faced with an identical problem for evaluating the performance of a computer: the ideal is to test it with the workload it is meant to run. Still, such a workload may be unknown, or it could be varying during lifetime. The standard solution to this problem has been to define benchmarks, by choosing some specific (but widely accepted and recognised) set ups, programs and conditions, trusting that the derived measurements that provide good clues on the behaviour of the target system. Such performance benchmarks are now well established, and are widely used, for example to rate database or web servers. A good example is the SPEC benchmarks developed for the Unix world [Gray 1993]. However, in the dependability area, no such comprehensive benchmarks exist yet, although some advances have been made (e.g., see [Koopman *et al.* 1997, Mukherjee & Siewiorek 1997]). Furthermore, a dedicated project is now been launched within the IST Programme to tackle this issue.

Nevertheless, even if they are somewhat restricted to a specific set of circumstances (e.g., *x*% of the injected faults of category *y* are detected when the target system is submitted to activity *z*), fault injection actually provides useful insights on the complex behaviour of a computer system in the presence of faults. Two examples taken from our previous studies are provided hereafter. The first one concerns the experimental validation of a distributed fault-tolerant architecture developed within the framework of the ESPRIT Delta-4 project [Powell 1994]. The second one is related to the characterisation of the failure modes of a COTS microkernel [Fabre *et al.* 1999].

Figure 8 shows a typical graphical representation of the experimental results obtained when submitting an instance of the Delta-4 (hardware and software) architecture to physical transient faults injected on IC pins with the MESSALINE tool [Arlat *et al.* 1990]. The percentages indicate the coverage values associated to *predicates* characterising the behaviour of the system in presence of faults: `Error` (activation of the injected fault), `Detected error` (the error is detected by the hardware mechanisms), `Tolerated error` (the error is tolerated by the internode communication protocol), `Failure` (the error cannot be tolerated, being it detected or not). For example, 94% of the injected faults were actually activated as errors[6] (which reveals a very high testing power for pin-level fault injection), and 86% of these errors were detected. One main feature of this graph concerns the inclusion of a transition that might have been omitted in an *a priori* model of system behaviour: this transition corresponds to the 13.5% of errors that were not detected, but still were tolerated at system level. Such a transition depicts a form of “extra-coverage” that results from the intrinsic resiliency of any real design due to non deliberate redundancies — in that case, of the Delta-4 internode atomic multicast protocol (AMp). These experiments complemented the other efforts (formal verification and analytical modelling) used to validate the architectural designs supporting

[6] This information is obtained by means of current sensors attached to each injection device connected to the faulted pins.

the Delta-4 architecture. The results were used both as design aids and for dependability evaluation purpose (in connection with Markov chains [Arlat *et al.* 1993]). In particular, the traces obtained during the fault injection experiments were very much helpful to the designers for developing the successive versions of the AMp.

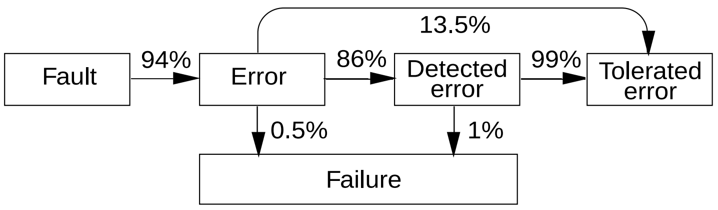

Figure 8 – Results of Fault Injection Experiments with MESSALINE

Figure 9 illustrates the types of results that were obtained when subjecting a Chorus/ClassiX microkernel instance (composed of basic executive services such as synchronisation, memory, scheduling, etc.) to a series of SWIFI experiments conducted using MAFALDA [Rodríguez *et al.* 1999].

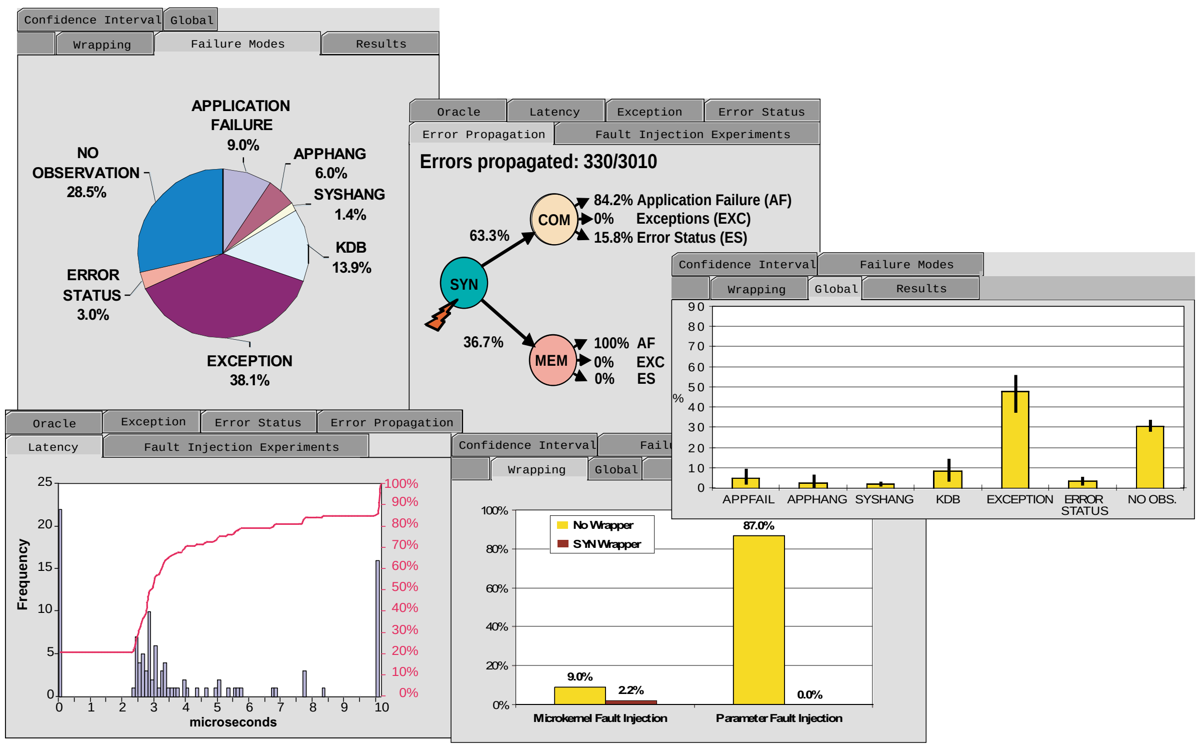

Figure 9 – Sample of Measures Obtained with MAFALDA

The four left windows show the detailed results concerning the synchronisation component; among these, the bottom right window compares the results obtained for the standard and wrapped component. The right most window shows the global results for all the faulted components of the Chorus/ClassiX microkernel. For instance, the pie diagram shows the failure modes observed when about 3000 faults (transient single bit flips) were injected in the code segment of the standard synchronisation component. Regarding the failure modes, about 50% of the errors were successfully detected by the microkernel error detection mechanisms (“error status”, “exception”, “kernel debugger [KDB]”), while a hang (“system hang [SYSHANG]”, “application hang [APPHANG]”) occurred in 7.4% of the cases. Nevertheless, 9% of the errors led to an incorrect service (“application failure [APPFAIL]”). Finally, the “no observation [NO OBS]” category (29%) corresponds to errors that had no observable consequences although the injected faults were actually activated. As far as error propagation between functional components is concerned, most of the errors propagated from the synchronisation to the communication component, which means that the communication component is more dependent on synchronisation than the memory component. Error latency is illustrated for exceptions: about 30% of the exceptions were raised immediately, while about 50% of them were distributed in an interval from 2.4 to 4 microseconds. Concerning wrapper efficiency, the synchronisation wrapper reduced the rate of application failures from 9% to 2.2% when injecting bit flips in the code segment of the microkernel. The wrapper detected all the errors when fault injection was performed on the parameters of the service requests to the microkernel components, resulting in a reduction from 87% to 0%. The 87% of application failure observed when applying fault injection to the parameters passed to the synchronisation primitives is very surprising. The reason for this singular behaviour is that Chorus basic synchronisation mechanisms (essentially mutex semaphores) do not make any verification of input parameters. This design option at the very basic microkernel layer was made on purpose in order to leave total freedom on this respect to the upper layer designer. Recently, MAFALDA was also used to analyse the LynxOS microkernel [Fabre *et al.* 2000].

4.2.3 Place of Fault Injection within DSoS

Due to its major impact on the validation process, as illustrated in the previous sections, fault injection is definitely part of the validation framework to be developed within DSoS. Two main forms of activities related to fault injection will be carried out.

The first one concerns the methodological point of view. It is intended to foster the relationship between formal specification, software testing and testing of fault tolerance. In fact, to some extent, fault injection can be viewed as a testing method for the fault tolerance mechanisms, where in addition to the usual functional activity, the input domain extends to the kind of specific inputs the fault tolerance mechanisms are meant to deal with: the faults. Elaborating on previous related research (e.g., see [Thévenod-Fosse *et al.* 1995, Avresky *et al.* 1996, Gaudel & James 1998]), the intent is to guide the testing with the help of a formal description of the fault tolerance mechanisms, both for i) selecting the input patterns (including dynamic links between faults and activity patterns), and ii) defining the observation predicates.

The second form of activity is more technology-oriented. It is intended to extend the application of the fault injection methodology developed for assessing COTS microkernels (i.e., the MAFALDA tool) to the CORBA middleware layer that will support the implementation of the linking interfaces (LIF). Indeed, such a layer is intended to embody the mechanisms that will be part of the fault tolerance framework (wrapping and recovery) developed by the project (see Chapters 2 and 3). The planned analyses encompass the conduct of fault injection experiments both on the standard and on the fault-tolerant middleware layers.

As such, both efforts will build upon and contribute to the conceptual model elaborated by the project.

4.3 Model Checking for Compositional Validation

Irfan Zakkiudin *(DERA)*

Model-checking [Clarke *et al*. 1999] has emerged, in recent years, as a powerful technology for system validation and verification. The fundamental idea behind model-checking is to:

1. Generate a finite representation of a system – in effect a model of the system.
2. Validate critical properties of the system by exhaustively exploring all possible behaviours of the finite model.

This simple and brutal approach has some appealing advantages, model-checking is:

– *Highly automated*, since the exhaustive exploration is done entirely by the model-checking tool.

– *Revealing*, since if the critical property fails, then the tool finds the behaviour leading to the violation.

– *Incisive*, since the exhaustive exploration can find property violations which may have gone undetected by non-exhaustive validation methods.

A disadvantage of exploring all reachable states is that the number of states can grow very fast, and the model-checker's task can quickly become intractable. This limitation is the proverbial *state explosion problem.* Effective applications of model-checking technologies, for system verification and validation, require solutions to the state explosion problem. Model-checking work, in the DSoS project, will develop techniques to mitigate the state explosion problem, thus enabling model-checking to support compositional validation.

The rest of this section is organised as follows. First, we give a brief overview of model-checking technologies and their limitations. Then, we introduce the process algebra CSP [Hoare 1985][Roscoe 1998], and the FDR model-checker [Formal Systems (Europe) Ltd 1976][Roscoe 1998] that supports CSP. The explanation of CSP and FDR will (it is hoped) be sufficient to describe how the property, required to be validated, can support alleviating the state explosion problem and we will, very briefly, describe how this can done. We conclude with a discussion of compositional model-checking techniques in a CSP/FDR context.

4.3.1 Model-checking Technologies and their Limitations

For the last decade research in model-checking has grown steadily and it is now a thriving research community, rich with ideas [Emerson & Sistla 2000]. The bulk of model-checking research addresses verifying models of *discrete reactive systems.* If two or more processes interact to affect each other's state, then this is a reactive system. There are a few model-checkers for analysing:

– Timed reactive systems [Bozga *et al*. 1998].

– Probabilistic reactive systems [Baier *et al*. 1997].

- Hybrid reactive systems, where states and state transitions may be parameterised by various continuous variables and differential equations between variables, the principal example is HyTech [Cassez & Laroussine 2000].
- Combinations of timed and probabilistic reactive systems [Kwiatkowska 1999], and others.

However, most model-checking tools are for discrete reactive systems. In such systems, the model's total state arises from the discrete interactions of many processes. Typically, these processes will themselves have small and simple local states. There are now very many model-checking tools for discrete reactive systems, the better known ones are FDR, SPIN [Holzmann 1991] and SMV [McMillan 1993].

An interesting departure from the paradigm of reactive systems is Dan Jackson's model-checker, the Alloy Constraint Analyser [Jackson 2000]. This tool can verify properties of relations defined over sets, using a Z-like language to specify the models. This approach can, for example, potentially be used to analyse a set of routes across a network, for properties such as loop freedom.

To understand the field we need to understand the basic aspects of a model-checking technology, these are:

1. The formalism used to specify models and their properties.
2. Methods by which the model's behaviours are represented.

We discuss them in the next two sections, we conclude our brief summary of the field with a discussion of the limitations of model-checking.

4.3.1.1 Specification Languages Used by Model-checkers

We have said that most model-checkers verify discrete reactive models and these models consist of interacting processes. The modelling language must be capable of expressing process interaction and concurrency; it usually consists of:

- An imperative style language to express sequential model behaviour.
- A shared variable model of concurrency to implement process interaction.

Some model-checkers have used established formalisms for shared variable concurrency as an inspiration. Examples include SPIN's use of the original shared variable version of CSP [Hoare 1978] and Murphi's use of UNITY [Chandy & Misra 1988].

The CSP/FDR technology is distinct from the mainstream, shared variable and imperative, style in two respects:

1. CSP is a process algebra, which uses shared events to implement process interaction (CSP is discussed below). FDR in fact uses a machine readable form of the blackboard language, called CSP_M.
2. The CSP algebra is augmented by embedding it in a *functional* programming language (all part of CSP_M).

A model-checker also needs a specification of the property to be verified of the model. The conventional approach is to use temporal logics [Pnueli 1977]. Examples of these are Linear Temporal Logic (LTL), branching temporal logic (CTL), Alternating Temporal Logic (ATL) and other variants. These logics are used to specify the two classic types of property, namely:

- *Safety*, being the absence of something bad.
- *Liveness*, being the availability of something good.

Alternatives to temporal logics include the μ-calculus, which is first order logic augmented with a least fixed point operator [Berezin *et al*. 2000].

