DSoS

IST-1999-11585

Dependable Systems of Systems

Architecture and Design

Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs

Report : Deliverable IC2

Report Delivery Date: 30 September 2001

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)

List of Authors

Jean Arlat..... LAAS-CNRS, Toulouse, F

Jean-Charles Fabre ..... LAAS-CNRS, Toulouse, F

Valérie Issarny..... INRIA, Rocquencourt, F

Christos Kloukinas .....INRIA, Rocquencourt, F

Viet Khoi Nguyen .....INRIA, Rocquencourt, F

Manuel Rodriguez..... LAAS-CNRS, Toulouse, F

Alexander Romanovsky .....University of Newcastle upon Tyne, UK

Apostolos Zarras .....INRIA, Rocquencourt, F

Table of Contents

Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs

Jean Arlat[1], Jean-Charles Fabre[1], Valérie Issarny[2], Christos Kloukinas[2], Viet Khoi Nguyen[2], Manuel Rodriguez[1], Alexander Romanovsky[3], Apostolos Zarras[2]

[1]LAAS-CNRS (Toulouse, F), [2]INRIA (Rocquencourt, F),
[3]University of Newcastle upon Tyne (UK)

Abstract

This deliverable presents the latest project results in the area of architecture and design of DSoSs. Activities in this area relate to the three following topics: (i) architecture-based development of DSoSs, (ii) mechanisms for enforcing dependability of SoSs, and (iii) wrapping technology for adapting and protecting component systems composing a DSoS. Results in the area of DSoS architecture-based development include an environment whose core component is an extensible, UML-based ADL; this ADL may be specialised for describing DSoS-specific architectural styles, and for assisting the design, analysis and implementation of DSoSs from their architectural description. Dependability mechanisms that are presented are aimed at application-specific fault tolerance, and include a solution to component-level error detection and exception handling, to enable the actual integration of component systems. Furthermore, an extension of the traditional CA Action scheme, which eases the integration of autonomous component systems by not enforcing strong synchronisation among action participants, is sketched. The main objective of developing wrapping technologies in the DSoS context relates to the need to build error confinement areas. The project's work in this area has been on the definition of a generic wrapping framework and on its specialisation using formal description techniques for generating and implementing wrappers.

Chapter 1 - Introduction

Work in the AD Workpackage on "Architecture and Design" relates to the three following complementary areas:

- **Architecture-based development of complex distributed software systems**: our objective here is to address both provisioning of an architecture-based development environment assisting the design, analysis and implementation of DSoSs, and the design of novel system architectures to tackle the specific requirements of DSoSs such as enhanced dependability and integration of legacy component systems.
- **Mechanisms for enforcing system dependability**: our objective here is to devise novel dependability mechanisms, both at the application- and middleware-level, that account for the characteristics of the DSoS building blocks, including in particular legacy component systems that are autonomous.
- **Wrapping technology:** our objective here is to enable the use of wrapping technology as a means of integrating legacy component systems and of protection, which requires provisioning rigorous solutions to the generation of wrappers from the specification of the expected behaviour of component systems or from a detailed analysis of its failure modes.

This deliverable introduces results achieved in the AD WP over the last 18 months in the three above areas, following the state of the art survey presented in the BC2 deliverable and past experience of the contributing partners.

Chapter 2 focuses on the architecture-based environment for assisting the development of DSoSs. The core part of the environment is an extensible Architecture Description Language (ADL) that is based on UML. Various specialisations of the ADL may be integrated into the environment, which may thus benefit from results of the software architecture field (e.g., assisting thorough system analysis through the coupling of ADLs with formal methods and tools). Another advantage of the ability to specialise the environment is that it offers development assistance specifically aimed at DSoSs (i.e., DSoS architecting and validation, as addressed in the AD and VA workpackages). Our work regarding specialisation of the ADL has so far been oriented towards the mechanical analysis of the DSoS quality from both a qualitative and a quantitative point of view, using previously existing tools, without requiring extensive expertise in formal methods from developers. Every ADL specialisation follows the same pattern: it amounts to enabling the specification of the required quality attributes within the architectural elements. The resulting ADL description of a DSoS is then translated into a formal model that can be analysed by existing tools. As a result, developers are not required to master various formal methods to be able to analyse their systems. Instead, they describe the architectures of their systems using languages with which they are familiar. Three ADL specialisations are presented, which respectively enable model checking, performance analysis and reliability analysis of DSoSs. The specifics of DSoSs require that novel methods and tools for validating their quality be devised, as investigated in the VA WP. However, as a first step, we rely on existing validation methods and tools, not specifically aimed at DSoSs, as they already contribute to the assessment of DSoS quality and enable the release of a first prototype of our environment at an early stage. It is part of the project's future work to enrich the environment with DSoS-specific validation solutions that will be offered in

the course of the project, which is supported given the extensible nature of the development environment.

Chapter 3 concentrates on the provision of application-specific fault tolerance mechanisms to enhance the dependability of SoSs. It discusses initial results of the project on developing such mechanisms to be applied at the application level during the integration of the complex systems composing a DSoS. In this context, it is mandatory for the component systems to behave as expected, either normally or exceptionally, when integrated. This calls for a disciplined and systematic way of introducing component-level error detection and exception handling, based on the wrapping technology. A base solution to this issue is sketched in the chapter and its elaboration is part of the project's future work. The core fault tolerant mechanism lies in the extension of the Coordinated Atomic (CA) Action scheme, which is based on exception handling. The proposed solution does not impose tight synchronisation on action entry and exit, as imposed by conventional atomic action schemes, and hence is more suitable for the integration of autonomous component systems. In the light of the above, the chapter concludes with the project's initial view on developing a general architecture to be used in developing structured fault-tolerant SoSs.

Chapter 4 is specifically dedicated to the definition of wrapping technologies aimed at the development of DSoSs. In the DSoS context, the main objective of developing wrapping technologies relates to the need to build error confinement areas. Indeed, when errors cannot be recovered at the level of the faulted component system, wrapping mechanisms are needed to contain the error and to report them to the outside world. Such an early and local error processing is mandatory to be able to use simple but efficient error recovery strategies at the level of the SoS. The project's work in this area has so far been on the definition of a generic wrapping framework and on its specialisation using formal description techniques for generating and implementing wrappers. The basic framework relies on modelling system requirements to derive expected properties. The properties of a given component system are described preferably in some formal syntax that gathers both behavioural and temporal aspects. For instance, temporal logic can be used to define these properties and wrappers can be automatically produced by compiling the formal specifications. Resulting wrappers can thus account both for timing and functional constraints. The error detection relies on the runtime verification of the properties by executing the wrappers on-line. Some initial experimental results obtained with a real-time application running on a COTS real-time microkernel illustrate the benefits of the approach in terms of error detection coverage in both time and value domains. Finally, several variants of the proposed framework are defined, showing the various possibilities of specialising it in the context of DSoS.

Chapter 5 concludes the deliverable with a summary of our contribution, together with an overview of planned future work on the architecture and design of DSoSs.

Chapter 2 – An Architecture-based Environment for the Development of DSoSs

2.1 Introduction

As addressed in Chapter 1 of the DSoS State of the Art Survey [BC2], architecture-based development is a convenient approach for the design, analysis and implementation of DSoSs. The abstraction of architectural elements enforced by architecture-based development naturally fits with the assembly of DSoS component systems. We are therefore designing and implementing an architecture-based environment to support the development of DSoSs. Hence, our primary objective is to offer an ADL and associated methods and tools that support the thorough development of DSoSs. In particular, this leads us to define an ADL that is based on existing formal methods for DSoS analysis, as illustrated by existing ADLs (e.g., see [BC2]). Another objective for our work is to provide the DSoS developers with an environment that offers adequate tool support facilitating the specification of models suitable for DSoS analysis, without requiring extensive knowledge of formal modelling techniques (e.g., process algebra, Markov chains, Petri nets, queuing nets). In other words, we aim at offering a *developer-oriented, architecture-based environment*.

The specifics of DSoSs call for novel solutions regarding: (i) system architecting and associated dependability mechanisms for improving the SoS dependability as investigated in the AD workpackage, and (ii) methods and tools for validating the quality of DSoSs, as examined in the VA workpackage. Preliminary results for the above issues have already been offered by the DSoS project - e.g., see the following chapters of this report and the DSoS Preliminary Dependability Modelling Framework [DMS1]. However, as a first step, this chapter considers the exploitation of previously existing formal modelling methods, and in particular of associated tools, for the architecture-based design and quality analysis of DSoSs, showing the approach that we undertake to ease the task of developers. As such, the environment presented in this chapter should be viewed as assisting the development of SoSs, while it is part of the project's future work to integrate the latest DSoS results into the presented environment.

A DSoS consists of the integration of numerous component systems, possibly distributed, of different qualities and having different functional behaviours. Quality analysis is thus required during the overall development process of a DSoS. DSoS quality is characterised by a number of quality attributes (e.g., security, performance, reliability, availability, etc.). Typically, the value of the quality attributes is improved through the use of certain means (e.g., encryption/decryption, load balancing, fault tolerance mechanisms). Accordingly, two different kinds of analysis are considered here:

- *Qualitative analysis*, which aims to facilitate and verify the correct use of certain means for improving the quality of a DSoS.
- *Quantitative analysis*, which aims to predict the values of the quality attributes characterising the overall quality of a DSoS.

The above kinds of analyses are complementary. In particular, the results of quantitative analysis are most probably affected by the use of certain means for quality enhancement, whose correct use is verified by the qualitative analysis. On the other hand, the use of such means may be guided by the results of the quantitative analysis at an early design stage. Performing quality analysis, either qualitatively or quantitatively, is not a new challenge and several techniques have been proposed and used for quite a long time [Kobayashi 1978, Laprie 1985, Magee *et al.* 1999, BC2]. Techniques for

qualitative analysis are mainly based on theorem proving and model checking. Typically, models specifying the system's behaviour are built using formalisms like CSP, CCS, Pi-Calculus, TLA, etc. Then, these models are checked against properties that must hold for the system to behave correctly. Techniques for quantitative analysis can be analytic, simulation-based, or measurement-based. Again models specifying the system's behaviour are built using formalisms like Markov-chains, Petri-nets, Queuing-nets, etc. Certain model parameters (e.g., failure rates of the system's primitive elements) are obtained using measurement-based techniques. Then, the models are analytically solved, or simulated, to obtain the values of the attributes that characterise the overall system's quality. The main problem today is that building good models requires lots of experience and effort. System engineers use architecture description languages, and object oriented notations (e.g., OMT, UML) to design the system architecture. Commonly, system engineers are not keen to build quality models using CSP, CCS Markov chains, Petri-nets, Queuing-nets, etc. Hence, the ideal would be to provide an environment that enables the specification of DSoS architectures in a language suitable for system engineers, and that further offers adequate tool support facilitating the specification of models suitable for DSoS quality analysis.

In this chapter, we propose a developer-oriented, architecture-based environment for the specification and quality analysis of (D)SoSs. The specification of DSoS architectures relies on an extensible UML-based architecture description language, which is defined in Section 2.2. Section 2.3 presents an approach that facilitates the qualitative analysis of DSoSs at the architectural level. Similarly, Section 2.4 discusses an approach that facilitates the quantitative analysis of DSoSs at the architectural level. Finally, Section 2.5 concludes this chapter with a summary of its contribution to DSoS architecting and design. It is the long term objective of our work to offer enhanced DSoS development support, based on architecture description, for:

- The design of DSoSs, by focusing on the definition of novel system architectures out of results from the CM WP regarding DSoS-specific architectural styles [IC1] and from the AD WP regarding DSoS-specific dependability mechanisms (see Chapter 3). This will in particular lead to corresponding specialisation of the ADL presented in Section 2.2.