Once more, the CSP/FDR approach is distinctive. FDR uses CSP to define models but properties of the models are also expressed in CSP. FDR verifies models by checking that the CSP model *refines* the CSP property, this is discussed below (see Section 4.3.2.3). There is no notion of safety and liveness, as such, in CSP, but it is common to verify safety and liveness properties with FDR by checking refinement in the appropriate denotational semantics, namely:

- Safety properties are verified by checking refinement in *traces semantics*.
- Liveness properties are verified by checking refinement in *failures semantics*.

It may be argued that temporal logics are an intuitively clear way to specify properties and that to express properties in CSP is counter intuitive, and so clumsy. However, the difficulty is not great and the benefits make the effort very worthwhile. The benefits include:

- Free support for compositional reasoning, obviating the need to develop complex compositionality rules [Namjoshi & Trefler 2000], this is outlined in Section 4.3.4.
- The ability to use the property to support reducing the state space, this is discussed in Section 4.3.3.

4.3.1.2 Representing Models Behaviours

A model-checker will need to compile the system model into a representation inside the model-checking engine. The model-checking engine will manipulate this to representation to do the verification.

The various ways to code a model's behaviour, in the model-checking engine, are now also very many, but the most common is the Binary Decision Diagram (BDD). This representation arose from the initial application of model-checking to circuit verification. A BDD is a particularly efficient way to code the Boolean values that characterise a circuit's state. It was subsequently discovered that a BDD could also code a Boolean function, and this Boolean function could represent the state transition relation for a circuit. Using BDDs to code Boolean functions is the origin of symbolic model-checking. It should be clear that BDD-based technologies are best suited for hardware verification, or for verifying similar synchronous systems. BDDs have themselves diversified into many flavours, including MDDs and QDDs [Wolper & Boigelot 1998].

The canonical representation of a model's behaviour is a labelled transition system, which is simply a state machine. For more asynchronous systems (such as protocol verification) an approach based on explicitly using state machines tends be superior. Tools such as FDR and SPIN are explicit state machine tools. While FDR simply used state machines, SPIN represents a state machine as a Buchi automata. Both tools employ an on-the-fly philosophy, in that they try to generate the state machine as they verify it, rather than first generating the whole machine and then verifying it.

As research in model-checking grows, more techniques are emerging to code and verify a model's behaviour. These include using Boolean formulas [Biere *et al*. 1999] instead of BDDs and labelled transition systems [Lazic & Nowak 2000], which are parameterised by type variables (for exploiting data independence).

4.3.1.3 Limitations of Model-checking Technologies

While model-checking thrives and its applications grow, model-checking technologies have a number of limitations. We discuss these limitations below:

1. *The state explosion*. This is commonly regarded as the most significant limitation and it probably receives the most attention from the research community. While the state explosion is a major limitation, model-checkers have been applied in a number of industrially significant contexts. This is possible because there are now many techniques to address this limitation. A skilled user can perform impressive verifications by carefully abstracting the problem and by using the various techniques that now exist. While an expert can often control the state explosion, it does require time and ability on the part of the user. Exploiting the benefits of model-checking requires enabling the state explosion to be controlled as quickly and easily as possible, for a full discussion see [Zakiuddin 1999]. The techniques discussed in 4.3.3 will automate one more attack on the state explosion. Recent work on automated abstractions is another attack on the state explosion, which is now receiving a lot of interest [Shankar 2000].

2. *The restriction to finite state systems.* A model-checker will always check a finite state space. In theory, this means that if a potentially unbounded problem is studied, then the analysis will be limited to a finite case of that problem. In practise a number of techniques have been developed to address this limitation, this is also an area that attracts significant research [Abdulla & Jonsson 2000]. Recent work on data independence provides powerful mathematical techniques, in the form of parametric operational semantics, to study the models parameterised by arbitrary types [Lazic & Nowak 2000]. Inductive approaches have long been studied [Kurshan & McMillan 1989]; and recent work on combining data independence and induction [Creese & Roscoe 2000] extends the state-of-the-art in interesting ways.

3. *The restriction to 'control intensive' applications*. 'Control intensive' is a more applied way of saying that model-checkers analyse discrete reactive systems. Model-checkers are clumsy at dealing with 'data intensive' problems – precisely because these cannot easily be modelled as a collection of discrete interacting processes. In the mid-to-late 90s, there was some research seeking to combine model-checking and theorem proving to remedy this limitation. The aim was to use model-checking for the control intensive parts and theorem proving for the data intensive parts. This line of investigation did not prove to be fruitful, because it is not easy to separate the two parts. As far as we are aware, this area is not an

active subject of research; though it is indirectly studied through recent work on automated abstractions [Shankar 2000]. For a different approach, recall the Alloy Constraint Analyser as an interesting departure from the reactive systems paradigm. Potentially, it offers a capability to verify data intensive problems. Hoare's recent work on Unified Theories [Hoare & He 1999] defines a framework for combining theories, like Z and CSP. This framework can be used to combine FDR and the Alloy Constraint Analyser to verify problems that combine control rich and data rich behaviour. Concrete problems that are both control rich and data rich include reliable multicasting and topology-aware distributed algorithms.

In our opinion, this limitation of model-checking is more significant, for industrial applications, than the limitation to finite state models.

4. *Separation from the subject of analysis.* A model-checker verifies a model of a system. It does not study the system itself. This obvious fact lies behind one of the hardest and least studied limitations of model-checking. A model of a system will invariably be an abstraction; it typically will elide a great deal. Often justifications for such abstractions are either informal on not given at all. This attitude may seem reprehensible, but recall that in many industrial applications the value of model-checking is seen in its ability to find design flaws. Thus if a design flaw is found, then value has accrued, but if it is not, then no claim of correctness in made. DERA's applications of model-checking (FDR) have been to support safety cases of military systems. Our approach to ensuring the veracity of the model is to obligate models to be subject to rigorous review by people not responsible for generating the models. Another approach to bridging the gap between the subject and its model is to compile the subject source code into the model-checking engine. This approach has two significant drawbacks:

   - Automatically generated models suffer from a very serious state explosion. Recall that the state explosion is often controlled by a skilled user carefully 'pulling out' a minimal model.
   - It is simply not possible for many applications, and this is the main reason we cannot deploy it in DERA. Our models are drawn from large sets of engineering documentation.

4.3.2 Model-checking Using CSP and FDR

Having discussed the technology field and our place in it, we turn to the specific technology deployed on the DSoS project, namely CSP and FDR.

4.3.2.1 The CSP Algebra

Communicating Sequential Processes [Hoare 1985][Roscoe 1998] is a process algebra for specifying and reasoning about concurrent systems. Processes in CSP are built from atomic, black box events; CSP processes perform sequences of events. The set of events a process performs is called its *alphabet*. Concurrency is implemented, in CSP, by shared event synchronisation. If P and Q are two CSP processes, then process P || Q, is a concurrent process, in which:

- Events in the alphabet of P, but not Q, occur only under the control of P, and similarly, events in the alphabet of Q, but not P, will occur only under the control of Q;
- Events in the alphabets of both P and Q must be performed as a single event, shared by both P and Q.

CSP is an algebra, thus CSP processes can be composed by a wide variety of algebraic operators, to yield further CSP processes. The ||-operator, mentioned above, is an example of an algebraic operator for composing CSP processes. The CSP algebra's expressiveness yields its utility for modelling and validation. The compositionality of the CSP algebra, as we shall see, supports compositional validation.

4.3.2.2 Ordering CSP Terms

CSP is an algebraic syntax for specifying processes. To use CSP for specifying, modelling and verifying systems it is necessary to compare processes. Comparing means identifying when processes are the same or when one process is *less* than another process. But how can we order or equate syntactic terms? The answer is to assign semantics to the syntactic terms. The semantics will have natural orderings and the order relations on the semantics can order their associated CSP processes. Ordering processes is the second ingredient for compositional reasoning in CSP.

The algebraic theory of CSP is supported by three main semantics, *viz*:

- *Denotational semantics* – where a process is assigned a set of values.
- *Operational semantics* – where a process is assigned a labelled transitions system, or a state machine.
- *Algebraic semantics* – where a process is assigned a set of processes, to which it is equivalent, according to some transformation rules.

There are other ways of comparing process, in addition to the above, but for the purposes of compositional validation with FDR we only need the denotational and operational semantics.

One way to order processes is to use the denotational semantics to assign a set of values to a process and then to use the set inclusion operator to order the processes. The denotational semantics are themselves of three types, but we can explain concepts using just the *traces* semantics. In this semantics, a process is assigned the set of sequences of events it can perform. Two processes P and Q can be ordered when the traces of one process are a subset of the traces of the other. For example, if

$P = a \rightarrow STOP, \qquad traces(P) = \{\langle a\rangle\},$

$Q = a \rightarrow Q, \qquad traces(Q) = \{\langle a\rangle, \langle a,a\rangle, \langle a,a,a\rangle,\ldots\},$

then P and Q can be ordered because traces(P) $\subseteq$ traces(Q).

This is conventionally written Q $\sqsubseteq$ P, and read as P *refines* Q, and this simply means that the behaviour of P is contained in the behaviour of Q.

Finally, we need to observe a few mathematical properties of our operators and orderings. We state them here:

1. *Monotonicity*. If P ⊑ Q||R and Q ⊑ Q', then P ⊑ Q'||R.
2. *Transitivity*. If P ⊑ Q and Q ⊑ R, then P ⊑ R.

Thus we have the basic mathematical ingredients to support a compositional validation technology – all we need is the tool.

4.3.2.3 Introducing FDR

FDR is a model-checker for CSP; it is also often described as a *refinement* checker. FDR verifies whether or not one CSP process is a refinement of another, in any one of the denotational models, by exhaustive exploration. Thus FDR is a model-checker that checks for refinement. If the refinement relation between two processes fails, then, as with all model-checkers, FDR finds at least one behaviour responsible for the violation.

FDR's basic capability can be used in a number of ways, but the most common is to verify whether a system implements a specification, or supports a property. To do this, we:

- Create a CSP model of the system, call it SYS.
- Define the specification, or property, in CSP, call it SPEC.
- Use FDR to check, in one of the denotational semantics, whether or not SPEC ⊑ SYS.

To understand how to do property based validation, we need to know a little about how FDR works.

FDR will compile CSP processes into state machines and then do model-checking on these state machines. Recall that state machines are an operational semantics for CSP. Thus FDR verifies refinement assertions over the denotational semantics, by manipulating its representations of processes, which are in fact operational semantics. Consequently, FDR depends on CSP processes having both operational and denotational semantics.

Recall that the perennial problem in model-checking is the state explosion problem. In FDR terms, the state explosion is that the state machine, which FDR manipulates for verification, becomes intractably large. FDR can mitigate this explosion with its *compression operators* [Roscoe *et al*. 1995]. These operators reduce the state machine (the operational semantics), without changing the associated denotational semantics. As a consequence, the state space is reduced, but the truth of any refinement assertions remains unchanged. For example,

> if *comp1* and *comp2* are two compressions, SPEC, P1 and P2 are CSP processes, then the following are equivalent:
>
> 1. SPEC ⊑ P1 || P2
> 2. SPEC ⊑ *comp1*(P1 || P2)
> 3. SPEC ⊑ *comp1*(*comp2*(P1) || *comp2*(P2))

While these compression operators can be effective, they are difficult to use effectively. In some cases the cost of computing the reduced state machine can greater than the benefit of having a reduced state machine.

To make effective use of the compression we use property based validation.

4.3.3 Property Based Validation Using CSP and FDR

Suppose we are using FDR to check the refinement assertion:

$$\mathrm{SPEC} \sqsubseteq \mathrm{SYS},$$

in the traces semantics. This means that FDR will check all behaviours of SYS to see if it can do any sequence of events not permitted by SPEC. It turns out that we can use the property, SPEC, to transform the right hand side of the refinement assertion so that compressions can be applied more effectively. Thus we are using the property to support validation, and in particular to reduce the state space. We will give a minimally technical description of how to do this, for the trace semantics.

The essential idea is to transform SPEC into a process that monitors SYS. This monitoring process, which we call W_SPEC, is composed in with SYS using the ||-operator. If SYS can perform a sequence of events, not permitted by SPEC, then W_SPEC will emit a distinguished event, which we call *fail*. The refinement assertion above will now be:

$$\mathrm{STOP} \sqsubseteq \mathrm{SYS} \parallel \mathrm{W\_SPEC} \setminus \bullet ,$$

Where:

- $\bullet$ is the alphabet of SYS, namely all the events that occur in the model.
- $fail \notin \bullet$.
- ‘\’ is the hiding operator, given a CSP process, P, and subset, A, of the alphabet of P, then P \ A is the same process as P, except that events from A have been hidden, they occur invisibly at the discretion of P.
- STOP is a special CSP process, it does no events, so its traces is the empty set.

This new refinement assertion has all events hidden, except the *fail* event. Now, this new refinement assertion will be violated only if the *fail* event can happen, and this corresponds to SYS violating SPEC. If the *fail* event cannot happen, then SYS must satisfy SPEC. Recall that the motivation for transforming the refinement assertion was to aid applying the compression operators. It is a fact of the compression operators that the new right hand expression:

$$\mathrm{SYS} \parallel \mathrm{W\_SPEC} \setminus \bullet ,$$

is much more amenable to compressing, than the former right hand expression. The reason for this is that the compression operators work by removing hidden events from the state machine. Clearly the state machine for the expression above, with only the *fail* event visible, will be much smaller than the state machine for SYS. Thus the property, SPEC, has been used to aid validation, by reducing the state explosion.

We conclude this section with an example of generating W_SPEC from SPEC. But to do this we need to introduce one more CSP operator, the []-operator. If P and Q are CSP processes, then P [] Q is the process that can either do an initial event from P and proceed as P, or an initial event from Q, and proceed as Q. If P [] Q is composed with another processes using the ||-operator, then the choice, of P or Q, is made by the other process. Now suppose:

SPEC = a $\rightarrow$ STOP, and

• = {a,b}, then

W_SPEC = a $\rightarrow$ STOP [] b $\rightarrow$ *fail* $\rightarrow$ STOP [] a $\rightarrow$ a $\rightarrow$ *fail* $\rightarrow$ STOP

Note that W_SPEC is a function of SPEC and • only, it does not depend on SYS, this is always the case. Furthermore, it is possible to systematically compile W_SPEC.