- The analysis of DSoSs, by exploiting in particular results of the VA WP, which will lead to enrich the tool support of the environment accordingly, as well as to offer additional methods for architecture specification. As an example of the latter, the developer may exploit the layered framework proposed in [DMS1] for specifying the architectures of complex DSoSs.

- The implementation of DSoSs, by examining the automatic generation of wrappers regarding the integration of both legacy component systems and mechanisms for dependability, as examined in the AD WP. This will lead to corresponding specialisation of the ADL, and to offer associated tools. For example, the framework presented in Chapter 4 introduces a solution to generate wrappers for error detection, whose one specialisation relies on specification of properties using temporal logic. Corresponding specialisation of the ADL would then lie in enriching the specification of architectural components with temporal logic formulas.

2.2 An Extensible, UML-based Architecture Description Language

2.2.1 Background and Related Work

Architecture Description Languages (ADLs) are notations enabling the rigorous specification of the structure and behaviour of systems [Medvidovic & Taylor 2000]. ADLs come along with tools that facilitate the analysis and the construction of systems, whose architecture is specified using them. Several ADLs have been proposed in the past years and they are all based on the same principles. In particular, the structure of systems is specified using the following basic concepts (see Chapter 1 of the DSoS State of the Art Survey [BC2]): components, connectors and configurations. It is worth noticing that existing ADLs have concise semantics and are widely known and used in academia, but their use in industry is quite limited. Industrials, nowadays, tend to use object-oriented notations for specifying the architecture of their software systems. UML, in particular, is becoming an industrial standard notation for the definition of a family of languages (i.e., UML profiles) for modelling software [UML v1.3]. However, there is a primary concern regarding the imprecision of the semantics of UML. To increase the impact of ADLs in practice, and to decrease the ambiguity of UML, we propose an ADL defined in relation to standard UML elements [Zarras *et al.* 2001][1]. Our main objective is the definition of a set of core extensible language constructs for the specification of components, connectors and configurations. This core set of extensible constructs will facilitate future attempts for mapping existing ADLs into UML [Medvidovic & Rosenblum 1999]. Our effort relates to the definition of architecture meta-languages like ACME [Garlan *et al.* 1997] and AML [Wile 1999]. Our work has similarities with the recent proposal of an extensible, XML-based ADL [Dashofy *et al.* 2001]. Our approach can be the basis for the definition of a standard UML profile for ADLs, while the ADL introduced in [Dashofy *et al.* 2001] can be the basis for a complementary standard DTD used to produce textual specifications from graphical ADL models. The remainder of this chapter defines our extensible ADL, and discusses its specialisation for quality analysis.

2.2.2 Basic Concepts

To define ADL components, connectors, and configurations in relation to standard UML model elements we followed the steps given below:

- Identify standard UML element(s), whose semantics are close to the ones needed for the specification of ADL components, connectors and configurations.
- If the semantics of the identified element(s) do not exactly match the ones needed for the specification of components, connectors, and configurations, extend them properly and define a corresponding UML stereotype(s)[2].
- If the semantics of the identified element(s) match exactly, adopt the element(s) as a part of the core ADL language constructs.

---

[1] There is ongoing work at OMG on the definition of UML elements for architecture description. However, so as not to delay our work, we have decided not to wait for the release of the standard; we will take account of it when available.

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

Component Definition

As discussed in the literature [Garlan *et al.* 2000], various UML modelling elements may be used to specify an ADL component. The most popular ones are the Class, Component, Package, and Subsystem elements. From our point of view, the UML Component element is semantically far more concrete compared to an ADL component, as it specifically corresponds to an executable software module. Moreover, the UML Class element is often considered as the basis for defining architectural components. However, a UML class does not directly support the hierarchical composition of systems. It is true that the definition of a UML Class may be composite, consisting of a number of constituent classes. However, a class specification cannot contain the relationships between constituent classes. Consequently, if an ADL composite component is mapped into a UML class, its definition may comprise a set of constituent components but we then have no means to describe how they are connected through connectors. Technically, to achieve this we would need to define a Package containing the UML class definitions and a static structure diagram showing how they are connected. However, packages cannot be instantiated or associated with other packages and are therefore not adequate for specifying ADL components. This leads us to use the UML Subsystem element to model ADL components. A UML Subsystem is a subtype of the UML Package and Classifier elements; it is defined as *"a grouping of model elements, of which some constitute a specification of the behaviour offered by the other contained elements"* [UML v1.3]. UML subsystems may be instantiated multiple times, and associated with other subsystems. Based on the above, we define an ADL component as a UML Subsystem, that may provide and require standard UML interfaces. The ADLComponent element is further characterised by a property, named "composite", which may be true or false depending on whether or not a component is built out of other components and connectors. More formally, we have the OCL[3] definition of the ADLComponent stereotype given below.

OCL Definition
ADLComponent: -- Additional Operations -- provides : Set(Interface) provides = self.provision.client->select(i | i.oclIsKindOf(Interface)) requires : Set(Interface) requires = self.requirement.supplier->select(i | i.oclIsKindOf(Interface)) -- Well-formedness rules -- self.baseClass = Subsystem and self.extendedElement.Instantiable = true

Connector Definition

A connector is an association representing the protocols through which components may interact. Hence, the natural choice for specifying it in UML is by stereotyping the standard UML Association element. A connector role corresponds to an association end. Moreover, the distinctive feature of a connector is a non-empty set of interfaces, named "Interfaces", representing the specific parts of

[3] The Object Constraint Language (OCL) is a first order logic notation, used to define constraints associated with UML model elements. The definitions in OCL are given here for readers who would be interested in reusing the provided extension for integration in an existing UML tool. Readers who are not expert in OCL may however skip these definitions, since they are given informally in the text.

components' functionality playing the roles. Each interface out of the set must be provided by at least one associated component. Equally, each interface out of the set must be required by at least one associated component[4]. Formally, we have the corresponding OCL definition[5], given below:

OCL Definition

```
ADLConnector:
-- Additional Operations --
interfaces : Set(Interface)
interfaces = self.extendedElement.taggedValue->select(tv | tv.name = "Interfaces").value
rolesProtocol : Set(String)
rolesProtocol  = self.extendedElement.taggedValue->select(tv | tv.name = "RolesProtocol").value
bodyProtocol : String
bodyProtocol = self.extendedElement.taggedValue->select(tv | tv.name = "BodyProtocol").value
-- Well-formedness rules --
self.baseClass = Association and  self->interfaces().value->isNotEmpty() and
self. ExtendedElement.allConnection->forall(
  ae | ae.type->requires()->exists(
   i | self->interfaces()->includes(i) implies self. ExtendedElement.allConnection->exists (
      ae'| ae'.type->provides()->includes(i)
   ) and
 self. ExtendedElement.allConnection->forall(
  ae | ae.type->provides()->exists(
   i | self->interfaces()->includes(i) implies self. ExtendedElement.allConnection->exists (
      ae'| ae'.type->requires()->includes(i)
   )
```

So far, we have considered connectors as associations representing communication protocols. However, we must not ignore the fact that, in practice, connectors are built from architectural elements, including components and more primitive connectors. Taking CORBA for example, a CORBA compliant connector can be seen as a combination of ORB functionality and basic CORBA services interacting using a primitive RPC connector. Hence, it is necessary to support hierarchical composition of connectors. At this point, we face a technical problem: UML Associations cannot be composed of other model elements. However, there exists a standard UML element called Refinement defined as *"a dependency where the clients are derived by the suppliers"*. The refinement element is characterised by a property called mapping. The values of this property describe how the client is derived by the supplier. Based on those remarks, and in order to support the hierarchical composition of connectors, we define the ADLConnectorRefinement stereotype whose base class is the standard UML Refinement element and is used to define the mapping between a connector and a composite

[4] In the provided definition, a required interface that gets bound to a provided interface through a connector is required to be equal to the provided interface. This requirement could be weakened by considering a matching definition based on the subtyping relationship; it is our plan to change the component definition accordingly.

[5] Notice that the given definition of connectors does not enable to directly bind connectors, as supported by some architectural styles. However, direct binding of connectors is enabled via the definition of connector refinement, which is introduced in the next paragraph.

component that realises the connector[6]. Formally, we have the corresponding OCL definition given below:

OCL Definition
ADLConnectorRefinement: self.baseClass = Refinement and self.extendedElement.client.oclIsKindOf(Association) and self.extendedElement.supplier.oclIsKindOf(Subsystem) and self.extendedElement.supplier.stereotype.oclIsKindOf(ADLComponent) and self.extendedElement.supplier.stereotype.composite = true

Configuration Definition

A configuration specifies the assembly of components and connectors. In UML, the assembly of model elements is specified by a model. The corresponding semantic element of a model is the standard UML Model element, defined as "*an abstraction of a modelled system specifying the system from a certain point of view and at a certain level of abstraction...the UML Model consists of a containment hierarchy where the top most package represents the boundary of the modelled system"*. Hence, a configuration is actually a UML model, consisting of a containment hierarchy where the top-most package is a composite ADLComponent.

The given definition of configuration is weak in that it enables the description of any architectural configuration provided it complies with the well-formedness rules associated with the component and connector elements. This results from our concern to support the description of various architectural styles and in particular DSoS-specific styles, which possibly come along with specific ADLs, as is, e.g., the case with the C2 style [Medvidovic *et al.* 1999]. Constraints that are specific to a style are introduced through the definition of a corresponding extension of the ADLConfiguration element, possibly combined with extension of the UML elements for component and connector definition.

2.2.3 Tools

The basic ideas described so far for the specification of software architectures have now been realised by a prototype implementation of the architecture-based development environment, which makes use of an existing UML modelling tool. More specifically, we use the Rational Rose tool[7] for the graphical specification of software architectures. The Rational Rose tool allows the definition of user specific add-ins that facilitate the definition and use of stereotyped elements[8]. Given the aforementioned facility, we implemented an add-in that facilitates the specification of architectural

---

[6] Note that association classes cannot be used here since ADL components are modelled using the subsystem element.

[7] www.rational.com

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

descriptions using the elements defined in the previous subsection. Moreover, we use an already existing add-in, which provides functionalities for generating XMI textual specifications of architectures specified graphically using the Rational Rose tool. Those textual specifications shall serve as input to the tools we use for qualitative and quantitative analyses of architectures. In addition to the previous add-ins, we developed an OCL verifier. Note at this point that we could have used an already existing verifier implemented in Java [Richters *et al.* 2000]. However, given that the expected complexity of our models is high we preferred to go for a more efficient implementation based on OCAML[9]. OCAML is a functional language developed at INRIA, which belongs to the family of ML languages. It has been used to efficiently develop large applications like the COQ theorem prover[10]. The OCL verifier accepts as input XMI architectural specifications and OCL constraints, and generates an OCAML program which checks whether the constraints hold for the architecture. More specifically, for all the elements defined in the architectural specification, the verifier generates corresponding OCAML definitions. Those definitions are instances of OCAML definitions of the base architectural elements (ADL Components and Connectors). The verifier further translates OCL constraints into corresponding OCAML logical expressions, which can be executed so as to check whether they hold or not. The use of the tools for the specification and verification of software architectures is further explained through an example given in the following subsection.

2.2.4 Example

To exemplify the use of the tools presented in the previous subsection for the specification of DSoS architectures, we use examples taken from the travel agent case study [DMS3]. The Travel Agent DSoS (TA) that is considered here offers services for flight, hotel, and car reservations. It consists of the integration of different kinds of existing component systems supporting air companies, hotel chains, and car rental companies.