4.3.4 Compositional Model-Checking Using CSP and FDR

CSP and FDR can also be used to support compositional reasoning. Compositional model-checking with FDR depends on the compositionality of CSP. In a model-checking context, compositional reasoning is another way to alleviate the state explosion problem. Of the operators in the CSP algebra, the ||-operator causes the largest blow up in states. As a rule, the size of the state space of P1 || P2 is a product of the sizes of the state spaces of P1 and of P2. Thus the focus of compositional reasoning is to decompose verifications down through the ||-operator. Suppose,

SYS = P1 || P2, and we want to verify if SPEC $\sqsubseteq$ SYS, then we can instead just verify,

SPEC $\sqsubseteq$ P1 and SPEC $\sqsubseteq$ P2, then

P1 || P2

$\sqsupseteq$ by the monotonicity rule on P1

SPEC || P2

$\sqsupseteq$ by the monotonicity rule on P2

SPEC || SPEC

$\Rightarrow$ by the transitivity rule

SPEC || SPEC $\sqsubseteq$ P1 || P2.

In the traces semantics, P || P = P. Using this and another application of transitivity, we get:

SPEC $\sqsubseteq$ P1 || P2.

Chapter 5 – Dependability Evaluation of Large Systems

Mohamed Kaâniche, Karama Kanoun *(LAAS-CNRS)*

5.1 Introduction

Two approaches can be used to support dependability evaluation of systems of systems: (i) analytical modelling, and (ii) measurement-based assessment. Modelling and measurement are complementary. Measurement experiments can provide estimates for the parameters used in the model as well as demonstrate the validity of the modelling assumptions and evaluations. On the other hand, models can be used to design measurement experiments in order to gather the evidence needed to support model elaboration and processing.

In this chapter, we summarise the state-of-the art related to dependability modelling and operational assessment of computer systems, and identify the research directions that will be followed for the dependability evaluation of systems of systems.

5.2 Analytical Modelling

Dependability evaluation of large computing systems based on analytical modelling requires the description of the failure and repair behaviour of hardware and software system components and the numerous interactions between them, resulting in complex models. Depending on the dependability measures to be evaluated, the modelling level of detail can furthermore increase this complexity. State-space models, in particular homogeneous Markov chains, are commonly used to model the dependability of fault-tolerant systems. The latter are able to capture various functional and stochastic dependencies among components and allow evaluation of various measures related to dependability and performance (i. e., performability measures) based on the same model, when a reward structure is associated to them. The resulting model is referred to as Reward Markov model.

To facilitate the generation of large state-space models, high-level specification languages such as GSPNs (Generalized Stochastic Petri Nets with timed and immediate transitions) are generally used. Also a reward structure can be associated to GSPNs leading to Generalised Stochastic Reward Petri Nets (GSRPNs). GSRPNs allow a compact representation of the behaviour of systems involving synchronisation, concurrency and conflict phenomena. Also, they provide means for structural verification of the model and can be automatically converted to Reward Markov models.

Surveys of the problems related to techniques and tools for dependability and performance evaluation can be found for example in [Reibman & Veeraraghavan 1991] and [Trivedi *et al.* 1994]. In this state-of-the-art report, we concentrate on techniques for mastering the complexity of **state-space models** associated with large-scale systems.

One of the major drawbacks for the use of state-space models, in dependability and performance evaluation of real systems, is the state explosion problem. Several techniques have been published to address model largeness; they can be grouped into two categories as suggested in [Trivedi *et al.* 1994]: “largeness avoidance” and “largeness tolerance” techniques.

In the rest of the section, we will first present the two categories of techniques before commenting on their combined use.

5.2.1 Largeness Avoidance Techniques

These techniques try to circumvent the generation of very large models. The basic idea is to construct small sub-models that can be processed in isolation. The results of the sub-models are integrated in a single overall model that is small enough to be processed. Among these techniques, we have:

- The decomposition technique for stochastic reward net models [Ciardo & Trivedi 1993].
- The behavioural decomposition technique [Balbo *et al.* 1988].
- The hybrid hierarchical modelling technique (fault trees and Markov chains) [Balakrishnam & Trivedi 1995].
- The resolution technique based on quasi-decomposability [Bobbio & Trivedi 1986].
- The net-driven decomposition technique for numerical computation of the stochastic Petri net (based on the reachability graph decomposition) [Pérez-Jiménez & Compos 1999].
- The data structure technique for he efficient Kronecker solution of GSPNs [Ciardo & Miner 1996, Ciardo & Miner 1999].

These techniques usually address model processing (in an exact way or using approximate solutions) and lead undeniably to a gain in memory (by avoiding complete storage of the model) and in computation time. However, from a practical point of view and to the best of our knowledge, most of these techniques are efficient when the sub-models are loosely coupled and become hard to implement when interactions are too complex. For the dependability modelling of fault-tolerant systems, multiple and complex interactions between system components have to be considered because of the dependencies induced by component failures and repairs.

5.2.2 Largeness Tolerance Techniques

The main objective of these techniques is to master the complexity of the generation of the global system model through the use of *concise specification methods* and *automated generation* of the model. The specification consists of a set of rules allowing an easy construction of the Markov chain. These rules may be either a) specifically defined for model construction (see e.g., [Goyal *et al.* 1986], [Carrasco & Figueras 1986], [Bouissou 1993] or [Berson *et al.* 1991]) or b) more well-known and formal rules, based on GSPNs or their off-springs. Most of the time, the specification methods are integrated into tools (respectively, SAVE, METFAC, FIGARO, TANGRAM and SMART for the above referenced methods).

Specification and construction methods based on GSPNs are modular. The basic idea is to generate the model of a modular system by composition of the sub-models of its components; they are referred to as *model composition techniques*. In addition to the GSNP formalism, these techniques i) make use of composition rules for sub-model interfacing and integration, ii) impose some restrictions or add new operators to the formalism to facilitate model generation, master the complexity and/or preserve the formalism properties. Several model composition techniques have been published. Some of them are intended to be general methods (see e.g., [Meyer & Sanders 1993], [Rojas 1996], [Kanoun & Borrel 1996], [Fota *et al.* 1999b], [Bondavalli *et al.* 1999], [Rabah & Kanoun 1999]); whereas the others are just applied to a specific system (see e.g., [Muppala *et al.* 1992]).

In [Meyer & Sanders 1993], two composition operators are defined (*join* and *replicate*) to compose system models from repeated structures (through the replicate operator) in an asynchronous manner (through the join operator). In [Rojas 1996], a more complete set of composition operators for the generation of Stochastic Well-formed Nets (SWN), (i.e., GSPNs permitting the identification of symmetry by means of symmetric reachability graph) from the SWN of it components has been defined. These operators preserve the functional structure of the model and support several types of communications between components. This approach is intended to support the modelling of distributed and parallel systems where both synchronous and asynchronous communications are required. However, it addresses only a class of systems that can be modelled by SWN.

The modelling approaches presented in [Kanoun & Borrel 1996], [Fota *et al.* 1999b], [Bondavalli *et al.* 1999] and [Rabah & Kanoun 1999] are more specially devoted to performability evaluation and are based on GSRPNs. [Bondavalli *et al.* 1999] and [Rabah & Kanoun 1999] define modelling approaches where the system model is composed of two levels.

[Bondavalli *et al.* 1999] addresses phased-mission systems: the upper-level models the mission phases and the lower-level details the behaviour of the system inside each phase. [Rabah & Kanoun 1999] addresses multipurpose multiprocessor systems: the upper level models the service concerns, it is related to the application running on the architectural system (it defines the service levels, the needs in terms of resources, the maintenance policy, etc) whereas the lower-level models the behaviour of the architecture components with their interactions.

[Kanoun & Borrel 1996] and [Fota *et al.* 1999b] present approaches for constructing a GSPN of a complex system from the GSPNs of its components taking into account the interactions between the components. These approaches are referred to as *block modelling approach* and *Incremental approach* respectively and are outlined hereafter.

The **block modelling** approach in [Kanoun & Borrel 1996] defines a framework for modelling the dependability of hardware and software fault-tolerant systems taking into account explicitly the dependency between the various components. These dependencies may result from functional or structural interactions between the components or from interactions due to global system fault-tolerance, reconfiguration and maintenance strategies. The modelling approach is modular: the behaviour of each component and each interaction is represented by its own GSPN, and the system model is obtained by composition of these GSPNs. The GSPNs of the components and interactions are called block nets. Composition rules are defined and formalised through identification of the interfaces between the component and interaction block nets. In addition to modularity, the formalism brings flexibility and re-usability thereby allowing for easy sensitivity analysis with respect to the assumptions that could be made about the behaviour of the components and the resulting interactions.

The main advantage of this modelling approach, lies in its efficiency for modelling several alternatives for the same system. These alternatives may differ by their architecture or by the fault tolerance and maintenance strategies. One can clearly identify from the beginning the components and interactions that are specific or common to all alternatives. The blocks related to components and interactions that are common are thus developed and validated only once. It has been applied to a Regional Centre of the French Air Traffic Control system for which 16 alternative architectures (based on a reference architecture) have been modelled in an efficient way [Kanoun *et al.* 1999]. Most of the block nets developed for the reference architecture have been re-used for the 15 other alternatives, and only a few block nets have been either modified or newly developed.

The block modelling approach is very suitable to model systems where the number of components is not very high and the number of interactions is relatively limited so as to be able to construct a model in a flat way.

In **the incremental modelling approach**, the model is built and validated in an *incremental* manner [Fota *et al.* 1999b]. At the initial step, the behaviour of the system is described taking into account the failures of only one selected component, assuming that the others are in an operational nominal state. The failures of the other components are then integrated progressively in the following steps of the modelling process. At each step, a new component is added and the GSPN model is updated by taking into account the impact of the additional assumptions on the behaviour of the components that have been already included in the model. The component's behaviour is described by sub-models (called *modules*) and the interactions between components are modelled using *module-coupling mechanisms*. To efficiently build the GSPN model, a set of guidelines related to module-coupling mechanisms have been defined to facilitate the construction of a structurally valid GSPN. In addition, as for the block modelling approach, the associated rules aim to assist the user in the implementation of the system behaviour and failure assumptions into the GSPN formalism, while mastering the model complexity and avoiding modelling errors.

At each integration step, the GSPN model is validated. The validation is carried out at the GSPN level (structural verifications) and also at the Markov level in order to check the different scenario represented by the model. When the Markov chain size increases, the exhaustive analysis of the Markov chain is impractical. In this case, sensitivity analyses are used to check the validity of the model assumptions. During model specification, emphasis is put on enhancing model readability and compactness (via the use of a reduced number of places and transitions and well-defined modelling constructs), as well as flexibility and reusability of parts of the model to ensure easy modification of the model when new assumptions are considered.

This approach has been successfully used to model the dependability of the main subsystems of the French air traffic control computing system referred to as CAUTRA [Fota *et al.* 1999a]. The CAUTRA is implemented on fault-tolerant computers geographically distributed over five Regional Centres (those modelled by the above presented block modelling approach) and one Centralised Operating Centre, connected through a Wide Area Network. The models (GSPN and Markov) constructed for the different subsystems are very complex. For example the GSPN of the Radar Data Processing and the Flight Plan Data Processing Systems has about 100 places and 500 transitions and corresponds to a reduced Markov chain of about 25 000 states.

The Incremental approach is suitable for the dependability modelling of computer systems with numerous hardware and software components, multiple interactions, and complex procedures for fault tolerance and restoration.

5.2.3 Largeness Avoidance-and-tolerance

It is worth noting that the two categories of techniques (largeness avoidance and largeness tolerance) are complementary and, most of the time, all techniques use both of them putting more emphasis on one or the other. For example, even though largeness avoidance, in the sense that the whole system model is not generated and processing is performed on the sub-models, is not the prime concern of largeness tolerance techniques, state-space reduction constitutes a real concern. Generally, in most of the largeness tolerance techniques, rules for model generation are also defined in such a way that they generate the less superfluous states by construction.

Finally, largeness avoidance by means of truncation of the least important states (i. e., states with very small probabilities) can be used to complement efficiently largeness tolerance techniques as in [Muppala *et al.* 1992].

5.3 Measurement-based Assessment

There is no better way to understand the dependability characteristics of an operational computing system than by direct measurement, analysis and assessment. Measuring a real system means monitoring and recording naturally occurring errors and failures in the system while it is running under user workloads. Analysis of such measurements can provide valuable information on actual error/failure behaviour, quantify dependability measures and identify system bottlenecks.

There is a wide variety of research related to the analysis of error and failure data collected from operational computer systems. The main issues addressed are summarised in this section, considering standalone as well as networked computer systems.

5.3.1 Measurement

Measurement involves three main steps: (1) data collection, (2) data validation and (3) data processing.

Data collection consists in the definition of *what* to collect and *how* to collect the data. The kind of data to be collected is directly linked to the kind of behaviour to be analysed and to the quantitative measures to be evaluated to characterise such behaviour. There are two ways to obtain the data: online automatic logging and human manual logging. Many computer systems such as IBM mainframes, Unix/Solaris and Windows NT based systems include an event-logging software in the operating system. This software records events occurring in the various components of the system including detected errors as well as other system events such as reboots and shutdowns. The main advantage of automatic online logging is its ability to record a large amount of information, in particular with respect to transient errors that cannot be done manually. Disadvantages are that an online log does not usually include information about the cause and propagation of the error or about offline diagnosis or maintenance. Also under some crash scenarios, the system may fail too quickly for any error messages to be recorded. Therefore, other sources of information are generally needed in conjunction with online logging to provide the missing data.

Data validation consists in analysing the collected data for correctness, consistency, and completeness. This consists in particular in filtering-out invalid data and in coalescing redundant or equivalent data. Usually, the collected data contains a large amount of redundant and irrelevant information, as well as incorrect or incomplete information. Such problems have been observed in several studies, e.g. those reported in [Kaâniche *et al.* 1990, Levendel 1990, Buckley & Siewiorek 1995, Thakur & Iyer 1996]. Thus, preliminary investigation of the data must be performed to classify this information and to facilitate subsequent analyses. In online error logs, a single fault in the system can result in many repeated errors occurring close in time. Indeed, as the effects of the fault propagate through a system, hardware and software detectors are triggered resulting in multiple error records. To ensure that the subsequent analyses will not be biased by these repeated reports, related events should be coalesced into a single event. This observation led to the development of the tuple, or cluster concept [Tsao & Siewiorek 1983]. Several techniques have been proposed for event tupling, e.g. in [Tsao & Siewiorek 1983, Iyer *et al.* 1986, Hansen & Siewiorek 1992]. A comparative analysis of some of these techniques is reported in [Buckley & Siewiorek 1996].

Once invalid data is filtered-out and data is coalesced, the basic dependability characteristics of the measured system can be identified through data processing.

Data processing consists in performing statistical analyses on the validated data to identify and analyse trends and to evaluate quantitative measures that characterise dependability. Descriptive statistics can be derived from the data to analyse the location of faults, errors and failures among system components; the severity of failures; the time to failure or time to repair distribution; the impact of the workload on the system behaviour; the efficiency of error detection and recovery mechanisms; etc. Commonly used statistical measures in the analysis include frequency, percentage, probability distribution, and hazard rate function. Basic statistical techniques can be applied to estimate the mean, variance, and confidence intervals of the parameters characterising these measures (see e.g. [Kendall 1977] for a comprehensive study of statistical methods). More sophisticated analyses can also be performed using trend tests [Kanoun & Laprie 1996] and analytical modelling [Reibman & Veeraraghavan 1991].

5.3.2 Study of Failures in Operational Computer Systems

There is a large body of literature related to the analysis of failures occurring in operational computer systems. These analyses cover several aspects including:

- Investigation of the classes of errors/failures reported in the field, their relative importance, and the correlation among errors;
- Analysis of error/failure inter-arrival times and recovery times distributions;
- Analysis and modelling of software and hardware error detection & recovery mechanisms and their efficiency

In this section, we outline some of the results and lessons learnt from these studies. Some of these results are detailed and discussed in [Iyer & Tang 1996].

Early dependability-related experimental studies based on field data focused on the measurement, analysis and modelling of transient or intermittent errors in digital systems. In particular, the analysis of errors collected on DEC computer systems reported in [Siewiorek *et al.* 1978, McConnel *et al.* 1979] showed that transient failures occur at least an order of magnitude higher than permanent failures. These studies also found that the inter-arrival time of transient errors follows a Weibull distribution with decreasing error rate, instead of the traditional exponential distribution generally assumed for modelling permanent failures. This distribution was shown to fit the software failure data collected from an IBM operating system [Iyer & Rossetti 1985]. The predominance of transient failures has been observed also for the software. In [Gray 1986], an analysis of operational failures in Tandem computers revealed that most software faults are soft faults that can be tolerated by simply restarting the failed process with different input conditions.

The dependence of system failure behaviour on system activity has been pointed out in the early 1980s. An early study at the University of Illinois, focussed at the analysis of system failures and workloads on IBM machines, showed that the average system failure rate was strongly correlated with the average workload on the system [Butner & Iyer 1980, Iyer & Rossetti 1985]. Similar results were presented in [Castillo & Siewiorek 1981], based on measurements at Carnegie-Mellon University, and in [Moran *et al.* 1990]; they are based on error data collected on VAX8600 computers.

Correlation among system component failures is another source of stochastic dependency that has a significant impact on computer systems dependability modelling and evaluation. Such correlation might exist among software failures, hardware failures as well as among both. For example, in [Iyer & Velardi 1985], nearly 35% of software failures observed on the MVS/SP operating system running on an IBM 3081 machine were hardware related. Measurements on VAXclusters [Tang *et al.* 1990, Wein & Sathaye 1990] and Tandem GUARDIAN machines [Lee *et al.* 1991] found that correlated failures exist significantly in distributed systems. Most of correlated failures observed on the VAXcluster study were related to the network interconnecting the VAX machines.

Software, and design faults in general, have become the major dependability bottleneck during the last fifteen years, This is confirmed by field data collected on largely deployed systems, e.g. [Gray 1990, Moran *et al.* 1990, Cramp *et al.* 1992, Wood 1995]. For example, the analysis of field failures in Tandem computer systems between 1985 and 1990 [Gray 1990] revealed that more than 60% of system failures reported in 1989 were due to software. Accordingly, many experimental studies focussed at the analysis of software related errors. The analysis and modelling of software errors to provide feedback to the development process have been addressed in several papers, e.g., [Musa *et al.* 1987, Lyu 1995, Kanoun *et al.* 1997]. Several experimental studies have been published to support the analysis of software error characteristics and the modeling of the impact of software failures on dependability, e.g. [Kanoun & Sabourin 1987, Levendel 1990, Kenney & Vouk 1992, Sullivan & Chillarege 1992, Kaâniche *et al.* 1994, Chillarege *et al.* 1995]. The issues addressed in these papers include: (1) categorisation of software errors; (2) monitoring of software processes and products through the use of trend tests and statistical quality control, (3) evaluation of quantitative measures characterising the software failure intensity and time to failure using reliability growth models.

Although there is a wide variety of research that is based upon the analysis of error and failure data collected from operational computer systems, very few studies have addressed interconnected systems. Distributed and network-based computing systems, are notoriously difficult environments in which to detect and diagnose faults. The research reported in [Maxion & Feather 1990] provided an elaborate discussion on the problems related to the diagnosis and analysis of network anomalies and proposed a methodology based on the monitoring of performance degradation or deviations from expected behaviour as a means for characterising dependability related problems in networked environments. Dependability analysis of networked applications in distributed environments requires the definition of representative fault models and meaningful dependability measures that characterise the system behaviour as perceived by the users. An example is provided in [Wood 1995] where a framework for analysing and measuring availability as perceived by the users in client-server distributed environments is defined and illustrated with experimental data collected from different sources (customers, network component providers, university studies, etc.) The lack of real data collected from the monitoring of networked applications and failures is one of the reasons for the lack of published results on dependability analysis and modelling of interconnected systems. Examples of such real data can be found in [Thakur & Iyer 1996], where an environment for collecting and analysing the failures in a network of Unix workstations is presented, and in [Kalyanakrishnan *et al.* 1999b], where failure data from a network of 70 Windows NT mail servers is analysed. Both studies concern measurement performed in local area network environments and are based only on event logs recorded by the hosts. In this context, it is difficult to diagnose the cause of a failed client request: it might result from a server failure, a network component failure, or simply to network performance degradation. The use of data collected with network management tools for the monitoring of network components and systems, in addition to the events recorded by the operating system, should help to improve failure and dependability analysis in networked environments, (see e.g., [Orfali *et al.* 1996, Harnedy 1998] for a presentation of network management functions, standards, and tools).

With the ever-increasing use and rapid expansion of the Internet, several recent research efforts focussed on Internet measurement, with the perspective of evaluating network performance metrics from an end-users perspective [Paxson *et al.* 1998, Matthews & Cottrell 2000]. Internet measurement studies focussed at dependability analysis is an emerging area of active research. Two main directions have been followed: (1) study of Internet hosts availability and reliability, and (2) study of failures affecting the Internet backbone infrastructures.

Internet hosts reliability studies address issues such as: (a) What is the probability that a user’s request to access an Internet host succeeds? (b) What percentage of hosts remain accessible to the user at a give moment? (c) What are the major causes of access failures as seen by the user? In [Sriram 1993, Long *et al.* 1995, Kalyanakrishnan *et al.* 1999a] availability and reliability measures are evaluated for a sample of Internet hosts by repeatedly polling the hosts from different sites. Generally, these studies consider the Internet and the underlying infrastructure as a “black box”. Studies focussing on Internet backbone infrastructures are aimed at the analysis of Internet routing problems and network related failures. For example, in [Labovitz *et al.* 1999], two categories of Internet failures are analysed: failures in the connections between service provider backbones, and failures occurring within provider backbones. This analysis is based on data collected from experimental measurements of largely deployed wide area networks and on data obtained from the operational records of an Internet service provider.

5.4 Conclusion

This chapter covered two complementary approaches that will be used to support SoS dependability evaluation: analytical modelling and measurement-based assessment.

With respect to analytical modelling, the main problem remains the state-space explosion when considering a large number of systems connected together. The work that will be performed within DSoS will address model construction based on a compositional approach. To master progressively the complexity of the model, we will explore methods for progressive refinement of a GSPN: starting from a high level model, some places and transitions can be expanded to include more details. Such approaches have been investigated for Non Stochastic Petri Nets. Our aim is to extend them to GSPNs.

As regards measurement-based assessment, although there is a wide variety of research that is based upon the analysis of error and failure data collected from operational computer systems, very few studies have addressed interconnected systems. Today’s network environments integrate a set of heterogeneous multi-vendor hardware, software and network resources, and are subject to continuous change in network configuration, applications, traffic, etc. The analysis of faults and errors in this context and the evaluation of their impact on dependability raise several open questions: (1) What is the definition of a failure? (2) What are the quantitative measures to be defined to assess the dependability of the system? (3) What kind of data needs to be collected to evaluate these measures? (4) What is the infrastructure needed to perform data collection and processing for dependability measurements? (5) How to evaluate comprehensive dependability measures for the global system based on measures evaluated for the components? (6) What kind of feedback can be provided to improve the design? DSoS aims to address these questions. The local area computing network at LAAS will be used as an experimental environment to support this research.

References

Chapter 1 – Architecture and Design

[Abowd *et al*. 1995] G. Abowd, R. Allen and D. Garlan "Formalizing Style to Understand Descriptions of Software Architecture", *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 4(4), pp. 319-364, 1995.

[Allen 1997] R. Allen, *A Formal Approach to Software Architecture*, PhD thesis, CMU, 1997.

[Allen & Garlan 1997] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 6(3), pp. 213-249, 1997.

[Allen *et al*. 1997] R. Allen, R. Douence and D. Garlan, "Specifying Dynamism in Software Architectures", in *Proc. Foundations of Component-based Systems Workshop,* 1997.

[Allen *et al*. 1998] R. Allen, D. Garlan and J. Ivers, "Formal Modeling and Analysis of the HLA Component Integration Standard", in *Proc. ACM SIGSOFT'98 Symposium on Foundations of Software Engineering* (*FSE*), pp. 70-79, 1998.

[Blair *et al*. 2000] G.S. Blair, L. Blair, V. Issarny, P. Tuma and A. Zarras, "The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms", in *Proc. Middleware'2000: IFIP/ACM International Conference on Distributed Systems Platforms*, LNCS 1795, pp. 164-184, 2000.

[Beder *et al*. 2000] D. M. Beder, A. Romanovsky, B. Randell, C.R. Snow and R.J. Stroud, "An Application of Fault Tolerance Patterns and Coordinated Atomic Actions to a Problem in Railway Scheduling", *ACM Operating System Reviews*, 34, 4, pp 21-31, October 2000.

[Borrmann & Newberry-Paulish 1999] L. Borrmann and F. Newberry-Paulish, "Software Architectures at Siemens: The Challenges, our Approaches and Some Open Issues", in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp. 539-544, KAP, P. Donohe ed., 1999.

[Bratthall & Runeson 1999] L. Bratthall and P. Runeson, "Architecture Design Recovery of a Family of Embedded Software Systems", in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp. 3-14, KAP, P. Donohe ed., 1999.

[Buschmann *et al.* 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *A System of Patterns: Patterns-Oriented Software*, John Wiley & Sons, 1996.

[Canal *et al.* 1999] C. Canal, E. Pimentel and J. M. Troya, "Specification and Refinement of Dynamic Software Architectures", in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp. 107-126, KAP, P. Donohe ed., 1999.

[Cheung & Kramer 1996] S.C. Cheung and J. Kramer, "Checking Subsystem Safety Properties in Compositional Reachability Analysis", in *Proc. 18th International Conference on Software Engineering* (*ICSE*), pp. 144-154, 1996.

[Cheung *et al*. 1997] S.C. Cheung, D. Giannakopoulou and J. Kramer, "Verification of Liveness Properties using Compositional Reachability Analysis", in *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE*), LNCS 1301, pp. 227-243, 1997.

[Dashofy *et al*. 1999] E.M. Dashofy, N. Medvidovic and R.N. Taylor, "Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures", in *Proc. 21st International Conference on Software Engineering* (*ICSE*), pp. 3-12, 1999.

[DeLine 1999] R. DeLine, "A Catalog of Techniques for Resolving Packaging Mismatch", in *Proc. Symposium on Software Reusability*, 1999.

[DiNitto & Rosenblum 1999] E. Di Nitto and D. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures", in *Proc. 21st International Conference on Software Engineering* (*ICSE*), pp. 13-22, 1999.

[Emmerich 2000] W. Emmerich, *Engineering Distributed Objects*, J. Wiley & Sons, 2000.

[Garcia *et al*. 2000] A. Garcia, D. Beder and C. Rubira, "A Software Architecture Based on Patterns for Exceptional Condition Handling", in *Proc. 5th IEEE International Symposium on High Assurance Systems Engineering* (*ISHASE*), 2000.

[Garlan *et al*. 1994a] D. Garlan, R. Allen and J. Ockerbloom, "Exploiting Style in Architectural Design Environments", in *Proc. ACM SIGSOFT'94 Symposium on Foundations of Software Engineering* (*FSE*), pp. 175-188, 1994.

[Garlan *et al*. 1994b] D. Garlan, R. Allen and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard", in *IEEE Software*, pp. 17-26, 1994.

[Garlan *et al*. 1997] D. Garlan, R. Monroe and D. Wile, "ACME: An Architecture Interchange Language", in *Proc. CASCON'97*, 1997.

[Garlan 2000] D. Garlan, "Architectures for Pervasive Computing", in *Proc. 4th International Software Architecture Workshop (ISAW)*, pp. 42-45, 2000.

[Guo *et al*. 1999] G.Y. Guo, J.M. Atlee and R. Kazman, "A Software Architecture Reconstruction Method", in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp. 15-34, KAP, P. Donohe ed., 1999.

[Hofmeister *et al*. 1999] C. Hofmeister, R.L. Nord and D. Soni, "Describing Software Architecture with UML", in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp. 145-160, KAP, P. Donohe ed., 1999.

[Issarny 97] V. Issarny, "Configuration-Based Programming Systems", in *Proc. SOFSEM'97: Theory and Practice of Informatics*, LNCS 1338, pp. 183-200, 1997.

[Issarny *et al.* 1998a] V. Issarny, C. Bidan and T. Saridakis, "Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the Aster Prototype", in *Proc. 4th International Conference on Configurable Distributed Systems* (*ICCDS*), pp. 275-283, 1998.

[Issarny *et al*. 1998b] V. Issarny, C. Bidan, and T. Saridakis, “Characterizing Coordination Architectures According to Their Non-Functional Execution Properties”, in *Proc. 31st Hawaii International Conference on System Sciences* (*HICSS*), pp. 275-285, 1998.

[Inverardi & Wolf 1995] P. Inverardi and A. Wolf, “Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model”, *IEEE Transactions on Software Engineering* (*TSE*), 21(4), pp. 373-386, 1995.

[ISOIEC 1995] ISO/IEC, *Reference Model for Open Distributed Processing Part 1: Overview*, Technical Report 10746-1, 1995.

[Kazman *et al*. 1994] R. Kazman, L. Bass, G. Abowd and M. Webb, “SAAM: A Method for Analyzing the Properties of Software Architectures”, in *Proc. 16th International Conference on Software Engineering* (*ICSE*), pp. 81-90, 1994.

[Klein et al. 1999] M. H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci and H. Lipson, “Attribute-based Architecture Style”, in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp.225-244, KAP, P. Donohe ed., 1999.