Figure 2-1 gives a screen shot of the actual architecture of the TA, as specified using the UML modelling tool, which we customised. The TA comprises the TravelAgentFrontEnd component, which serves as a GUI for potential customers wanting to reserve tickets, rooms, and cars. The TA further includes the HotelReservation, FlightReservation, CarReservation components, which accept as input individual parts of a customer request for hotel, ticket and car reservation, and translate them into requests for services provided by specific hotel, air company and car company components. The set of the hotel components is represented by the Hotels composite component. Similarly, the sets of air company and car company components are represented by the AirCompanies and CarCompanies composite components. These three composite components abstract more concrete component realisations. In particular, interactions with the abstracted, embedded components may be via either multicast or point-to-point communication, depending on the behaviour of the component and connector with which the embedding composite component is bound. Both realisations will be considered in the next sections. The HotelReservation, FlightReservation, and CarReservation components provide standard interfaces to the front-end component and require a number of

[9] www.caml.inria.fr/ocaml/

[10] www.coq.inria.fr

interfaces from the existing component systems. Figure 2-2 gives, as an illustration, the detailed specification of the FlightReservation component. Two different kinds of ADLConnector elements are used in our architecture. HTTP connectors represent the interaction protocol among customers and the TravelAgentFrontEnd, and among components translating requests and existing component systems implementing Web servers. The RPC connector represents the protocol used among the front-end component and the components that translate requests. The multi-party connectors abstract complex connector realisations, which may actually be refined into various protocols, depending on the intended behaviour. For instance, the HTTP connector depicted at the bottom of the figure may be refined into a number of bi-party connectors as well as into a complex multicast connector. Figure 2-3 gives the specification of the abstract HTTP connector among the AirCompanies and the FlightReservation components. The AirCompanies component plays the role of the HTTP server, while the FlightReservation plays the role of the HTTP client.

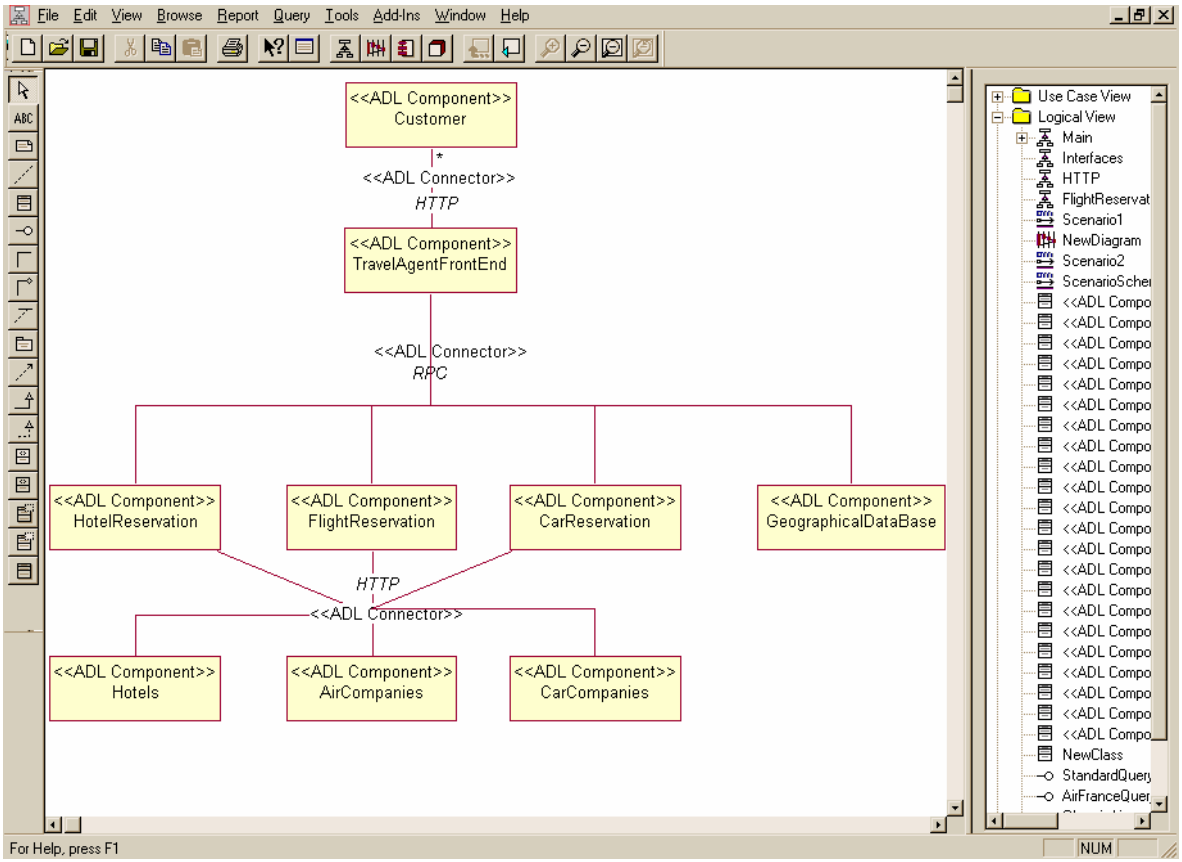

Figure 2-1: The Architecture of the Travel Agent DSoS.

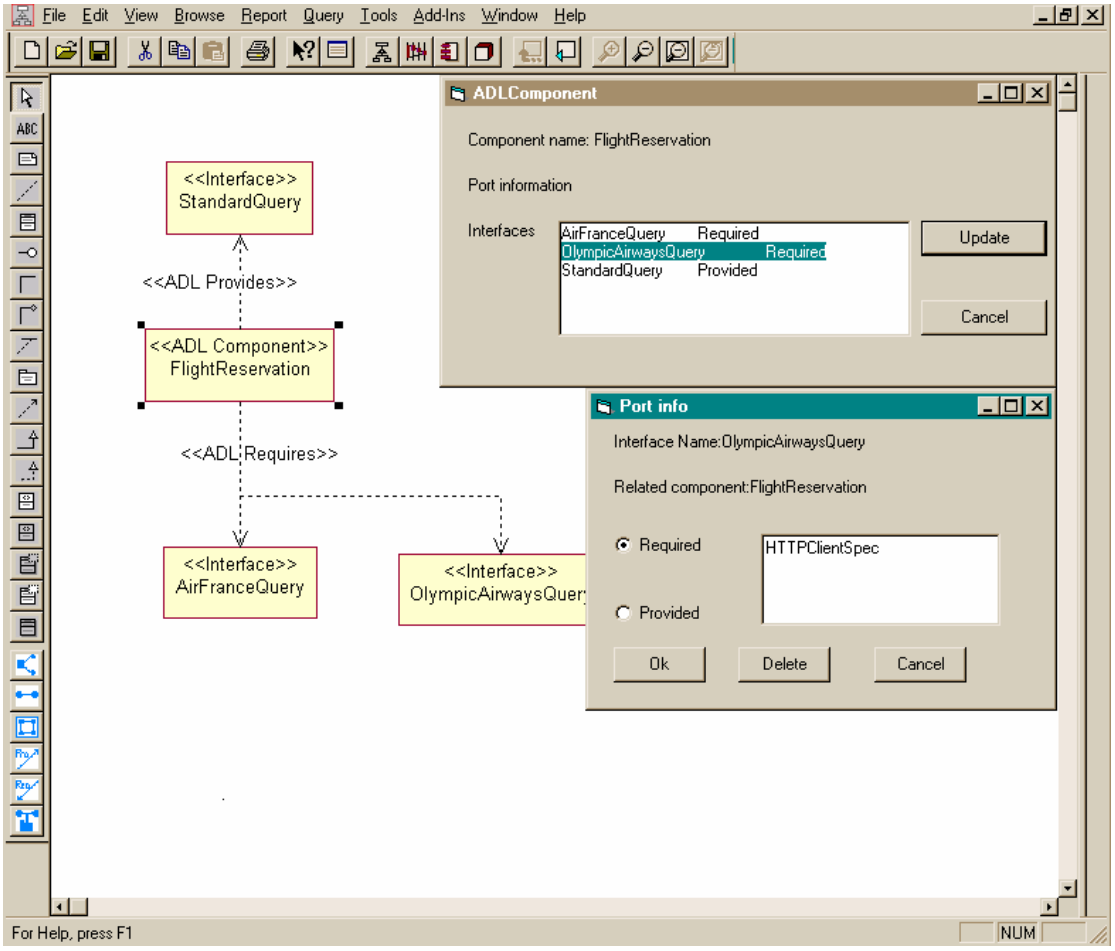

Figure 2-2: Detailed specification of the FlightReservation component.

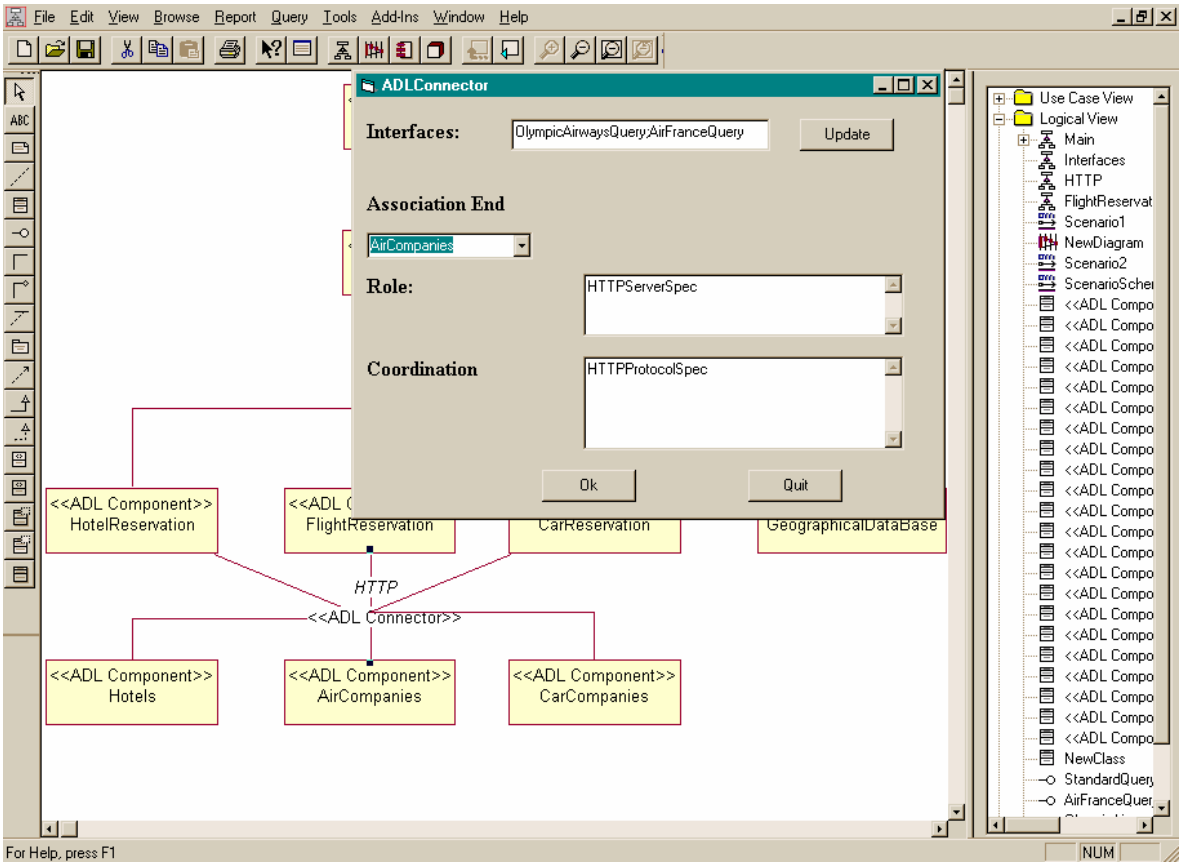

Figure 2-3: Detailed specification of the HTTP connector.