[Kruchten 1995] P. Kruchten, “Architectural Blueprints: The 4+1 View Model of Software Architecture”, *IEEE Software*, 12(6), pp. 42-50, 1995.

[Kruchten 1999] P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999.

[Kuusela 1999] J. Kuusela, “Architectural Evolution”, in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp.471-478, KAP, P. Donohe ed., 1999.

[Le Metayer 1996] D. Le Métayer, “Software Architecture Styles as Graph Grammars”, in *Proc. ACM SIGSOFT’96 Symposium on Foundations of Software Engineering* (*FSE*), pp. 15-23, 1996.

[Lewandowski 1998] S. C. Lewandowski, “Frameworks for Component-Based Client/Server Computing”, in *ACM Computing Surveys*, 30(1), pp. 3-27, 1998.

[Luckham *et al*. 1995] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan and W. Mann, “Specification and Analysis of System Architecture Using Rapide”, *IEEE Transactions on Software Engineering* (*TSE*), 21(4), pp. 336-355, 1995.

[Magee *et al*. 1995] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, “Specifying Distributed Software Architectures”, in *Proc. 5th European Software Engineering Conference* (*ESEC*), LNCS 989, pp. 137-153, 1995.

[Magee *et al*. 1997] J. Magee, A. Tseng and J. Kramer, “Composing Distributed Objects in CORBA”, in *Proc. International Symposium on Autonomous Decentralized Systems* (*ISADS*), 1997.

[Medvidovic & Taylor 2000] N. Medvidovic and R. N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, *IEEE Transactions on Software Engineering* (*TSE*), 26(1), pp. 70-93, 2000.

[Medvidovic *et al*. 1999] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, “A Language and Environment for Architecture-based Software Development and Evolution”, in *Proc. 21st International Conference on Software Engineering* (*ICSE*), pp. 44-53, 1999.

[Mehta *et al.* 2000] N.R. Mehta, N. Medvidovic and S. Phadke, “Towards a Taxonomy of Software Connectors”, in *Proc. 22nd International Conference on Software Engineering* (*ICSE*), pp. 178-187, 2000.

[Microsoft 1998] Microsoft, *DCOM: A Technical Overview*, Technical Report, Microsoft Corporation, 1998, http://www.microsoft.com/ntserver/appservice/.

[Moriconi & Riemenschneider 1997] M. Moriconi and N. Riemenschneider, *Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*, Technical Report SRI-CSL-97-01, SRI Int’l, 1997.

[Moriconi *et al*. 1995] M. Moriconi, X. Qian and N. Riemenschneider, “Correct Architecture Refinement”, *IEEE Transactions on Software Engineering* (*TSE*), 21(4), pp. 356-372, 1995.

[OMG 1995] OMG, *The Common Object Request Broker: Architecture and Specification Revision 2.0*, 1995.

[OMG 1997a] OMG, *UML Notation Guide*, 1.1 Edition, 1997.

[OMG 1997b] OMG, *UML Semantics*, 1.1 Edition, 1997.

[OMG 1997c] OMG, *Object Constraint Language Specification*, 1.1 Edition, 1997.

[OMG 1998] OMG, *CORBA Services: Common Object Services Specification*, 1998.

[Oreizy *et al*. 1999] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum and A.L. Wolf, “An Architecture-Based Approach to Self-Adaptive Software”, *IEEE Intelligent Systems*, pp. 54-62, May/June 1999.

[Perry 1998] D. Perry, “Generic Architecture Description for Product Lines”, in *Proc. Workshop on Software Architecture for Product Families*, 1998.

[Perry & Wolf 1992] D. Perry and A. Wolf, “Foundations for the Study of Software Architecture”, in *Software Engineering Notes*, 17(4), pp. 40-52, 1992.

[Pree & Koskimies 1999] W. Pree and K. Koskimies, “Rearchitecting Legacy Systems”, in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp.51-64, KAP, P. Donohe ed., 1999.

[Riemenschneider *et al*. 2000] R.A. Riemenscheneider, J. Salasin and A. van Lamsweerde, “From System Requirements to System Architecture”, in *Proc. 4th International Software Architecture Workshop (ISAW)*, pp. 1-6, 2000.

[Robbins et al. 1998] J.E. Robbins, N. medvidovic, D.F. Redmiles and D.S. Rosenblum, “Integrating Architecture Description Languages with a Standard Design Model”, in *Proc. 20th International Conference on Software Engineering* (*ICSE*), pp. 209-217, 1998.

[Saridakis & Issarny 1999] T. Saridakis and V. Issarny, “Developing Dependable Systems Using Software Architecture”. in *Proc. 1st Working IFIP Conference on Software Architecture* (*WICSA*), pp.83-103, KAP, P. Donohe ed., 1999.

[Shaw 1989] M. Shaw, “Larger Scale Systems Require Higher-Level Abstractions”, in *Proc. 5th International Workshop on Software Specification and Design*, Appeared in *ACM SIGSOFT Notes* 14(3), pp. 143-146, 1989.

[Shaw 2000] M. Shaw, “Sufficient Correctness and Homeostasis in Open Resource Coalitions”, in *Proc. 4th International Software Architecture Workshop (ISAW)*, pp. 46-50, 2000.

[Shaw & Garlan 1996] M. Shaw and D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Shaw *et al*. 1995] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young and G. Zelesnik, “Abstractions for Software Architecture and Tools to Support Them”, in *IEEE Transactions on Software Engineering* (*TSE*), 21(4), pp. 314-335, 1995.

[Schwanke *et al*. 1989] R.W. Schwanke, R.Z. Altucher and M.A. Platoff, “Discovering, Visualizing and Controlling Software Structure”, in *Proc. 5th International Workshop on Software Specification and Design*, Appeared in *ACM SIGSOFT Notes* 14(3), pp. 147-150, 1989.

[Sun 1998] Sun Microsystems, *Enterprise Java Beans Technology*, Technical Report, 1998, http://java.sun.com/products/ejb/

[Wile 1999] D. Wile, “AML: An Architecture Meta-Language”, in *Proc. IEEE International Conference on Automated Software Engineering* (*ASE*), 1999.

[Zarras 2000] A. Zarras, *Systematic Customization of Middleware, Thèse de Doctorat de l’Université de Rennes I*, English version available from http://www-rocq.inria.fr/solidor/doc/doc.html, 2000.

[Zarras & Issarny 2000] A. Zarras and V. Issarny, Assessing Software Reliability at the Architectural Level, in *Proc. 4th International Software Architecture Workshop (ISAW)*, 2000.

Chapter 2 – Mechanisms for Enforcing Dependability of Services

[Barrett *et al.* 1990] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues and N. A. Speirs, “The Delta-4 Extra Performance Architecture (XPA)”, in *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20),* (Newcastle upon Tyne, UK), pp.481-8, IEEE Computer Society Press, 1990.

[Birman 1985] K. P. Birman, “Replication and Fault-Tolerance in the ISIS System”, *ACM Operating Systems Review,* 19 (5), pp.79-86, 1985.

[Campbell & Randell 1986] R. H. Campbell and B. Randell, “Error Recovery in Asynchronous Systems”, *IEEE Transactions on Soft. Eng*., SE-12, 8, pp.811-26, 1986.

[Chandra & Toueg 1996] T. D. S. Chandra, Toueg, “Unreliable Failure Detectors for Reliable distributed Systems”, *Journal of the ACM,* Vol. 43 (2), pp.225-67, 1996.

[Chen & Avizienis 1978] L. Chen, A. Avizienis, “N-version Programming: A Fault Tolerant Approach to Reliability of Software Operation”, in *Proc. of the 8th IEEE Int. Symp. on Fault Tolerant Computing* (FTCS-8), IEEE Computer Society Press, pp.3-9, 1978.

[Chérèque *et al.* 1992] M. Chérèque, D. Powell, P. Reynier, J-L. Richier and J. Voiron, “Active Replication in Delta-4”, in *Proc. of the 22nd IEEE Int. Symp. on Fault Tolerant Computing* (*FTCS-22),* Boston (MA, USA), pp.28-37, 1992.

[Chiba 1995] S. Chiba, “A Metaobject Protocol for C++”, in *Proc. of OOPSLA’95* (ACM Conference on Object-Oriented Programming, Systems, Languages and Applications), Austin (TX-USA), October 1995, pp.285-99.

[Cristian 1995] F. Cristian, “Exception Handling and Tolerance of Software Faults,” In *Software Fault Tolerance* (ed. M. Lyu), Wiley, pp.81-107, 1995.

[Cristian & Fetzer 1998] F. Cristian and C. Fetzer, “The Timed Asynchronous System Model”, in *28th Int. Symp. on Fault-Tolerant Computing (FTCS-28),* (Munich, Germany), pp.140-9, IEEE Computer Society Press, 1998.

[Davies 1979] C. T. Davies, “Data Processing Integrity”, in *Computing System Reliability,* T. Anderson and B. Randell (Eds.), Cambridge University Press. 1979.

[Deswarte *et al.* 1991] Y. Deswarte, L. Blain, J.C. Fabre, “Intrusion Tolerance in Distributed Computing Systems”, in *Proc. of the 1991 IEEE Symp. on Research in Security and Privacy*, Oakland, May 1991, pp. 110-121.

[Dolev *et al.* 1997] D. Dolev, “On the Minimal Synchrony Needed for Distributed Consensus”, *Journal of the ACM,* Vol. 43(2), Jan., pp.77-97, 1997.

[Dumant *et al.* 1998] B. Dumant, F. Horn, F. Dang Tran and J.-B. Stefani, “Jonathan: an Open Distributed Processing environment in Java”, in *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing,* Sept., pp.175-90, 1998.

[Elmagarmid 1993] A. K. Elmagarmid (ed.), “Database Transaction Models for Advanced Applications”, *Morgan Kaufmann Publ.*, 1993.

[Fabre & Perennou 1998] J. C. Fabre, T. Pérennou, “A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach”, *IEEE Transactions on Computers*, Special Issue on Dependability of Computing Systems, Jan., pp.78-95, 1998.

[Felber 1998] P. Felber, *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*, PhD thesis, EPFL, Switzerland, 1998.

[Fisher *et al.* 1985] M. Fisher, N. Lynch, M. Paterson, “Impossibility of Distributed Consensus with One Faulty Process”, *Journal of the ACM*, Vol. 32 (2), April, pp.374-82, 1988.

[Garbinato *et al.* 1995] B. Garbinato, R. Guerraoui and K. Mazouni, “Implementation of the GARF Replicated Objects Platform”, *Distributed Systems Engineering Journal*, 2, March pp.14-27, 1995.

[Garcia-Molina & Salem 1987] H. Garcia-Molina and K. Salem, “SAGAS”, in *Proc. of the SIGMod 1987 Annual Conference*, Dayal, U.; Traiger, I. (Eds.), San Francisco, CA, May 1987, ACM, ACM Press, pp.249–59.

[Gray & Reuter 1993] J. Gray and A. Reuter, “Transaction Processing: Concepts and Techniques”, *Morgan Kaufmann Publishers*, San Mateo, California, 1993.

[Guerraoui & Shiper 1997] R. Guerraoui and A.Shiper, “Consensus: the Big Misunderstanding”, in *Proc. of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems* (*FTDCS’97*), Tunis (Tunisia), Oct., pp.183-8, 1997.

[Hagen & Alonco 1998] C. Hagen and G. Alonco, “Flexible Exception Handling in the OPERA Process Support System”, in *Proc. 18th Int. Conf. on Distributed Computing Systems*, Amsterdam, The Netherlands. 1998.

[Hoare 1976] C. A. R. Hoare, “Parallel Programming: an Axiomatic Approach”, in *Languages, Hierarchies and Interfaces* (G. Goos and J. Hartmaur, Eds.), *LNCS-46*. Spinger-Verlag, Berlin. 1976.

[Horning & Randell 1973] J. J. Horning and B. Randell, “Process Structuring”, in *Comp. Surveys*, 5, pp.5-30. 1973.

[Hsu 1993] M. Hsu, “Special Issue on Workflow and Extended Transaction Systems”, *Data Engineering*, 16, 2, 1993.

[Killijian & Fabre 1998] M. O. Killijian, J. C. Fabre, J. C. Ruiz-Garcia, S. Chiba, “A Metaobject Protocol for Fault Tolerant CORBA Applications”, IEEE *Symposium on Reliable Distributed Systems* (SRDS’98), West Lafayette, October 98, pp.127-34.

[Killijian & Fabre 2000] M. O. Killijian and J. C. Fabre, “Implementing a Reflective Fault-Tolerant CORBA System”, in *Proc. of 19th IEEE Symposium on Reliable Distributed Systems* (SRDS2000), Nurnberg, Germany, Oct. 2000.

[Landis & Maffeis 1997] S. Landis and S. Maffeis, “Building Reliable Distributed Systems with Corba”, in *Theory and Practice of Object Systems*, (special issue on the future of Corba 3), 3 (1), pp.59-66, 1997.

[Laprie *et al.* 1995] J. C. Laprie, J. Arlat, C. Beounes, K. Kanoun, “Definition And Analysis Of Hardware -And-Software Fault-Tolerant Architectures”, *Advances In Ultra-Dependable Distributed Systems*, Eds. N. Suri, C. J. Walter, M. M. Hugue, IEEE Computer Society Press, pp.42-54, 1995.

[Leymann & Roller 1999] F. Leymann and D. Roller, “Production Workflow: Concepts and Techniques”, *Prentice Hall*. 1999, 479 p.

[Liskov 1988] B. Liskov, “Distributed Programming in Argus,” *Communications of the ACM*, 31, 3, pp.300-12, 1988.

[Maffeis & Schmidt 1997] S. Maffeis and D. C. Schmidt, “Constructing Reliable Distributed Communication Systems with Corba”, in *IEEE Communications Magazine*, Vol. 14(2), Feb. 1997, 6p.

[Moser & Melliar-Smith 1997] L. E. Moser and P. M. Melliar-Smith, “The Interception Approach to Reliable Distributed CORBA Objects”, P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Panel on Reliable Distributed Objects, in *3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, (Or, USA), June, pp 245-8, 1997.

[Moss 1981] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. Thesis (Tech. Report 260), MIT Lab. for Computer Science, Cambridge, MA, 1981.

[Natarajan *et al.* 2000] B. Natarajan, A. Gokhale, D. Schmidt, and S. Yajnik, “DOORS: Towards high-performance fault-tolerant CORBA”, in *Proc. of the 2nd International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, September 2000.

[Nett & Mock 1995] E. Nett and M. Mock, “How to Commit Concurrent, Non-Isolated Computations”, in *Proc. of the 5th IEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, August 1995.

[Parrington *et al.* 1995] G. D. Parrington, S. K. Shrivastava, S. M. Wheater and M. C. Little, “The Design and Implementation of Arjuna”, in *USENIX Computing Systems Journal*, 8, Summer, pp.255 – 308, 1995.

[Powell 1991] D. Powell (Ed.), “Delta-4: A Generic Architecture for Dependable Distributed Computing”, *Research Reports ESPRIT*, Springer-Verlag, Berlin, Germany, 1991.

[Powell 1991] D. Powell (Ed.), *Delta-4: a Generic Architecture for Dependable Distributed Computing*, Research Reports ESPRIT, 484p., Springer-Verlag, Berlin, Germany, 1991.

[Powell 1992] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, in *22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, (Boston, MA, USA), pp.386-95, IEEE Computer Society Press, 1992.

[Randell 1975] B. Randell, “System Structuring for Software Fault Tolerance”, in *IEEE Trans. on Software Engineering*, SE-1 (2), pp.220-32, 1975.

[Roman *et al.* 1999] M. Roman, F. Kon and R. H. Campbell, “Supporting Dynamic Reconfiguration in the dynamicTAO Reflective ORB”, *Technical Report UIUCDCS-R-99-2085*, University of Illinois at Urbana-Champaign, March 1999.

[Romanovsky *et al.* 1998] A. Romanovsky, J. Xu and B. Randell, "Exception Handling in Object-Oriented Real-Time Distributed Systems", in *Proc. the 1st International Symposium on Object-oriented Real-time Distributed Computing* (ISORC'98), Kyoto, Japan, pp.32-42, 1998.

[Rubira 1994] C. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation*, PhD thesis, University of Newcastle upon Tyne, Department of Computing Science, October 1994.

[Speirs & Barrett 1989] N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to Provide Dependable Computing", in *Proc. of the 19th IEEE Int. Symp. on Fault Tolerant Computing* (FTCS-19), CSPress, pp.184-90, 1989.

[Tatsubori 1999] M. Tatsubori, "An Extension Mechanism for the Java Language", *Master Thesis*, University of Tsukuba, Tsukuba, Ibaraki, Japan, 1999.

[Veríssimo & Almeida 1995] P. Veríssimo and C. Almeida, "Quasi-Synchronism: a Step Away from the Traditional Fault-Tolerant Real-Time System Models", *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7 (4), pp.35-9, 1995

[Wheater *et al.* 2000] S. M. Wheater, S. K. Shrivastava and F. Ranno, "OPENflow: A CORBA Based Transactional Workflow System", In *Advances in Distributed Systems*, S. Krakowiak, S. Shrivastava (Eds). LNCS-1752, pp.354-74, 2000.

[Xu *et al.* 1995] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery" in *Proc. the 25th International Symposium on Fault Tolerant Computing* (FTCS-25), Pasadena, California, pp.499 – 509, 1995.

[Xu *et al.*1999] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. . Zorzo, E. Canver and F. von Henke. "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions", in *Proc. the 29th International Symposium on Fault Tolerant Computing* (FTCS-29), Madison, USA, IEEE CS, pp.68-75, 1999.

[Zorzo *et al.* 1999] A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. J. Stroud and I. S. Welch. "Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study", *Software: Practice & Experience*, 29, 7, pp.1-21, 1999.

Chapter 3 – Wrapping Technology

[Carreira *et al.* 1998] J. Carreira, H. Madeira and J. G. Silva, Xception: A Technique for the Experimental *Evaluation of Dependability in Modern Computers, IEEE Trans. on Software Engineering*, SE-24, pp.125-36, 1998.

[Cheswick & Bellovin 1994] W. R. Cheswick and S. M. Bellovin, *Firewalls and Internet Security*, Addison-Wesley, 1994, 306 p.

[ConceptualModel 2000] *The DSoS Conceptual Model*. Deliverable BC1, European IST DSoS project (IST-1999-11585), September 2000.

[Chiba 2000] S. Chiba, "Load-time Structural Reflection in Java", in *ECOOP 2000 - European Conference on Object-Oriented Programming*. E. Bertino (Ed.), LNCS-1850, pp.313-37, 2000.

[de Oliveira Guimarães 1998] J. de Oliveira Guimarães, "Reflection for Statically Typed Languages", in *ECOOP'98 - Object-Oriented Programming*. E. Jul (Ed.), Springer-Verlag, Berlin Heidelberg, pp.440-61, 1998.

[Erlingsson & Schneider 1999] U. Erlingsson and F. Schneider. "SASI Enforcement of Security Policies: A Retrospective", *New Security Paradigms Workshop*, Caledon Hills, Canada, pp.87-95, 1999.

[Fabre *et al.* 1999] J. C. Fabre, F. Salles, M. Rodriguez-Moreno and J. Arlat, "Assessment of COTS Microkernels by Fault Injection", *IFIP Dependable Computing for Critical Applications* (DCCA'99), San José, pp.19-38, 1999.

[Fraser *et al.* 1999] T. Fraser, L. Badger and M. Feldman, "Hardening COTS Software with Generic Software Wrappers", *IEEE Symposium on Security and Privacy*, Oakland, CA, pp.2-16, 1999.

[Holzle 1993] U. Holzle, "Integrating Independently-Developed Components in Object-Oriented Languages", in *ECOOP'93 - Object-Oriented Programming*, O. M. Nierstrasz (Ed.). Springer-Verlag, Berlin Heidelberg pp.36-56, 1993.

[Hsueh *et al.* 1997] M. C. Hsueh, T. Tsai and R. Iyer, "Fault Injection Techniques and Tools", *Computer*, vol. 30, pp.75-82, April 1997.

[Kao *et al.* 1993] W. Kao, R. K. Iyer and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults", *IEEE Trans. on Software Engineering*, SE-19, pp.1105-18, 1993.

[Keller & Holze 1998] R. Keller and U. Holzle, "Binary Component Adaptation", in *ECOOP'98 - Object-Oriented Programming*, E. Jul (Ed.) Springer-Verlag, Berlin Heidelberg, pp.307-29, 1998.

[Kiczales *et al.* 1991] G. Kiczales, J. des Rivieres and D. G. Bobrow, *The Art of the Metaobject Protocol*. Massachusetts Institute of Technology, 1991.

[Kroop *et al.* 1998] N. P. Kropp, P. J. Koopman and D. P. Siewiorek, “Automated Robustness Testing of Off-The-Shelf Software Components”, in *Proc. of the 28th IEEE Symp. on Fault Tolerant Computing*, Munich, Germany, pp.230-39, 1998.

[Maes 1987] P. Maes, “Concepts and Experiments in Computational Reflection”, in *Proc. of OOPSLA’87*, ACM, pp. 147-155, October 1987.

[OMG 2000] *CORBA Security Services Specification*, v. 1.5, Object Management Group, Inc, 2000 (http://www.omg.org/technology/documents/formal/security_service.htm).

[Powell 1992] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, *Proc. FTCS-22* (Boston, MA, USA), pp.386-95, 1992.

[Rodriguez *et al.* 2000] M. Rodriguez, J. C. Fabre and J. Arlat, “Formal Specification for Building Robust Real-time Microkernels”, in *Proc. of the IEEE Real-Time Systems Symposium* (RTSS 2000), Orlando (FL, USA), Nov. 2000, (to appear).

[Salles *et al.* 1999] F. Salles, M.R. Moreno, J. C. Fabre and J. Arlat, “MetaKernel and Fault Containment Wrappers”, in *Proc. of the 29th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS’29)*, Madison, WI, USA, pp.22-9, 1999.

[Shelton *et al.* 2000] C. Shelton, P. Koopman and K. Devale, “Robustness Testing of the Microsoft WIN32 API”, in *Proc. of the IEEE Conference on Dependable Systems and Networks* (DSN 2000), New-York (NJ, USA), pp.261-70, 2000.

[Szyperski 1998] C. Szyperski, *Component Software - Beyond Object Oriented Programming.* Addison-Wesley, 1998.

[Voas & Miller 1997] J. Voas and K. Miller, “Interface Robustness for COTS-Based Systems”, Digest no. 97/013, Colloquium on COTS and Safety Critical, Systems Inst. of Electrical Eng., Computing and Control Division, pp. 7/1-7/12, 1997.

[Voas 1998] J. Voas, “Certifying Off-The-Shelf Software Components”, *Computer*, 31 (6), pp. 53-9, 1998.

[Welch & Stroud 2000] I. Welch and R. J. Stroud, “Kava - A Reflective Java Based on Bytecode Rewriting,” in *Reflection and Software Engineering*, LNCS-1826, W. Cazzola, R. J. Stroud, F. Tisato, Eds. Springer-Verlag, pp.157-69. 2000.

Chapter 4 – Validation Techniques

[Abdulla & Jonsson 2000] P. A. Abdulla and B. Jonsson, “Invited Tutorial: Verification of Infinite-State Systems and Parameterised Systems”, *in Proc.12th International Conference on Computer Aided Verification,* 2000.

[Arlat et al. 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, “Fault Injection for Dependability Validation — A Methodology and Some Applications”, IEEE Transactions on Software Engineering, 16 (2), pp.166-82, February 1990.

[Arlat *et al.* 1999] J. Arlat, J. Boué and Y. Crouzet, “Validation-based Development of Dependable Systems”, *IEEE Micro*, 19 (4), pp.66-79, July-August 1999.

[Arlat *et al.* 1993] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems”, *IEEE Transactions on Computers*, 42 (8), pp.913-23, August 1993.

[Avizienis & Rennels 1972] A. Avizienis and D. Rennels, “Fault-Tolerance Experiments with the JPL STAR Computer”, in *Proc. 6th Annual IEEE Computer Society Conference (COMPCON’72),* (San Francisco, CA, USA), pp.321-4, IEEE Computer Society Press, 1972.

[Avresky *et al.* 1996] D. Avresky, J. Arlat, J.-C. Laprie and Y. Crouzet, “Fault Injection for the Formal Testing of Fault Tolerance”, *IEEE Transactions on Reliability*, 45 (3), pp.443-55, 1996.

[Baier *et al.* 1997] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, “Symbolic Model-checking for Probabilistic Processes”, in *Proc. 24th ICALP*, LNCS 1256, pp 430-440, 1997.

[Behnia & Waeselynck 1999] S. Behnia, and H. Waeselynck, “Test Criteria Definition for B Models”, in *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM’99), Toulouse France,* Springer Verlag, LNCS 1708, Vol I, pp.509-29, 1999.

[Beizer 1990] B. Beizer, *Software Testing Techniques,* 2nd edition, Van Nostrand Reinhold, 1990.

[Berezin *et al*. 2000] S. Berezin, E. Clarke and S. Jha, W. Marrero, “Model Checking Algorithms for the mu-Calculus”, in *Proof, Logic and Interaction: Essays in Honour of Robin Milner*, C. Stirling, G. Plotkin and M. Tofte (eds), MIT Press, 2000.

[Bernot *et al.* 1991] G. Bernot, M.-C. Gaudel, and B. Marre, “Software Testing Based on Formal Specifications: a Theory and a Tool”, *Software Engineering Journal*, 6 (6), pp.387-405, 1991.

[Biere *et al.* 1999] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “Symbolic Model Checking using SAT procedures instead of BDDs”, *in Proc. Design Automation Conference (DAC’99),* 1999.

[Bozga *et al.* 1998] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis and S. Yovine, “Kronos: A Model-Checking Tool for Real-time Systems”, in *Proc. 10th Conference on Computer Aided Verification*, 1998.

[Brinksma 1989] E. Brinksma, “A Theory for the Derivation of Tests”, in *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project*, Elsevier Science Publishers North Holland, pp.235-47, 1989.

[Carreira *et al.* 1998] J. Carreira, H. Madeira and J. G. Silva, “Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers”, *IEEE Transactions on Software Engineering*, 24 (2), pp.125-36, February 1998.

[Carreira *et al.* 1999] J. V. Carreira, D. Costa and J. G. Silva, “Fault Injection Spot-checks Computer System Dependability”, *IEEE Spectrum*, 36, pp.50-5, August 1999.

[Cassez & Laroussine 2000] F. Cassez and F. Laroussine, “Model-Checking for Hybrid Systems by Quotienting and COnstraint Solving”, *in Proc. 12th International Conference on Computer Aided Verification*, pp 373-389, 2000.

[Chandy & Misra 1988] K. Mani Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison Wesley Publishing Company, 1988.

[Chow 1978] T. S. Chow, “Testing Software Design Modeled by Finite-State Machine”, *IEEE Transactions on Software Engineering*, 4 (3), 1978.

[Clarke *et al*. 1999] E. Clarke, O Grumberg and D. Peled, *Model-checking*, MIT Press 1999.

[Creese & Roscoe 2000] S. Creese and A. W. Roscoe, “Data Independent Induction over Structured Networks”, in *Proc. International Conference on Parallel Distributed Programming Techiques and Applications*, 2000.

[Cukier *et al.* 1999] M. Cukier, D. Powell and J. Arlat, “Coverage Estimation for Stratified Fault-Injection”, *IEEE Transactions on Computers*, 48 (7), pp.707-23, July 1999.

[DeLong *et al.* 1996] T. A. DeLong, B. W. Johnson and J. A. Profeta III, “A Fault Injection Technique for VHDL Behavioral-Level Models”, *IEEE Design and Test of Computers*, Winter, pp.24-33, 1996.

[De Nicola & Hennessy 1984] R. de Nicola, and M. Hennessy, “Testing Equivalences for Processes”, *Theoretical Computer Science*, 34, 1984.

[Dick & Faivre 1993] J. Dick, and A. Faivre, “Automating the Generation and Sequencing of Test Cases from Model-Based Specifications”, in *Proc. 1st Int. Symp. of Formal Methods Europe (FME-93),* Springer Verlag, LNCS 670, pp.268-84, 1993.

[Doong & Frankl 1994] R. K. Doong, and P. G. Frankl, “The ASTOOT Approach to Testing Object-Oriented Programs”, *ACM Trans. on Software Engineering and Methodology*, 3 (2), pp.101-30, 1994.

[Du Bousquet *et al*. 1999] L. du Bousquet, F. ouabdesselam, J.-L. Richier and N. Zuanon, “Lutess: a Specification-Driven Testing Environment for Synchronous Software”, in *Proc. 21st Int. Conf. on Software Engineering (ICSE’99)*, Los Angeles, USA, ACM Press, pp.267-76, May 1999.

[Emerson & Sistla 2000] E. A. Emerson and A. P. Sistla (Eds.), “Computer Aided Verification”, *Proc. of the 12th International Conference CAV 2000*, Springer, 2000.