Getting to the verification of OCL properties using the OCL verifier, consider the case of the HTTP connector between AirCompanies and FlightReservation. For this element, the OCL verifier generates a corresponding OCAML definition, named "HTTPOcaml", which is an instance of the predefined adlConnectorOcaml OCAML definition; HTTPOcaml is characterised by a list of interfaces containing the OlympicAirwaysQueryOcaml and the AirFranceQueryOcaml interfaces. The set of association ends of the connector contains the FlightReservationOcaml component and the AirCompaniesOcaml component. More precisely, we have the following OCAML expression generated for the HTTP connector:

OCAML Definition of the HTTP Connector

```
let self = new adlConnectorOcaml "HTTPOCaml"
                    [new Interface "OlympicAirwaysQueryOcaml"; new Interface "AirFranceQueryOcaml"]
                    [new adlComponentOcaml "AirCompaniesOcaml"
                        [] [new Interface "OlympicAirwaysQueryOcaml""; new Interface "AirFranceQueryOcaml"]
                     new adlComponentOcaml "FlightReservationOcaml "
                        [new Interface "OlympicAirwaysQueryOcaml" "; new Interface "AirFranceQueryOcaml"][]
                    ];;
```

Finally, given the OCL expressions for the ADLConnector that is defined in Section 2.2.2, the OCL verifier generates the OCAML expression that is given in appendix A1.1, which for HTTPOCaml evaluates to true. Hence, the definition of the HTTP connector is well-formed.

This example has illustrated base architectural definition using the extensible ADL provided by our environment. The developer defines the architectural elements making up his system using a graphical tool (e.g., see Figures 2-1 to 2-3), and the architecture is automatically checked for well-formedness. Notice that the provided definitions may be reused for the design of other systems using the repository of modelling elements. The two next sections present specialisation of the ADL regarding support for system analysis, showing in particular how to make more tractable for developers the specification of architectures that may be thoroughly analysed, using existing formal modelling techniques although not requiring extensive knowledge of these techniques from developers.

2.3 Qualitative Analysis

This section addresses qualitative analysis of DSoSs from their architectural description, which is typically achieved using either model checking or theorem proving. Hence, future enhancement of the development environment in this area will be based on results from the VA workpackage on model checking.

2.3.1 Background and Related Work

Since the early work on ADL definition, there has been a number of efforts on the definition of ADLs based on formal specification, for the qualitative analysis of system architectures from the standpoint of offered functional properties. Specifically, a number of existing ADLs allow for the behavioural analysis of system architectures [BC2]. With respect to the formalism used for describing behaviour, we can classify these ADLs into two groups: (i) those using some form of logic, and (ii) those using

modelling languages. Each formalism has its advantages as well as its disadvantages. Using logic allows one to clearly express what task the architecture should complete, without having to describe the particular mechanism used to achieve this. This allows for greater flexibility on the developers' side, since they are more free to choose among possible implementations, while still adhering to the requirements expressed in the architecture. It is also easier to describe and prove properties of families of systems, as well as properties in infinite domains. For example, using logic, it is easy to describe systems consisting of N replicas of a particular component, without having to consider a particular value for N. It is also easy to specify general properties on data structures (e.g., messages) without having to restrain these to a finite domain, as is the case with model checkers. Modelling languages, on the other hand, provide a formalism that looks more natural to developers, due to their resemblance to programming languages. In addition, it is easy to automatically validate models described in a modelling language against some property, by using a model checker. The equivalent tools used for proving properties of models described in logic (i.e., theorem provers) demand substantial user intervention and can be quite complex to use, usually needing a large period of time to get used to and learn how to use them effectively, which is not so much the case for model checkers. Another significant advantage of model checkers over theorem provers is that, when the model is not correct, they can provide the user with a counterexample that highlights the erroneous behaviour. With theorem provers, one can never be sure whether a theorem that the tool cannot prove is indeed incorrect, or whether the tool and its user are simply not able to prove it for some other reasons. Indeed, the lack of appropriate clues is one of the aspects that makes theorem provers difficult to use. We should mention here that one of the current trends in formal methods is in integrating various different techniques, such as model checking, abstract interpretation, static checking and decision procedures, which, presumably, will allow for a greater applicability on real-world systems. Early examples of such unification are the PVS theorem prover [PVS 2001], which has a small model checker embedded, or the Cadence SMV model checker [SMV 2001], which contains a small theorem prover. Other work that may be of interest, as far as the integration of theorem provers and model checkers is concerned, is [Rushby 1999], while in [Roscoe and Broadfoot 1999], the authors use data independent techniques along with CSP and its model checker, FDR, in order to verify that cryptographic protocols having an infinite number of resources, such as secret keys, are free of attacks.

Regarding the support offered in our environment for the qualitative analysis of (D)SoSs at the architectural level, we have chosen a modelling language as a formalism, so as to increase as far as possible the automation of the resulting solution, relying as little as possible on user intervention. Then, to perform qualitative analysis, developers a priori have to learn these formalisms and tools. They further have to derive mappings between the basic architectural concepts (e.g., components, connectors, ports, roles, etc.) they use to specify software architectures and the basic constructs provided by the formalism that is to be used for specifying the system's functional behaviour (e.g., processes, channels, etc.), if these mappings are not already provided by the ADL itself. Neither of the previous tasks is however straightforward for everyday developers who are very experienced and educated on the use of object-oriented modelling methods, e.g., using UML, and several programming languages like C, C++, Java, but are not experts in logic and process algebra. Up to now, we were able to identify very few approaches that try to alleviate the previous complexities towards rendering the use of qualitative analysis more tractable to nowadays developers. In particular, in [Lilius & Paltor 1999], the authors propose a tool for model checking UML models. Developers have to specify these using state-chart diagrams, which are then used for generating

models that serve as input to the SPIN model checker [Holzmann 1997]. However, state-chart specifications of system behaviour are quite low level and certainly not easy to produce. Take for instance the usual case where developers need to specify loops, procedure calls, synchronisation and communication using state-charts. In this case developers would prefer using a modelling language, which resembles more to a real programming language instead of using automata-based notations such as state-charts. Based on the previous, we consider that the approach proposed in [Lilius & Paltor 1999] is not a satisfying solution. In the following, we propose an alternative solution that is based on the specification of the behaviour of architectural model elements using a formal modelling language, which however does not require much expertise from the developer through both adequate specialisation of the ADL and use of a modelling language that is close to a programming language.

2.3.2 Basic Concepts

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

- ADL components are characterized by a property, called "BodyBehavior", whose value can be assigned to a textual specification, given in any behavioral modeling formalism, describing the components' behavior.
- UML interfaces provided/required by ADL components are characterized by a property, called "PortBehavior", whose value describes in some textual specification, the particular protocol used at that point of interaction.
- ADL connectors are characterized by:
    - A property, named "Coordination" (see Figure 3), whose value specifies the role-independent part of the interaction protocol.
    - A set of properties (see Figure 3), named "Role". Each one of these corresponds to an association end, i.e., a role. The value of each property specifies the role-dependant part of the interaction protocol represented by the connector.

2.3.3 Tools

The primary use of model checkers is to identify *errors* in a model. The undesired behaviour/states are, in most cases, described symbolically by the user with some variant of Temporal Logic, such as a linear-time logic (e.g., Linear Temporal Logic - LTL) or a branching-time logic (e.g., Computational Tree Logic - CTL). Their major difference is that the former considers that at any moment there exists only one possible future, while the latter considers that, at each moment, time may split into alternate courses representing different possible futures. Even though it seems at first sight that CTL should be a superset of, i.e., more expressive than, LTL, this is not true. In fact, there are cases that can be expressed with one of them but not with the other. A fuller comparison of them can be found in [Emerson & Halpern 1986], where CTL*, a superset of both, is also presented. We have investigated

three major model checking tools: FDR2[11], SMV[12], and SPIN. Unlike the other two tools, which use modelling languages similar to a normal programming language, FDR2 [Formal Systems (Europe) 2001] is based on the CSP algebra [Hoare 1995]. SMV was originally created for verifying hardware designs, while SPIN [Holzmann 1997] was created for verifying communication protocols. SPIN is an *explicit state* model checker; when trying to verify some property for a system, it may create the whole state space of the system, using nevertheless an *on-the-fly* method for constructing it, so as not to suffer from the state space explosion problem. SMV, in contrast, is a *symbolic* model checker; it will create a symbolic representation of the state space that is usually substantially smaller than its explicit representation. FDR2 is an *on-the-fly, explicit state* model checker augmented with various compression strategies designed to reduce the size of the state space representation.

Of the three aforementioned model checking tools, we have chosen to use SPIN for a number of reasons. First, SPIN is provided free of charge along with its source code. SMV is also provided free of charge and some versions of it like CMU's SMV [Formal Methods group, CMU 2001] and NuSMV [Cimatti *et al.* 2001] also make available the source code. SPIN has further spawned quite a large interest, with its own annual conference[13], which groups the continuing efforts of the SPIN community to improve it, by treating subjects as model checking, automatic generation of invariants, automatic construction of abstract models, model slicing, real-time verification, model checking Java programs, UML models, or even verifying AI spacecraft control systems used by NASA. Another vote of confidence is the use of SPIN, in the next version of STeP [Bjorner *et al.* 2000], for model checking [Browne *et al.* 2000]. The fact that the modelling language used with SPIN, PROMELA[14], as well as the one used with SMV, look very much like the programming language C makes them good candidates for actual use since the designers and the developers will not feel intimidated by them. Finally, the reason for choosing SPIN over SMV is that it has built-in channels with which we can easily model the bindings among the output and input ports of components. Thus, a model of an architectural element, such as a simple lossless FIFO connector (i.e., a pipe) can be described in SPIN very naturally, as shown below.

A Lossless FIFO Connector Modelled with PROMELA/SPIN

```
proctype FIFO_connector (chan MyInChannel, chan MyOutChannel)
{
  Msg a_message ;
  do
  :: true ->  /* Repeat this forever */
    MyInChannel? a_message; /* Read a message */
    MyOutChannel!a_message /* Send a message */
  od}
```

While the use of SPIN in our environment follows from the aforementioned reasons, this does not prevent us from later integrating other model checking tools, given the extensibility of our ADL. In

[11] Failures-Divergence Refinement

[12] Symbolic Model Verifier

[13] See http://netlib.bell-labs.com/netlib/spin/whatispin.html for the on-line proceedings.

[14] PROtocol Meta LAnguage

particular, it is part of the project's work in the VA workpackage to investigate the benefit of using FDR2 for DSoS analysis, which may further lead to enrich the architecture-based development environment accordingly.

A PROMELA model consists of a number of independent processes, i.e., each one has its own thread of execution, which communicate either through global variables or through special communication channels by message-passing, as is done in CSP, at least in its machine readable version. Therefore, the mapping of our basic architectural elements to the constructs of PROMELA can be done in a way analogous to the mapping used by the Wright ADL for CSP [Allen & Garlan 1997]. In particular, in [Allen & Garlan 1997], for each component, connector, port and role a corresponding process can be generated. Each generated process shall communicate with the rest through channels, generated as prescribed by the configuration of the DSoS. However, the previous mapping results in the generation of a large number of processes and requires a substantial amount of resources for model checking. To alleviate this problem, we have chosen to generate independent processes for each component and connector specified in a DSoS architectural description, while for each port and role we generate PROMELA inline procedures. This inline procedure construct of PROMELA allows us to define new functions that can be used by processes but which do not introduce their own threads of execution. In this manner, we can keep the number of different processes that the model-checker will be asked to verify to a minimum, thus allowing for verification of larger architectures. Then, for each port of an ADL component, we declare in the PROMELA description of the component, a communication channel named after that port. This channel will be used by the process related to the ADL component for communicating through that specific port. Since ports of ADL components are bound to specific roles of ADL connectors, their channels are passed as arguments to the processes created for these connectors, at the time of their initiation. Thus, messages sent from a process of an ADL component at a channel corresponding to a port of it will be received by a process of an ADL connector. Similarly, messages sent from a process of an ADL connector to a channel it has received as argument at initiation time, will be in fact received by a process of an ADL component, whose port was mapped to that channel. Based on the mapping discussed above, the steps to be followed for generating a complete PROMELA model from an architectural description are given below.

	Generating PROMELA Models from Architectural Descriptions
Component	For each component c: • For each port p of c, create an "inline" procedure whose name is the appending of the component's and the port's name, i.e., c_p. This procedure contains the behaviour of the respective port p. For interacting with its environment, c_p uses a channel named after the port's name, i.e., p.
Connector	For each connector c: • Create a "proctype" named after the connector. Unlike the processes corresponding to ADL components which take no arguments, these processes receive as arguments at initiation time the channels they will be using for their respective roles. These channels are named after the roles themselves.
Configuration	Create a special process called "init" in PROMELA, which will be responsible for instantiating the rest of the architecture. More specifically: • The "init" process creates as many instances of the processes corresponding to particular ADL components, as there are instances of these components in the configuration. • Afterwards, it does the same for each instance of an ADL connector but it uses the

	attachments of component ports to connector roles to deduce the specific channels that should be passed as arguments to the processes corresponding to the connector.

Regarding, more specifically, the use of the above tool support for qualitative analysis of the DSoS quality, we have been actively working on tool support for assisting the generation of the middleware architecture for a given DSoS. This work builds on past experience of INRIA in this area[15], where we have proposed an ADL-based toolset for the systematic integration of middleware architectures within applications (basically, retrieving the adequate middleware services, and generating the adequate wrappers for binding among them and the application), focussing on middleware architectures offering one type of non-functional property. In the context of the DSoS project, we have been concentrating on assisting the design of middleware architectures enforcing distinct types of properties (e.g., security and availability) from existing middleware architectures enforcing each of the properties, individually. Hence, we have investigated a solution to the systematic composition of middleware architectures to elaborate more complex ones. The proposed solution lies in a composition tool based on middleware architecture modelling using SPIN[16] [Kloukinas & Issarny 2001]. Another, complementary, tool identifies valid compositions from a structural point of view, as directly supported by architecture modelling, so as to limit the state space searched by the model checker.

2.3.4 Example

Taking the TA example, we show how the model-checking toolset made available in our environment enables qualitative assessment of the system's quality. A typical property that is often required over RPC-based connectors (including the HTTP connector) is for message exchange to be reliable (assuming the absence of failure of the underlying infrastructure) and for reply messages to be received by the client in the order it sent the corresponding request messages. The former requirement is typically met by implementing the RPC protocol over TCP. On the other hand, the latter is the responsibility of the connector realisation, possibly in conjunction with the server. For instance, the HTTP/1.0 and HTTP/1.1 versions of the HTTP protocol differ in that the latter supports persistent connections and allows pipelining of request messages, which leads to the explicit requirement that the server ensures it sends back reply messages in the order it received the corresponding request messages.

Appendix A1.2 gives a PROMELA specification of part of the TA example, focusing on the flight service, where we consider both realisations of the HTTP protocol as well as two realisations of the Web servers, which process request messages sequentially and concurrently, respectively. SPIN is then used to assess whether ordered delivery of reply messages to the clients is guaranteed by the TA model[17], for the four combinations of HTTP and Web server versions. The model checking identified an erroneous architecture for the TA, in which Web components interact via HTTP/1.1 and the Web servers handle request messages concurrently. Note that the given specification is actually integrated

[15] http://www-rocq.inria.fr/solidor/work/aster.html

[16] This work was another reason for choosing SPIN; it can provide counter-examples for all existing errors in a system, instead of just the first one it finds.

[17] We use here three colour messages [Wolper 1986, Aggarwal *et al.* 1990].

within the UML-based architecture model of the TA according to the ADL specialisation and modelling guidelines, respectively discussed in Sections 2.3.2 and 2.3.3, and can, in principle, be derived from it automatically. However, this facility is still under development within our environment and its description would unduly complicate this presentation. Appendix 1.2 gives the PROMELA system model that, it is planned, would be generated from the integrated UML architectural model. For the sake of conciseness, the given PROMELA specification includes a single HTTP server module that groups all four cases (use of HTTP/1.0 vs 1.1, and sequential vs concurrent handling of request messages).

2.4 Quantitative Analysis

This section addresses easing the task of developers for performing quantitative analysis of DSoSs from their architectural description, focusing more specifically on performance and reliability evaluation. Hence, future enhancement of the development environment in this area will be based on results from the VA workpackage. In particular, the DSoS Preliminary Dependability Modelling Framework [DMS1] proposes a layered approach for describing a SoS. The individual layers are:

- The user layer, describing a user/service profile.
- The function layer, describing basic functions provided by the SoS and combined in a way prescribed by the profile of each user.
- The service layer, describing the software architecture that provides the functions specified at the function layer.
- The resource layer, describing the execution platform used to execute the software architecture described at the service level.

All four layers constitute the specification of a DSoS architecture. Moreover, [DMS1] details how to use the DSoS layered specification to systematically evaluate the availability of DSoSs. More specifically, if there exists no strong dependencies among the elements specified at the function layer, availability can be evaluated using traditional combinatorial logic. For example, the availability of a particular realisation of a function equals to the product of the availability of the services needed to realise the particular function. If more than one alternative realisations exist, the availability of the function is the sum of the availability of the alternative realisations. However, if dependencies exist among the different elements of the function layer, a modelling approach based on Markov-chains, or stochastic Petri nets, considering the three lowest layers of the SoS architectural description is required, as it is more faithful to reality. In this case, models get complicated and inexperienced developers may need substantial help in their development. In this section, we present an approach for automating the generation of complex models from SoS architectural descriptions, regarding both performance and reliability evaluation.

2.4.1 Background and Related Work

Pioneer work on the quantitative analysis of software systems in architecture-based development includes Attribute-Based Architectural Styles (ABAS) proposed in [Klein *et al.* 1999][19]. In general, an architectural style includes the specification of types of basic architectural elements (e.g., pipe and filter) that can be used for specifying a software architecture. Moreover, an architectural style includes the specification of constraints on using those types of architectural elements and patterns, describing the data and control interaction among them. An ABAS is an architectural style, which additionally provides modelling support for the quantitative analysis of a particular quality attribute (e.g., performance, reliability, availability). More specifically, an ABAS includes the specification of:

- *Quality attribute stimuli*, i.e., events affecting the quality attribute of the system (e.g., failures, service requests).
- *Quality attribute parameters*, i.e., architectural properties affecting the quality attribute of the system (e.g., faults, redundancy, thread policy).
- *Quality attribute measures* characterising the quality attribute (e.g., the probability that the system correctly provides a service for a given duration, mean response time).
- *Quality attribute models*, i.e., traditional models that formally relate the above elements (e.g., a Markov model that predicts reliability based on the failure rates and the redundancy used, a Queuing network that enables predicting the system's response time given the rate of service requests and based on the performance parameters).

From the general definition of ABAS, there is a problem when trying to produce a quality attribute model from an architectural description that incorporates the specification of quality attribute stimuli, parameters and measures. Since the architecture is not coupled with scenarios specifying the system usage, it is difficult to analyse the performance or reliability of a system for the set of all possible system behaviours. Typically, only a subset of those behaviours is of interest. This subset of system behaviours manifests the way the system is used and is often called a service (or user) profile. In [Kazman *et al.* 2000], the authors introduce an Architecture Tradeoff Analysis Method (ATAM) where the use of an ABAS is coupled with the specification of a set of scenarios, which roughly constitutes the specification of a service profile. ATAM has been tested for the analysis of qualities like performance, availability, modifiability [Kazman *et al.* 1999 c], and real-time behaviour [Kazman *et al.* 1999 b]. In each case, quality attribute models (e.g., Markov models, queuing networks, etc.) were manually built given the specification of a set of scenarios and the ABAS-based architectural description. However, in [Kazman *et al.* 2000], the authors recognise the complexity of this task. Moreover, it is our opinion that the need to manually generate quality attribute models significantly decreases the benefits of using a disciplined method such as ATAM for the quantitative analysis of software systems. ATAM is a promising approach for doing things right. Nowadays, however, there is a constant additional requirement for doing things fast and easy. Asking software system engineers to build performance and reliability models from scratch is certainly a drawback.

[19] Notice that there is work on transforming UML models into dependability and performance models, which is now part of the work on defining a UML profile for modelling real-time systems. However, we focus here on the work done at SEI, as it is specifically aimed at quantitative analysis in the context of architecture-based development and is at least as elaborated as work done in the context of UML.

The objective of the approach presented in this section is to overcome this drawback by automating the generation of quality attribute models from architectural descriptions. To accomplish this goal, there is a need for specifying the mapping between architectural descriptions and traditional models for quantitative analysis. Hence, we need more formal definitions of ABAS. Indeed, it is not feasible to generate traditional quality attribute models starting from scenarios described in natural language and architectural descriptions within which the relationships among basic architectural elements and quality attribute measures, parameters and stimuli are not precisely defined.

Based on the preceding remarks, in the following subsections, we present the definitions of architectural styles that enable the modelling of performance and reliability stimuli, parameters and measures. The architectural styles are defined using the specialisation of the core ADL language constructs of Section 2.1. Moreover, we give the definitions of the mappings between architectural models and traditional performance and reliability models, hence allowing the automatic generation of the latter from the former. This proposal builds on previous work of INRIA in the context of the ESPRIT LTR C3DS[20] project, where we proposed a solution for the generation of reliability and performance models from the description of workflow schemas enriched with quality parameters. This solution has since been integrated within UML modelling of workflow-based systems [Zarras & Issarny 2001] and generalised to UML-based architecture modelling, as presented in the following.

2.4.2 Basic Concepts

As previously stated, to perform quantitative analysis we have to specify a service profile, i.e., a set of scenarios, describing how the inspected system is used. In UML, scenarios are specified using UML collaboration or sequence diagrams. The semantics of both collaboration and sequence diagrams are given by the UML Collaboration semantic element. The UML Collaboration element *"defines the context for performing tasks defined by interactions"*. A UML Interaction *"specifies messages sent between instances performing a specific task"*. In our particular case, a scenario is a collaboration specifying the interaction among a set of component and connector instances, structured as prescribed by the configuration of the inspected system. In the remainder, we discuss how the definitions of the base ADL elements are extended to support the specification of quality stimuli, parameters and measures for performance and then reliability analysis. Notice that for performance and reliability analysis, the architectural model must integrate deployment information such as the node on which the components run. Hence, in the following, we refer to architecture-specific UML elements as well as traditional ones (i.e., the UML message element from UML collaborations and the UML node element that serves to specify processing entities).