[Fabre *et al.* 2000] J.-C. Fabre, M. Rodríguez, F. Salles, J. Arlat and J.-M. Sizun, “Building Dependable COTS Microkernel-based Systems using MAFALDA”, in *Proc. 2000 Pacific Rim International Symposium on Dependable Computing (PRDC-2000),* (Los Angeles, CA, USA), IEEE CS Press, 2000.

[Fabre *et al.* 1999] J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, “Assessment of COTS Microkernels by Fault Injection”, in *Dependable Computing for Critical Applications (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999)* (C. B. Weinstock and J. Rushby, Eds.), Dependable Computing and Fault-Tolerant Systems, 12, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.25-44, IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[Fink & Bishop 1997] G. Fink and M. Bishop, “Property-Based Testing: A New Approach to Testing for Assurance”, *ACM SIGSOFT Software Engineering Notes,* 22(4), pp.74-80, 1997.

[Formal System (Europe) Ltd 1996] Formal Systems (Europe) Ltd, *FDR User Manual*, 1996.

[Fujiwara *et al*. 1991] G. Fujiwara, S. von Bochmann, F. Khendek, and A. Ghedamsi, “Test Selection Based on Finite State Models”, *IEEE Transactions on Software Engineering*, 17 (6), 1991.

[Gaudel & James 1998] M.-C. Gaudel, and P. R. James**.** “Testing algebraic data types and processes: a unifying theory”**,** *Formal Aspects of Computing,* BCS, 10, pp.436-51, 1998.

[Gray 1986] J. Gray, “Why Do Computers Stop and What Can Be Done About It?”, in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems (SRDSDS-5),* (Los Angeles, CA, USA), pp.3-12, IEEE Computer Society Press, 1986.

[Gray 1993] J. Gray (Ed.), *The Benchmark Handbook,* Morgan Kaufmann Publishers, San Francisco, CA, USA, 1993.

[Harrold 2000] M. J. Harrold, “Testing: A Roadmap”, in Proc. *The Future of Software Engineering*, A. Finkelstein (Editor), ACM Press, pp.61-72, June 2000.

[Hoare 1978] C. A. R. Hoare, “Communicating Sequential Processes”, *Communications of the ACM*, 21, 8, pp 666-677, 1978.

[Hoare 1985] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1999.

[Hoare & He 1999] C. A. R. Hoare and J. He, *Unifying Theories of Programming,* Prentice-Hall, 1999.

[Holzmann 1991] G. Holzmann, *Design and Validation of Computer Protocols,* Prentice-Hall, 1991.

[Jackson 2000] D. Jackson, “Automating First-Order Relational Logic”, *in Proc. ACM SIGSOFT Conf. on Foundations of Software Engineering*, San Diego, November 2000.

[Jagadeesan *et al.* 1997] “Specification-Based Testing of Reactive Software: Tools and Experiments”, in *Proc. 19th Int. Conf. on Software Engineering (ICSE’97),* ACM Press, May 1997.

[Jenn *et al.* 1995] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), pp.329-46, Springer, Berlin, Germany, 1995.

[Kaâniche *et al.* 1998] M. Kaâniche, L. Romano, Z. Kalbarczyk, R. K. Iyer and R. Karcich, “A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Bsaed RAID Storage Architecture”, in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28),* (Munich, Germany), pp.6-15, IEEE Computer Society Press, 1998.

[Kanawati *et al.* 1995] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System”, *IEEE Transactions on Computers,* 44 (2), pp.248-60, February 1995.

[Kao & Iyer 1995] W.-L. Kao and R. K. Iyer, “DEFINE: A Distributed Fault Injection and Monitoring Environment”, in *Fault-Tolerant Parallel and Distributed Systems* (D. Pradhan and D. R. Avresky, Eds.), pp.252-9, IEEE CS Press, Los Alamitos, CA, USA, 1995.

[Kao *et al.* 1993] W.-L. Kao, R. K. Iyer and D. Tang, “FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults”, *IEEE Transactions on Software Engineering,* 19 (11), pp.1105-18, November 1993.

[Karlsson *et al.* 1998] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber and J. Reisinger, “Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture”, in *Dependable Computing for Critical Applications (Proc. 5th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-5, Urbana-Champaign, IL, USA, September 1995)* (R. K. Iyer, M. Morganti, W. K. Fuchs and V. Gligor, Eds.), Dependable Computing and Fault-Tolerant Systems, 10, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.267-87, IEEE Computer Society Press, Los Vaqueros, CA, USA, 1998.

[Koopman *et al.* 1997] P. J. Koopman, J. Sung, C. Dingman, D. P. Siewiorek and T. Marz, “Comparing Operating Systems using Robustness Benchmarks”, in *Proc. 16th Symp. on Reliable Distributed Systems (SRDS-16),* (Durham, NC, USA), pp.72-9, IEEE Computer Society Press, 1997.

[Kurshan & McMillan 1989] R. P. Kurshan and K. McMillan, “A Structural Induction Theorem for Prcoesses”, in *Proc. 8th Symposium on Principles of Distributed Computing*, 1989.

[Kwiatkowska 1999] M. Kwiatkowska, *Verification of Quality of Service Properties in Timed Systems: Case for Support*, School of Computer Science, University of Birmingham, 1999.

[Labovitz *et al.* 1999] C. Labovitz, A. Ahuja and F. Jahanian, “Experimental Study of Internet Stability and Backbone Failures”, in *Proc. 29th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-29),* (Madison, WI, USA), pp.278-85, IEEE Computer Society Press, 1999.

[Lazic & Nowak 2000] R. Lazic and D. Nowak, “A Unifying Approach to Data-independence”, *in Proc. 11th International Conference on Concurrency Theory (CONCUR 2000),* Springer-Verlag, 2000.

[Lee & Yannakakis 1994] D. Lee, and M. Yannakis, “Testing Finite-State Machines: State Identification and Verification”, *IEEE Transactions on Computers,* 43 (3), 1994.

[Madeira *et al.* 2000] H. Madeira, D. Costa and M. Vieira, “On the Emulation of Software Faults by Software Fault Injection”, in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.417-26, IEEE Computer Society Press, 2000.

[Marre & Arnould 2000] B. Marre, and A. Arnould, “Test Sequences Generation from LUSTRE Descriptions: GATEL”, in *Proc. 15th IEEE ACM Int. Conf. on Automated Software Engineering (ASE’00), Grenoble, France,* IEEE Comp. Soc. Press, Sept. 2000.

[Martínez *et al.* 1999] R. J. Martínez, P. J. Gil, G. Martín, C. Pérez and J. J. Serrano, “Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection”, in *Dependable Computing for Critical Applications (Proc. 7th IFIP Working Conf. on Dependable Computing for Critical Applications: DCCA-7, San Jose, CA, USA, January 1999)* (C. B. Weinstock and J. Rushby, Eds.), Dependable Computing and Fault-Tolerant Systems, 12, (A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), pp.249-65, IEEE Computer Society Press, San Jose, CA, USA, 1999.

[McMillan 1993] K. McMillan, *Symbolic Model-checking*, Kluwer, 1993.

[Mukherjee & Siewiorek 1997] A. Mukherjee and D. P. Siewiorek, “Measuring Software Dependability by Robustness Benchmarking”, *IEEE Transactions of Software Engineering*, 23 (6) 1997.

[Namjoshi & Trefler 2000] K. S. Namjoshi and R. J. Trefler. “On the Completeness of Compositional Reasoning”, in *Proc. 12th International Conference on Computer Aided Verification*, pp. 139-154, 2000.

[Pan 2000] J. Pan, “A Method to Evaluate CORBA ORB Robustness”, in *Suppl. Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp.A.31-A.3, IEEE Computer Society Press, 2000.

[Pitt & Freestone 1990] D. H. Pitt, and D. Freestone, “The Derivation of Conformance Tests from LOTOS Specifications”, *IEEE Transactions on Software Engineering*, 16 (12), 1990.

[Pnueli 1977] A. Pnueli, “The Temporal Logic of Programs”, in *FOCS*, 1977.

[Powell 1994] D. Powell, “Distributed Fault-Tolerance — Lessons from Delta-4”, *IEEE Micro*, 14 (1), pp.36-47, February 1994.

[Powell *et al.* 1995] D. Powell, E. Martins, J. Arlat and Y. Crouzet, “Estimators for Fault Tolerance Coverage Evaluation”, *IEEE Transactions on Computers,* 44 (2), pp.261-74, February 1995.

[Raymond *et al*. 1998] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs, “Automatic Testing of Reactive Systems”, in *Proc. 19th IEEE Real Time Systems Symp,* IEEE, 1998.

[Rennels & Avizienis 1973] D. A. Rennels and A. Avizienis, “RMS: A Reliability Modeling System for Self-Repairing Computers”, in *Proc. 3rd Int. Symp. on Fault-Tolerant Computing (FTCS-3),* (Palo Alto, CA, USA), pp.131-5, IEEE Computer Society Press, 1973.

[Rodríguez *et al.* 1999] M. Rodríguez, F. Salles, J.-C. Fabre and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, in *Proc. 3rd European Dependable Computing*

Conf. (EDCC-3), (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-60, Springer, 1999.

[Roscoe 1998] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall International, 1998.

[Roscoe *et al.* 95] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson and J. B. Scattergood, "Hierarchical Compression for Model-checking CSP or How to Check 10^20 Dining Philosophers for Deadlock", in *Proc. 1st TACAS*, BRICS Notes Series NS-95-2, Department of Computer Science, University of Aarhus, 1995.

[Shankar 2000] N. Shankar, "Combining Theorem Proving and Model Checking through Symbolic Analysis", *in Proc. 11th International Conference on Concurrency Theory (CONCUR 2000)*, Springer-Verlag, 2000.

[Thévenod-Fosse & Waeselynck 1993] P. Thévenod-Fosse and H. Waeselynck, "STATEMATE Applied to Statistical Software Testing", in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'93)*, Cambridge, Massachusetts, USA, ACM Press, pp.99-109, June 1993.

[Thévenod-Fosse *et al.* 1994] P. Thévenod-Fosse, C. Mazuet and Y. Crouzet, "On Statistical Testing of Synchronous Data Flow Programs", in *Proc. 1st European Dependable Computing Conf. (EDCC-1)*, (K. Echtle, D. Hammer and D. Powell, Eds.), (Berlin, Germany), Lecture Notes in Computer Science, 852, pp.250-67, Springer-Verlag, 1994.

[Thévenod-Fosse *et al.* 1995] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software Statistical Testing", in *Predictably Dependable Computing Systems* (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), pp.253-72, Springer, Berlin, Germany, 1995.

[Tsai & Singh 2000] T. Tsai and N. Singh, "Reliability Testing of Applications on Windows NT", in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2000)*, (New York, NY, USA), pp.427-36, IEEE Computer Society Press, 2000.

[Tracey *et al.* 1998] N. Tracey, J. Clark and K. Mander, "Automated Program Flaw Finding using Simulated Annealing", in *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'98)*, Clearwater Beach, Florida, USA, ACM Press, pp.73-81, March 1998.

[Tretmans 1992] J. Tretmans, *A Formal Approach to Conformance Testing*, PhD thesis, University of Twente, the Netherlands, Dec. 1992.

[Van Aertryck *et al.* 1997] L. Van Aertryck, M. Benveniste, and D. Le Metayer, "Casting: a Formally Based Software Test Generation", in *Proc. IEEE Int. Conf. on Formal Engineering Methods*, pp.101-11, 1997.

[Voas & McGraw 1998] J. M. Voas and G. McGraw, *Software Fault Injection*, 353p., Wiley Computer Publishing, New York, 1998.

[Waeselynck & Thévenod-Fosse 1999] H. Waeselynck and P. Thévenod-Fosse, "A Case Study in Statistical Testing of Reusable Concurrent Objects", in *Proc. 3rd European Dependable Computing Conference (EDCC-3)*, Prague, Czech Republic, Springer, LNCS 1667, pp.401-18, Sept. 1999.

[Wolper & Boigelot 1998] P. Wolper and B. Boigelot, “Verifying Systems with Infinite but Regular State Spaces”, *in Proc. 10th International Conference on Computer Aided Verification*, pp. 81-97, 1998.

[Yount & Siewiorek 1996] C. R. Yount and D. P. Siewiorek, “A Methodology for the Rapid Injection of Transient Hardware Errors”, *IEEE Transactions on Computers*, 45 (8), pp.881-91, August 1996.

[Zakiuddin 1999] I. Zakiuddin, “Current Limits for Exploiting Automated Verification”, in *Proc. International Conference on Parallel Distributed Programming Techiques and Applications*, 1999.

Chapter 5 – Dependability Evaluation of Large Systems

[Balakrishnam & Trivedi 1995] M. Balakrishnam and K. S. Trivedi, “Component-wise Decomposition for an Efficient Reliability Computation of Systems with Repairable Components”, in *25th International Symp. on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.259-68, IEEE Computer Society Press, 1995.

[Balbo *et al.* 1988] G. Balbo, S. C. Bruell and S. Ghanta, “Combining Queuing Networks and GSPNs for the Solution of Complex Models of System Behaviour”, *IEEE Trans. on Computers,* 37, pp.1251-68, 1988.

[Berson *et al.* 1991] S. Berson, E. de Souza e Silva and R. R. Muntz, “A Methodology for the Specification and Generation of Markov Models”, in *Numerical Solutions for Markov Chains* (W. Stewart, Ed.), pp.11-36, Maecel Dekker, 1991.

[Bobbio & Trivedi 1986] A. Bobbio and K. S. Trivedi, “An Aggregation Technique for the Transient Analysis of Stiff Markov Chains”, *IEEE Trans. on Computers,* C-35 (9), pp.803-14, 1986.

[Bondavalli *et al.* 1999] A. Bondavalli, I. Mura and K. S. Trivedi, “Dependability Modelling and Sensitivity Analysis of Scheduled Maintenance Systems”, in *3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.7-23, Springer, 1999.

[Bouissou 1993] M. Bouissou, “The FIGARO Dependability Evaluation Workbench in Use: Case Studies for Fault- Tolerant Computer Systems”, in *23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23),* (Toulouse, France), pp.680-5, IEEE Computer Society Press, 1993.

[Buckley & Siewiorek 1995] M. F. Buckley and D. P. Siewiorek, “VAX/VMS Event Monitoring and Analysis”, in *25th International Symposium on Fault-Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.414-23, IEEE Computer Society, 1995.