Performance Stimuli/Parameters/Measures

The basic stimuli affecting performance are the service requests (modelled using messages in UML) that cause the beginning of a scenario. The initiation of those requests is characterised by a particular statistical pattern.

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

Performance Stimuli	Value Range	Architectural Element
statistical-pattern	constant/discrete/histogram/normal/uniform/Poisson/ erlang/other exponential distributions	Message

The basic parameters affecting performance are: the thread-policy of ADL components; the service rate of nodes on which components are deployed, defined as the number of work units performed per time unit; the work-demands of a component, defined as the number of work units spent for providing a service; the scheduling policies of ADL connectors; the capacity and delays of connectors.

Performance Parameters	Value Range	Architectural Element
service-rate	Real	Node
thread-policy	single, multi, pool	Component
work-demands	constant/discrete/histogram/normal/uniform/Poisson/ erlang/other distr.	
scheduling-policy	FIFO/LIFO/prio/quantum/order-preserving/ sharing	Connector
capacity	infinite|finite	
delays	constant/discrete/histogram/normal/uniform/Poisson/ erlang/other distr.	

Finally, the basic measures characterising a scenario are[21]: the mean time requests are waiting to be served; the mean time spent to serve requests; the mean response time, defined as the sum of the previous two measures; the mean throughput, defined as the number of requests served per time unit.

Performance Measures	Value Range	Architectural Element
mean-service-time	Real	Message
mean-waiting-time		
mean-response-time		
mean-system-throughput		

Reliability Stimuli/Parameters/Measures

A scenario[22] may fail if instances of components, nodes, and connectors used in it fail because of faults causing errors in their state. The manifestations of errors are failures. Hence, faults are the basic parameters that affect the reliability of an inspected system, while failures are the stimuli causing changes in the value of the reliability measure. As discussed in [Laprie *et al.* 1990, Butler & Johnson 1995], faults and failures can be characterised by the properties given in the tables below. Different combinations of the values of those properties lead to the definition of fault and failure taxonomies, facilitating the automated generation of traditional reliability models.

Reliability Stimuli: Failures	Value Range	Architectural Element
domain	time/value	Component/Connector/Node
perception	consistent | inconsistent	

[21] Although we give only mean values here, other values may be considered, e.g., worst case.

[22] Notice that evaluation for one scenario does not imply that the evaluation will be with respect to a single customer, as this depends on the definition of the given scenario.

Reliability Parameters: Faults	Value Range	Architectural Element
nature	intentional | accidental	Component/Connector/Node
phase	Design | operational	
causes	Physical | human	
boundaries	Internal | external	
persistence	Permanent | temporary	
arrival-rate	Real	
active-to-benign-rate	Real	
benign-to-active-rate	Real	
disappearance-rate	Real	

In addition to faults and errors, another parameter affecting reliability is redundancy. Redundancy schemas can be defined using the base ADL language constructs defined in Section 2.2. More specifically, a redundancy schema is a configuration of redundant architectural elements, which behave as a single fault tolerant unit. According to [Laprie *et al.* 1990], a redundant schema can be characterised by the mechanism used to detect errors, the way the constituent elements execute to serve incoming requests, the confidence that can be placed on the results of the error detection mechanism, and the number of component and node faults that can be tolerated.

Reliability Parameters: Redundancy	Value Range	Architectural element
error-detection	vote/comparison/acceptance	Component
execution	parallel/sequential	
confidence	absolute/relative	
service-delivery	continuous/suspended	
no-of-component-faults	Integer	
no-of-node-faults	Integer	

Finally, the basic reliability measure is the probability that a scenario successfully completes within a given time duration.

Reliability Measures	Value Range	Architectural Element
Reliability	[0..1]	Scenario

2.4.3 Tools

The performance and reliability analysis of DSoSs is supported by automated procedures, which take as input, architectural specifications defined using the basic concepts discussed so far, and generate traditional performance and reliability models. In this subsection, we further detail the definitions of mappings between architectural specifications, and traditional performance and reliability models, given the underlying tools we are using in our prototype. As for the work presented in the previous section, the specific tools that we have chosen to integrate in our environment does not prevent us from integrating other tools, and in particular tools aimed at dependability analysis of DSoSs, according to results from the VA workpackage. In this context, the remainder gives the necessary guidelines about how to integrate such tools.

Mapping Architectural Specifications into Traditional Performance Models

For DSoS performance analysis, we use a tool-set, called QNAP2[23], which provides a variety of both analytic and simulation techniques. QNAP2 accepts as input a queuing network model of the system that is to be analysed. The general structure of a queuing network model is given in Figure 2-4. A queuing network model consists of a set of stations providing services requested by customers. A service is associated with a set of transition rules describing what happens to a customer after the customer is served. A station is further associated with queues that store requesting customers. In a queuing network, we may have special stations, called source stations, whose purpose is to create new customers. Those stations are characterised by a statistical pattern according to which they generate customers.

Given a DSoS architectural description and a particular service profile describing how the DSoS is used, the steps for mapping it to the corresponding queuing network are the following. First, a set of stations is generated for every collaboration included in the service profile. A station corresponds to a node on which instances of DSoS components are deployed. Then, for every component instance and for every interface it provides, corresponding services are generated and associated with the stations that represent the nodes on which the component instance is deployed. For all the connector instances used in the collaboration, corresponding queues are generated and associated with these stations. Formally, the post condition for these steps is:

Mapping Architectural Configurations into Queuing Networks

```
Collaboration:
self.associationRole->forall(connector |
  connector.connection->forall(component |
            component.base.implementation.deployment->forall(
             n |  self.queuingNetworkModel.station->exists(
               st | st.name = n.name and
          st.rate = n.rate and
               component.base->provides()->forall(
                 i | st.service->exists(
            s | s.name = i.name and
            s.work-demands = el.work-demands->select(wd | wd.name = i.name)
   ) ))) ) and
   self.queuingNetworkModel.queue->exists(
            q | q.name = connector.name and
            q.capacity = connector.capacity and
            q.scheduling-policy = connector.scheduling-policy
   ))
```

[23] www.simulog.com

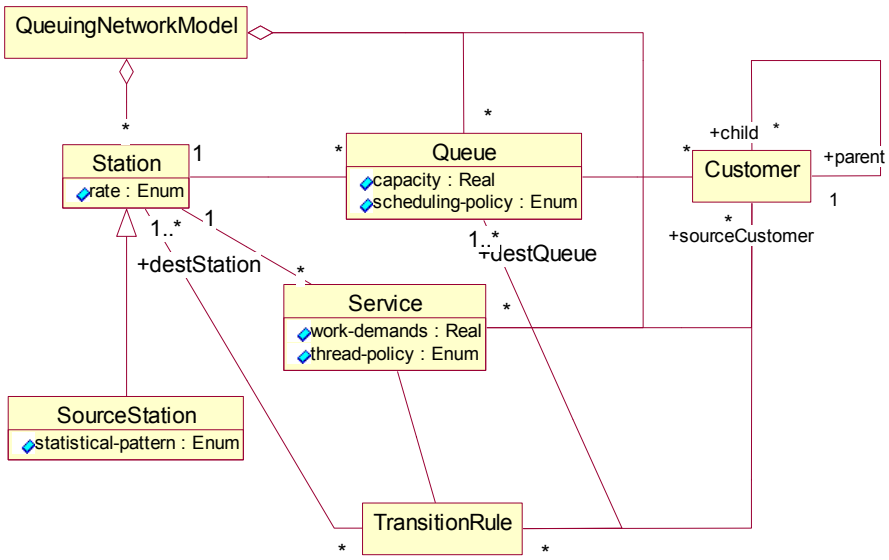

Figure 2-4: The general structure of a queuing network.

A subset of the generated stations are source stations, i.e., they represent nodes hosting components responsible for initiating a collaboration. A component, mi.sender, is responsible for initiating a collaboration if it is the sender of messages mi, the sending of which is not caused by other messages (i.e., mi.activator.isEmpty() holds). A source station creates new customers, ci, representing initiation messages, according to the statistical pattern that characterises those messages. ci customers are transferred according to a generated transition rule tr. According to tr, the destination queue used to store ci is the one that represents the connector between mi.sender and mi.receiver. The destination station is the one that represents the node on which mi.receiver is deployed. The destination service is the one that represents the corresponding interface required by mi.sender. Serving an initiation customer may result in the creation of new customers corresponding to messages in reply to message mi. Those customers are transferred according to transitions rules generated as discussed above.

Mapping Architectural Specifications into Traditional Reliability Models

To perform reliability analysis, we use a tool named SURE-ASSIST [Butler 1992]. The tool calculates reliability bounds given a state space model describing the failure and repair behaviour of the inspected system[24]. The tool was selected because it is very highly rated compared to other reliability tools [Geist & Trivedi 1990] and because it is available for free. However, the automated support provided by our environment for reliability analysis can be coupled with any other tool that accepts as input state space models. The general structure of a state space model is given in Figure 2-

[24] In the following, we do not consider the repair behaviour, which is handled as the failure behaviour, with the additional definition of corresponding stimuli, parameters and measures.

5. Briefly, a state space model consists of a set of transitions between states of the system. A state describes a situation where either the system operates correctly, or not. In the latter case, the system is said to be in a death state. The state of the system depends on the state of its constituent elements. Hence, it can be seen as a composition of sub states, each one representing the situation of a constituent element. A state is constrained by the range of all possible situations that may occur. A state range can be modelled as a composition of sub state ranges, constraining the state of the elements that constitute the system. A transition is characterised by the rate at which the source situation changes into the target situation. If, for instance, the difference between the source and the target situation is the failure of a component, the transition rate equals to the failure rate of the component.

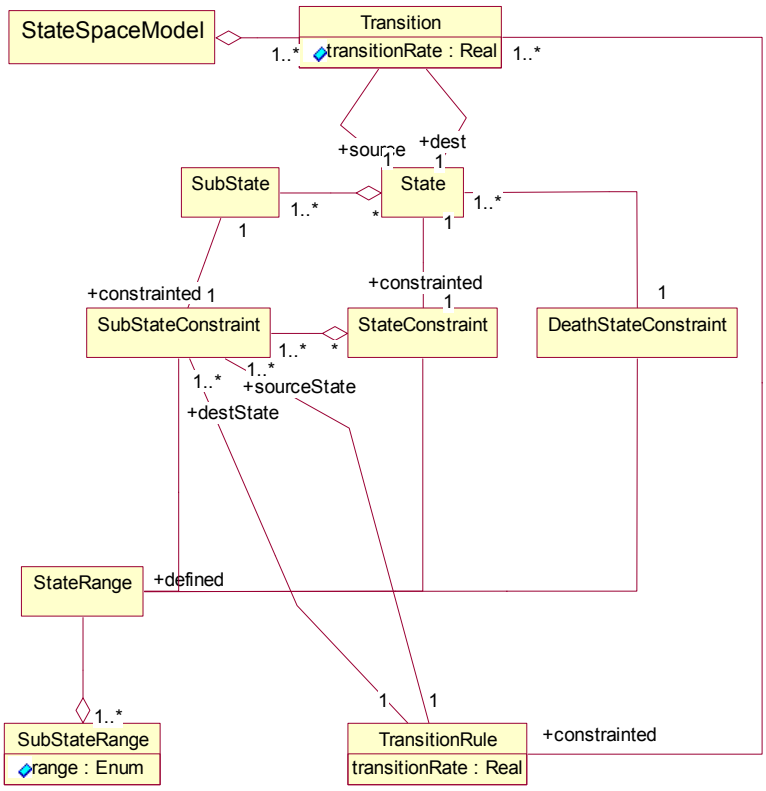

Figure 2-5: The general structure of a state space model.

The specification of large state-space models is often too complex and error-prone. The approach proposed in [Johnson 1988] alleviates this problem. In particular, instead of specifying all possible state transitions, the authors propose specifying the following: (i) the state range of the system, (ii) transition rules between sets of states of the system, (iii) the initial state of the system, and (iv) a death state constraint. In a transition rule, the source set and target set of states are each represented by constraints on the state range (e.g., if the system is in a state where more than 2 subsystems are operational, then the system may reach a state where the number of subsystems is reduced by one). A complete state space model can then be generated using the algorithm described in [Johnson 1988]. Briefly, the algorithm takes as input an initial system state. Then, the algorithm applies recursively the set of transition rules. During a recursive step, the algorithm produces a transition to a state

derived from the initial one. Depending on the rule that is applied, in the resulting state, one or more elements are modelled as being failed, or operational, while in the initial state they were modelled as being operational or failed, respectively. If the resulting state is a death state the recursion ends. Based on the above, in the remainder of this subsection, we detail how to exploit DSoS architectural descriptions to generate the information needed for the generation of a corresponding complete state space model

The first step towards our goal is to generate a state range definition for each collaboration belonging to a given service profile. The state of a collaboration is composed of the states of the component and connector instances used within the collaboration and the state of nodes on which the component instances execute. If a component is composite, its state is composed of the states of the constituent elements. The range of states for a component/connector/node depends on the kind of faults that may cause failures. In the case of permanent faults for instance, a component/connector/node may be either in an operational, or in a failed state. In the case of intermittent faults, a component/connector/node may be in an operational state, or it may be in a failed-active or in a failed-passive state. The range of states for a component further depends on the kind of redundancy used. If for instance a component represents a schema that tolerates 1 failure, then it may be in an operational state where no failures occurred, or in an operational state where a failure occurred and no additional failure can be tolerated, or in a failed state where 2 failures occurred. The post condition of the generation of state range definitions for collaborations is formally defined as follows.

Mapping Architectural Descriptions into State Space Models
Collaboration: self.associationRole->forall(connector | self.stateSpaceModel.stateRange.subStateRange->exists(str | str.name = connector.name) and connector.connection->forall(component | self.stateSpaceModel.stateRange.subStateRange->exists(str | str.name = component.name) and component.base.implementation.deployment->forall(Node | str.subStateRange->exists(str | str.name = node.name)) and component.contents->select(c | c.stereotype.oclIsKindOf(ADLComponent)or c.stereotype.oclIsKindOf(ADLConnector))->forall (el | str.subStateRange->exists(str | str.name = el.name))))

After generating the state range definition for a collaboration collab, the step that follows comprises the generation of transition rules for components/connectors/nodes used in the collaboration. Those rules depend on the kinds of faults of the corresponding architectural element. More specifically for permanent faults, the rules follow the pattern below:

Architectural Element	**Rule**

Component	• For all instances of primitive components, c:
	  • If collab is in a state where c is in an OPERATIONAL state st, then collab may reach a state st' where c is FAILED. The rate of those transitions is equal to the arrival rates of the faults that cause the failure of c, c.Faults.failure-rate.
	• For all instances of composite components, c:
	  • If collab is in a state st where c is OPERATIONAL, then collab may

	reach a state st' where c is FAILED due to a failure of a constituent element c'. The rate of those transitions is equal to the arrival rate of the faults that cause the failure of c', c'.Faults.failure-rate.
	• For all instances of composite components rc representing a redundancy schema of n components:
	  • If collab is in a state st where rc is OPERATIONAL, and the number of failed redundant component instances is fc, then collab may reach a state st' where the number of failed components of rc is fc+m. The difference between st and st' is m redundant component instances of the same type t, which in st were OPERATIONAL and in st' are FAILED. The rate of those transitions is equal to the fault arrival rate specified for t. This rule captures failure dependencies among redundant component instances of the same type. Those components are used in the same conditions and with the same input. Hence, if one of them fails due to a design fault, all of them will fail.
	  • If collab is in a state st where rc is OPERATIONAL and the number of failed nodes is fn, while the number of failed components is fc, then collab may reach a state st' where the number of failed nodes is fn+1 and the number of failed components is fc+m. The difference between st and st' is a failed node and m failed redundant component instances, which were deployed on the failed node. The rate of those transitions is equal to the fault arrival rate specified for the failed node.

Connector	• For all instances of primitive connectors see the case of primitive components. • For all instances of composite connectors, see the case of composite components.
Node	• We assume that nodes fail independently from each other. Hence, for all nodes in collab: • If collab is in a state st where a node n is in an OPERATIONAL state, then collab may reach a state st' where n is in a FAILED state. Moreover, in st', all instances of components c deployed on n are in a FAILED state. The rate of those transitions is equal to the arrival rate of the faults that caused the failure of n, n.Faults.failure-rate.

What is left at this point is to generate the definition of the initial state of the collaboration, and the definition of the death state constraint. The initial state is a state where all of the elements used in the collaboration are operational. A collaboration is in a death state if at least one of the architectural elements used within it is not operational. Hence, the death state constraint consists of the disjunction of base predicates, each one of which defines the death state constraint for an individual element used in the collaboration. More specifically, the base predicate for a component, connector, or a node states that the element is in a FAILED state. The base predicate for a redundancy schema is the disjunction of two predicates. The first one states that the number of failed redundant component instances is greater than the number of component faults that can be tolerated. Similarly, the second one states that the number of failed redundant nodes is greater than the number of node faults that can be tolerated.

2.4.4 Example

To demonstrate the automated performance and reliability analysis methods detailed in the previous subsection, we use the TA case study. The goal of our analysis is not to obtain precise values of the performance and reliability measures since this would require a precise model of the Internet. In general, such a model is considered unattainable [Floyd & Paxson 2001]. For that reason we concentrate on comparing different scenarios intended to improve the design of our system, while

assuming certain invariants proposed in the literature for modelling issues related to the Web [Floyd & Paxson 2001].

Scenario 1

Figure 2-6 gives a generic execution scenario where one or more customers use an instance, ta, of the TravelAgentFrontEnd to request the reservation of a flight ticket, a hotel room and a car. The ta component instance breaks down such a request into 3 separate requests. The first one relates to the flight ticket reservation and is sent to an instance, fr, of the FlightReservationComponent. The fr component instance uses this request to generate a new set of requests, each one of which is specific to an air company that collaborates with the TA system. The set of specific requests is finally sent to an instance, ac, of the AirCompanies composite component, which represents the current set of collaborating air companies. Similarly, the second and the third requests are related to the hotel and the car reservations, respectively. Those requests are sent to instances of the HotelReservation and CarReservation components, which reproduce them properly and send them to the current sets of collaborating hotels and car companies.

Performance Analysis

To analyse the performance of the TA, based on the scenario we detailed above, we have to specify first the performance parameters and stimuli that characterise the architectural elements used in the scenario. The ta, hr, fr and cr component instances are deployed on top of the same node, while component instances representing hotel, airline company and car company component systems are deployed on top of different sets of nodes. Moreover, the different components used in the scenario are multi-threaded and the work demands needed for performing their provided services are normally distributed. For the HTTP and RPC connectors, we assume that they are FIFO and that their capacity is limited. HTTP and RPC connector delays are normally distributed. Finally, we assume that requests to TA are initiated by customers according to the Poisson distribution.

Given the scenario in Figure 2-6 and the performance parameters and stimuli discussed above, a queuing network model can be generated based on the mapping detailed at the beginning of this section. In particular, a station that corresponds to the node on which the ta, hr, fr, and cr instances execute, is generated. The exact code for this station is given in Appendix A1.3.1.

Results from simulating the generated model using QNAP2 are given in Figure 2-9, and will be discussed later in comparison with results from a second scenario.

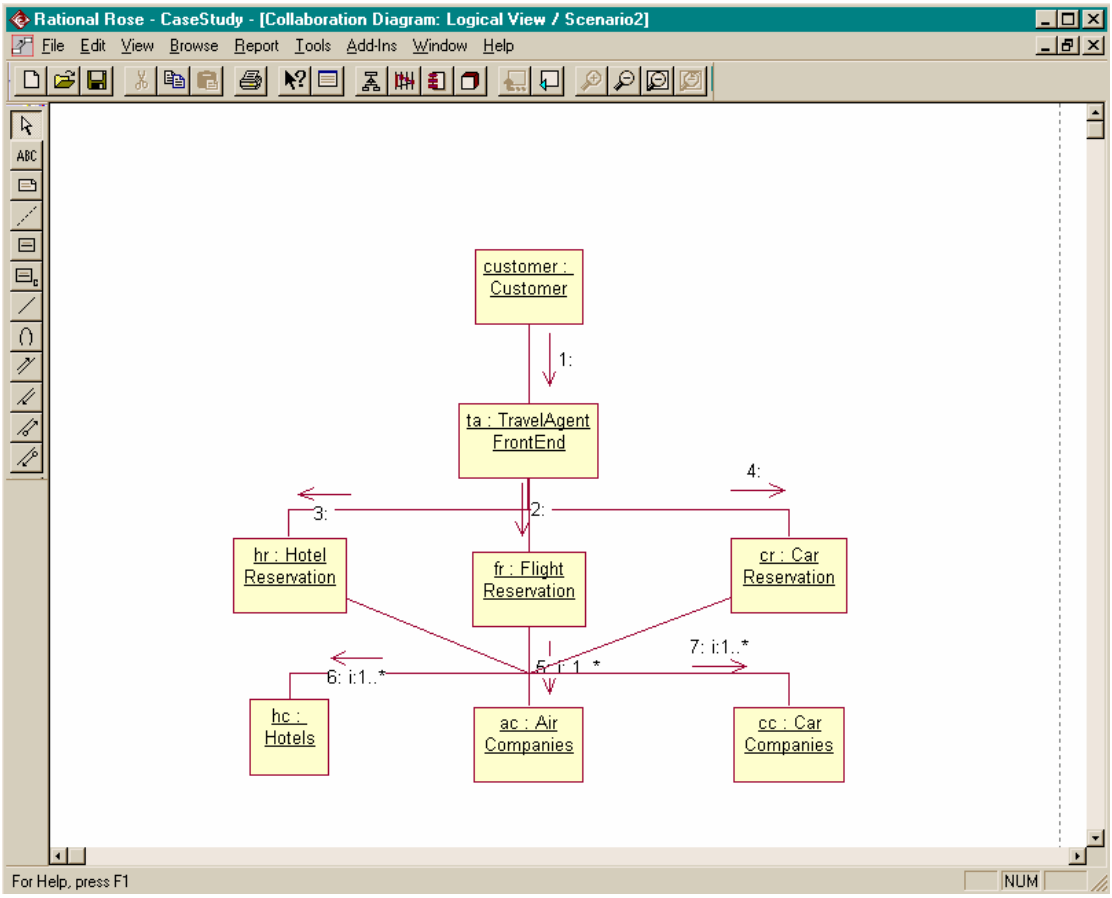

Figure 2-6: A generic execution scenario.

Reliability Analysis

As in the case of performance analysis, for reliability analysis we have to specify the reliability parameters and stimuli that characterise the architectural elements used in the scenario.