[Buckley & Siewiorek 1996] M. F. Buckley and D. P. Siewiorek, “A Comparative Analysis of Event Typling Schemes”, in *26th International Symposium on Fault-Tolerant Computing (FTCS-26),* (Sendai, Japan), pp.294-303, IEEE Computer Society, 1996.

[Butner & Iyer 1980] S. E. Butner and R. K. Iyer, “A Statistical Study of Reliability and System Load at SLAC”, in *10th International Symposium on Fault Tolerant Computing (FTCS-10),* (Kyoto, Japan), IEEE Computer Society, 1980.

[Carrasco & Figueras 1986] J. A. Carrasco and J. Figueras, “METFAC: Design and Implementation of a Software Tool for Modeling and Evaluation of Complex Fault-Tolerant Computing Systems”, in *16th Int Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.424-9, IEEE Computer Society Press, 1986.

[Castillo & Siewiorek 1981] X. Castillo and D. P. Siewiorek, “Workload, Performance, and Reliability of Digital Computing Systems”, in *11th IEEE Int. Symp. Fault-Tolerant Computing (FTCS-11),* (Portland, Maine, USA), pp.279-85, 1981.

[Chillarege *et al.* 1995] R. Chillarege, S. Biyani and J. Rosenthal, “Measurement of Failure Rate in Widely Distributed Software”, in *25th IEEE International Symposium On Fault Tolerant Computing (FTCS-25),* (Pasadena, CA, USA), pp.424-33, IEEE Computer Society, 1995.

[Ciardo & Miner 1999] G. Ciardo and A. Miner, “A Data Structure for the Efficient Kroneker Solution of GSPNs”, in *8th Int. Workshop on Petri Nets and Performance Models*, (Zaragoza, Spain), pp.22-31, IEEE Computer Society Press, 1999.

[Ciardo & Miner 1996] G. Ciardo and A. S. Miner, “SMART: Simulation and Markovian Analyzer for Reliability and timing”, in *2nd IEEE Int. Computer Performance and Dependability Symp. (IPDS’96),* (Urbana-Champain, IL, USA), p.60, IEEE Computer Society Press, 1996.

[Ciardo & Trivedi 1993] G. Ciardo and K. S. Trivedi, “Decomposition Approach to Stochastic Reward Net Models”, *Performance Evaluation*, 18 (1), pp.37-59, 1993.

[Cramp *et al.* 1992] R. Cramp, M. A. Vouk and W. Jones, “An Operational Availability of a Large Software-Based Telecommunications System”, in *3rd Int. Symp. on Software Reliability Engineering,* (Research Triangle Park, NC, USA), pp.358-66, 1992.

[Fota *et al.* 1999a] N. Fota, M. Kâaniche and K. Kanoun, “Dependability Evaluation of an Air Traffic Control Computing System”, *Performance Evaluation*, 35 (3-4), pp.553-73, 1999a.

[Fota *et al.* 1999b] N. Fota, M. Kâaniche and K. Kanoun, “Incremental Approach for Building Stochastic Petri Nets for Dependability Modeling”, in *Statistical and Probabilistic Models in Reliability* (D. C. Ionescu and N. Limnios, Eds.), pp.321-35, Birkhäuser, 1999b.

[Goyal *et al.* 1986] A. Goyal, W. C. Carter, E. de Souza e Silva and S. S. Lavenberg, “The System Availability Estimator”, in *16th IEEE Int Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.84-9, 1986.

[Gray 1986] J. Gray, “Why Do Computers Stop and What Can be Done About it?”, in *5th Int. Symposium on Reliability in Distributed Software and Database Systems*, (Los Angeles, CA, USA), pp.3-12, 1986.

[Gray 1990] J. Gray, “A Census of Tandem System Availability Between 1985 and 1990”, *IEEE Transactions on Reliability*, R-39 (4), pp.409-18, 1990.

[Hansen & Siewiorek 1992] J. P. Hansen and D. P. Siewiorek, “Models of Time Coalescence in Event Logs”, in *22nd International Symposium on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.221-7, IEEE Computer Society, 1992.

[Harnedy 1998] S. Harnedy, *Total SNMP: Exploring the Simple Network Management Protocol,* Prentice Hall PTR, 1998.

[Iyer & Rossetti 1985] R. K. Iyer and D. J. Rossetti, “Effect of System Workload on Operating System Reliability: A Study on IBM 3081”, *IEEE Transactions on Software Engineering*, SE-11 (12), pp.1438-48, December 1985.

[Iyer & Tang 1996] R. K. Iyer and D. Tang, “Experimental Analysis of Computer System Dependability”, in *Fault-Tolerant Computer System Design* (D. K. Pradhan, Ed.), pp.282-392 (chapter 5), Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.

[Iyer & Velardi 1985] R. K. Iyer and P. Velardi, “Hardware-Related Software Errors: Measurement and Analysis”, *IEEE Transactions on Software Engineering*, SE-11 (2), pp.223-31, 1985.

[Iyer *et al.* 1986] R. K. Iyer, L. T. Young and V. Sridhar, “Recognition of Error Symptoms in Large Systems”, in *1986 IEEE/ACM Fall Joint Computer Conference,* (Dallas, TX, USA), pp.797-806, IEEE-ACM, 1986.

[Kaâniche *et al.* 1994] M. Kaâniche, K. Kanoun, M. Cukier and M. Bastos Martini, “Software Reliability Analysis of Three Successive Generations of a Switching System”, in *First European Conference on Dependable Computing (EDCC-1),* (D. H. K. Echtle, D. Powell, Ed.), (Berlin, Germany), Lecture Notes in Computer Science, 852, pp.473-90, Springer-Verlag, 1994.

[Kaâniche *et al.* 1990] M. Kaâniche, K. Kanoun and S. Metge, “Failure Analysis and Validation of a Telecommunication Equipment Software System”, *Annales des Telecommunications*, 45 (11-12), pp.657-70, 1990.

[Kalyanakrishnan *et al.* 1999a] M. Kalyanakrishnan, R. K. Iyer and J. U. Patel, “Reliability of Internet Hosts: a Case Study from the End User’s Perspective”, *Computer Networks*, 31, pp.47-57, 1999.

[Kalyanakrishnan *et al.* 1999b] M. Kalyanakrishnan, Z. Kalbarczyk and R. K. Iyer, “Failure data Analysis of a LAN of Windows NT Based Computers”, in *18th IEEE Symposium on Reliable Distributed Systems (SRDS-18),* (Lausanne, Switzerland), pp.178-87, IEEE Computer Society, 1999.

[Kanoun & Borrel 1996] K. Kanoun and M. Borrel, “Dependability of Fault-tolerant Systems — Explicit Modeling of the Interactions Between Hardware and Software Components”, in *2nd IEEE Int. Computer Performance and Dependability Symposium (IPDS),* (Urbana-Champaign, IL, USA), pp.252-61, 1996.

[Kanoun *et al.* 1999] K. Kanoun, M. Borrel, T. Moreteveille and A. Peytavin, “Modeling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System”, *IEEE Transactions on Computers*, 48 (5), pp.528-35, 1999.

[Kanoun *et al.* 1997] K. Kanoun, M. Kaâniche and J.-C. Laprie, “Qualitative and Quantitative Reliability Assessment”, *IEEE Software*, 14 (2), pp.77-86, mars 1997.

[Kanoun & Laprie 1996] K. Kanoun and J.-C. Laprie, “Trend Analysis”, in *Handbook of Software Reliability Engineering* (M. Lyu, Ed.), pp.401-37 (Chapter 10), McGraw Hill, 1996.

[Kanoun & Sabourin 1987] K. Kanoun and T. Sabourin, “Software Dependability of a Telephone Switching System”, in *17th IEEE Int Symp. on Fault-Tolerant Computing (FTCS-17),* (Pittsburgh, PA, USA), pp.236-41, IEEE Computer Society Press, 1987.

[Kendall 1977] M. G. Kendall, *The Advanced Theory of Statistics,* Oxford University Press, 1977.

[Kenney & Vouk 1992] G. Q. Kenney and M. A. Vouk, “Measuring the Field Quality of Wide-Distribution Commercial Software”, in *3rd IEEE Int. Symp. on Software Reliability Engineering (ISSRE’92)*, (Raleigh, NC, USA), pp.351-7, IEEE Computer Society Press, 1992.

[Labovitz *et al.* 1999] C. Labovitz, A. Ahuja and F. Jahanian, “Experimental Study of Internet Stability and Backbone Failures”, in *29th International Symposium on Fault Tolerant Computing (FTCS-29)*, (Madison, Wisconsin, USA), pp.278-85, IEEE Computer Society, 1999.

[Lee *et al.* 1991] I. Lee, R. K. Iyer and D. Tang, “Error/Failure Analysis Using Event Logs from Fault Tolerant Systems”, in *21st International Symposium on Fault Tolerant Computing (FTCS-21)*, (Montreal, Canada), pp.10-7, IEEE Computer Society, 1991.

[Levendel 1990] Y. Levendel, “Reliability Analysis of Large Software Systems: Defects Data Modeling”, *IEEE Transactions on Software Engineering*, SE-16 (2), pp.141-52, February 1990.

[Long *et al.* 1995] D. Long, A. Muir and R. Golding, “A Longitudinal Survey of Internet Host Reliability”, in *14th Symposium on Reliable Distributed Systems (SRDS-95)*, (Bad Neuenahr, Germany), pp.2-9, 1995.

[Lyu 1995] M. R. Lyu (Ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1995.

[Matthews & Cottrell 2000] W. Matthews and L. Cottrell, “The PingER Project: Active Internet Performance Monitoring for the HENP Community”, *IEEE Communications Magazine*, pp.130-6, May 2000.

[Maxion & Feather 1990] R. A. Maxion and F. E. Feather, “A Case Study of Ethernet Anomalies in a Distributed Computing Environment”, *IEEE Transactions on Reliability*, 39 (4), pp.433-43, 1990.

[McConnel *et al.* 1979] S. R. McConnel, D. P. Siewiorek and M. M. Tsao, “The Measurement and Analysis of Transient Errors in Digital Computer Systems”, in *9th International symposium on Fault-Tolerant Computing*, (Madison, Wisconsin,), pp.67-70, IEEE Computer Society, 1979.

[Meyer & Sanders 1993] J. F. Meyer and W. H. Sanders, “Specification and Construction of Performability Models”, in *Int. Workshop on Performability Modeling of Computer and Communication Systems*, (Mont Saint Michel, France), pp.1-32, 1993.

[Moran *et al.* 1990] P. Moran, P. Gaffney, J. Melody, M. Condon and M. Hayden, “System Availability Monitoring”, *IEEE Transactions on Reliability*, R-39 (4), pp.480-5, 1990.

[Muppala *et al.* 1992] J. K. Muppala, A. Sathaye, R. Howe, C and K. S. Trivedi, “Dependability Modeling of a Heterogeneous VAX-cluster System Using Stochastic Reward Nets”, in *Hardware and Software Fault Tolerance in Parallel Computing Systems* (D. R. Avresky, Ed.), pp.33-59, 1992.

[Musa *et al.* 1987] J. Musa, A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application,* Computer Science Series, 621p., McGraw-Hill, New-York, 1987.

[Orfali *et al.* 1996] R. Orfali, D. Harkey and J. Edwards, *The Essential Client-Server Survival Guide (Second Edition),* John Wiley & Sons, Inc., 1996.

[Paxson *et al.* 1998] V. Paxson, J. Mahdavi, A. Adams and M. Mathis, “An Architecture for Large-Scale Internet Measurement”, *IEEE Communications Magazine* (August), pp.48-54, 1998.

[Pérez-Jiménez & Compos 1999] C. J. Pérez-Jiménez and J. Compos, “On State Space Decomposition for the Numerical Analysis of Stochastic Petri Nets”, in *8th Int. Worshop on Petri Nets and Performance Models*, (Zaragoza, Spain), pp.32-41, IEEE Computer Society Press, 1999.

[Rabah & Kanoun 1999] M. Rabah and K. Kanoun, “Dependability Evaluation of a Distributed Shared Memory Multiprocessor System”, in *3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, J. Hlavicka and E. Maehle, Eds.), (Prague, Czech Republic), pp.42-59, Springer, 1999.

[Reibman & Veeraraghavan 1991] A. Reibman and M. Veeraraghavan, “Reliability Modeling: An Overview for System Designers”, *IEEE Computer*, April, pp.49-57, 1991.

[Rojas 1996] I. Rojas, “Compositional Construction of SWN Models”, *The Computer Journal*, 38 (7), pp.612-21, 1996.

[Siewiorek *et al.* 1978] D. P. Siewiorek, V. kini, H. Mashburn, S. R. McConnel and M. M. Tsao, “A Case Study of C.mmp, Cm*, and C.vmp: Part I—Experience with Fault Tolerance in Multiprocessor Systems”, *Proceedings of the IEEE,* 66 (10), pp.1178-99, 1978.

[Sriram 1993] K. B. Sriram, *A Study of the Reliability of Hosts on the Internet,* Master Thesis, University of California Santa Cruz, 1993.

[Sullivan & Chillarege 1992] M. Sullivan and R. Chillarege, “A Comparison of Software Defects in Database Management Systems and Operating Systems”, in *22nd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.475-84, 1992.

[Tang *et al.* 1990] D. Tang, R. K. Iyer and S. Subramani, “Failure Analysis and Modeling of a VAXcluster System”, in *20th International Symposium on Fault Tolerant Computing (FTCS-20),* (Newcastle Upon Tyne, UK), pp.244-51, IEEE Computer Society, 1990.

[Thakur & Iyer 1996] A. Thakur and R. K. Iyer, “Analyze-NOW — An Environment for Collection & Analysis of Failures in a Network of Workstations”, *IEEE Transactions on Reliability,* 45 (4), pp.561-70, 1996.

[Trivedi *et al.* 1994] K. S. Trivedi, B. R. Haverkort, A. Rindos and V. Mainkar, “Techniques and Tools for Reliability and Performance Evaluation: Problems and Perspectives”, in *7th International Conference on Modeling techniques and Tools for Computer Performance evaluation,* (L. N. i. C. Sciences, Ed.), pp.1-24, Springer, 1994.

[Tsao & Siewiorek 1983] M. Tsao and D. P. Siewiorek, “Trend Analysis on System Error Files”, in *13th International Symposium on Fault-Tolerant Computing (FTCS-13),* (Milano, Italy), pp.116-9, IEEE Computer Society, 1983.

[Wein & Sathaye 1990] A. S. Wein and A. Sathaye, “Validating Computer System Availability Models”, *IEEE Transactions on Reliability*, 39 (4), pp.468-79, 1990.

[Wood 1995] A. Wood, “Predicting Client/Server Availability”, *Computer* (April), pp.41-8, 1995.