The component instances used in the scenario may fail to give answers to customers within a given timeout (Failures.domain = time). Component failures are manifestations of design faults (Faults.phase = design). We assume that those faults are accidental (Faults.cause=accidental), created by the component developers (Faults.causes=human). Moreover, component faults are all permanent (Faults.persistence=permanent) and their arrival rates vary depending on the types of the components. More specifically, the fault arrival rates for the components that represent component systems supporting hotels, air companies and car companies are much smaller compared to the fault arrival rates of the rest of the components that make up the TA system. The reason behind this is that the component systems supporting hotels, air companies and car companies have already been in use and their implementations are quite stable. On the other hand, the TA front-end and reservation components are still under development and hence their implementations are likely to be less reliable. Similarly, nodes may fail because of permanent faults and connector instances may fail because of transient faults. The arrival rates of node faults are much smaller than the arrival rates of component

ones. This holds similarly for the RPC connector. On the contrary, the HTTP connector is expected to be quite unreliable, with a failure rate greater than those of the components used in the TA.

By taking a closer look at the architecture of the TA system we can deduce that some sort of redundancy is used. In particular, the Hotels, AirCompanies and CarCompanies components are composite, consisting of n components that represent the information systems supporting hotels, air companies and car companies. The reservation components request from them, room, ticket and car reservations. For the scenario to be successful, we need answers from at least one hotel, one air company, and one car company. Hence, for the Hotels, AirCompanies, and CarCompanies composite components, the number of component and node faults that can be tolerated is n-1 (redundancy.no-of-component-faults = n-1 and redundancy.no-of-node-faults = n-1).

Based on our scenario and the parameters and stimuli given above, a complete state space model can be generated by following the method we discussed previously. The first step is to generate the range of all possible states of our scenario. The state range of our scenario is composed of the state ranges of the ta, hr, fr, and cr, component instances. Those component instances may fail due to permanent faults. Hence, they can be either OPERATIONAL, or FAILED. Moreover, the state range of our scenario is composed of the state ranges of the hc, ac, cc, redundancy schema instances. Each one of them can go through states (j, k):n*n, where j components and k nodes are FAILED, while n-j components and n-k nodes are OPERATIONAL. Moreover, for all states we have n-k >= n-j, since a node failure further causes the failure of the component instances that execute on top of it. Finally, the state range of our scenario is composed of the state range of the node, taNode, on which, the ta, hr, fr, and cr instances execute, and the state ranges of the RPC and HTTP connector instances[25]. Those elements can be either OPERATIONAL, or FAILED. The second step is to generate transition rules between sets of states of the scenario; Appendix 1.3.2 gives parts of those rules. The initial state for our scenario is a state where: ta, hr, fr, cr, and taNode are OPERATIONAL. Moreover, in the initial state, hc, fc, cc are in state (0, 0), i.e., all the components and nodes in the redundancy schemas are OPERATIONAL. In the initial state the instances of the RPC and HTTP connectors are OPERATIONAL. The death state constraint definition generated is the disjunction of predicates, which is given in Appendix 1.3.3. The predicates for the redundancy schemas state that a death state is a state where the number of failed elements is greater than n-1 (the number of faults that can be tolerated). For the rest of the elements, the predicates state that a death state is a state where the element is FAILED.

The generated model is used as input to the ASSIST [Johnson *et al.* 1988] tool, which generates a complete state space model for our scenario. Analytically solving the generated model using the SURE tool gives the results shown in Figure 2-8 (n=1). Those results will be discussed later in comparison with results from a second scenario, which follows.

[25] The HTTP connector instances represent the nodes and communication links involved in the interactions among the component instances they bind.

Scenario 2

The objective of the second scenario is to try to improve the reliability of the TA system. Our additional requirements are: (i) to keep the cost of the required changes in the TA system low, and (ii) to produce no negative side effects on the performance of the TA system. The basic means for improving the reliability of a system is to use some sort of redundancy. The question, therefore, is what sort of redundancy to use, and where. As previously discussed, the components that represent the component systems supporting the different available hotels, air companies and car companies are already organised into corresponding redundancy schemas. Hence, we are left with the option of using different, redundant, versions of the HotelReservation, FlightReservation, and CarReservation components.

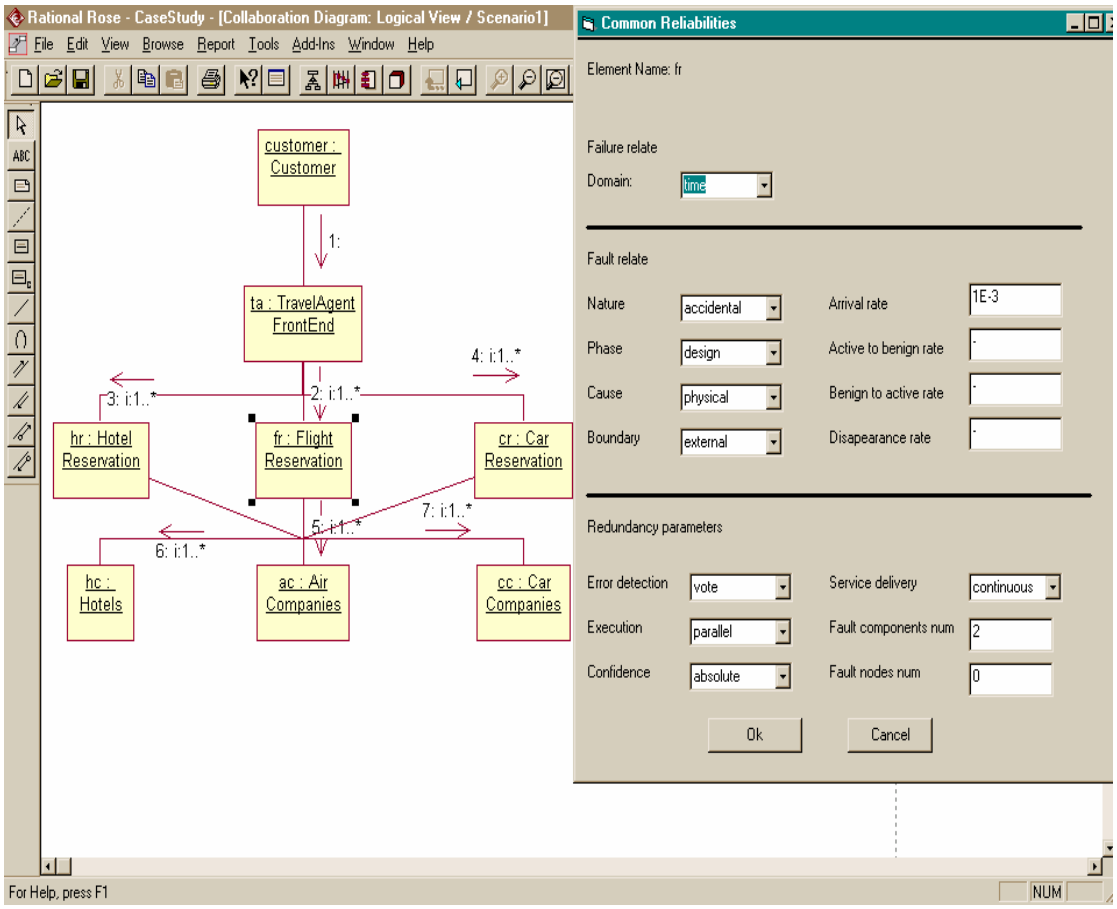

Figure 2-7: Using redundancy to improve the reliability of scenario 1.

Based on the above remarks, we designed three redundancy schemas. The first one contains n different versions of the HotelReservation component. Upon instantiation of the schema, n component instances are created, one of each version. Those instances execute in parallel and are deployed on n different nodes. The second schema contains n versions of the FlightReservation component, the instances of which are also deployed on the n nodes, on which the instances of the HotelReservation component execute. Finally, the last schema contains n versions of the

CarReservation component, the instances of which are also deployed on the nodes used to execute the instances of the HotelReservation component.

At runtime (see Figure 2-7), a customer request is broken down by the instance of the TravelAgentFrontEnd component into individual requests for flight ticket, hotel room and car reservation. Each one of those requests is replicated and sent to all the redundant instances of the corresponding reservation component. Each instance of the reservation component translates the request into specific requests for the corresponding available component systems and sends them. When the instance of the TravelAgentFrontEnd starts receiving offers for flight tickets, hotel rooms and cars, it removes replicates and combines them into answers that are returned to the customer.

We tried scenario 2 for n = 2 and n = 3. We generated the corresponding state space models and solved them using the SURE tool. The results are given in Figure 2-8, in comparison with results obtained from the first scenario (n=1). The main observation we make is that the reliability of the scenario does increase. However, the improvement when we use redundant versions is certainly not spectacular. The explanation for this is simple. In our scenario, the most unreliable element used is the HTTP connector. This is the main source causing the reliability measure to have small values. Any improvement in the rest of the architectural elements used will not solve this problem, which unfortunately cannot be easily alleviated.

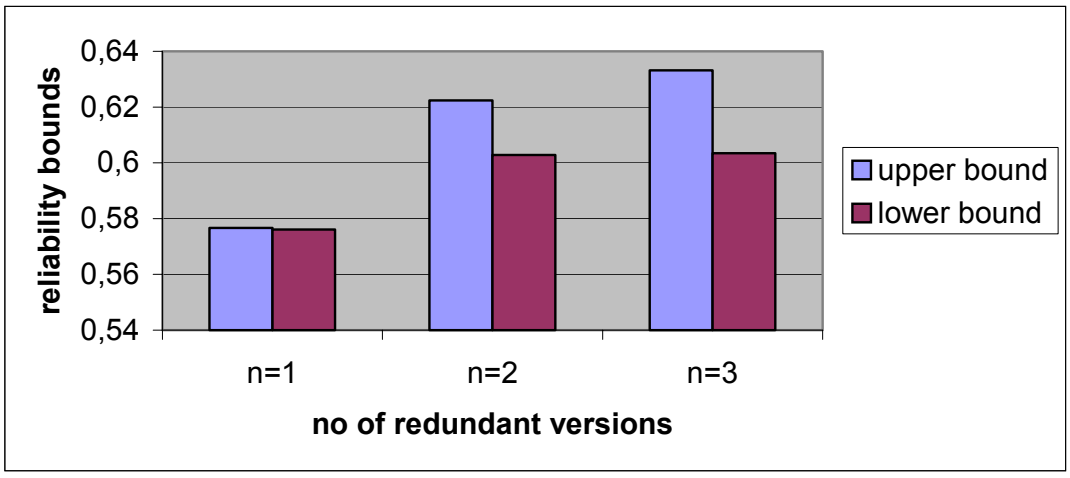

Figure 2-8: Results produced by the SURE tool

Hence, using multiple versions does not bring much gain. However, the good news is that the cost of using multiple versions is not prohibitive. The elements for which we produced multiple versions just translate TA specific requests into component systems' specific requests. Since the functionality of those components is quite simple, re-implementing them differently (e.g., using different developers) is not a complex task. Note here that the fact that the functionality of the redundant components is simple does not mean that there can be no bugs in their implementation. Actually, bugs in the mapping of TA requests into component systems' specific requests could potentially occur quite often. It is interesting at this point to take a look at results coming from the performance analysis. Simulating the model which results from the scenario given in Figure 2-7 gives us the results shown in Figure 2-9 (ta, hr, fr, cr, hc, fc, cc). Based on those results, component response times in scenario 1

(n=1) and scenario 2 (n=2,3) are close. This is due to the fact that the synchronisation among the different versions of scenario 2 is very simple (i.e., filtering of the results at the TravelAgentFrontEnd). However, it is clear that in scenario 2 the traffic in the components representing the different available component systems is significantly increased compared to scenario 1 (in Figure 2-7, for every customer request, the component systems receive n requests instead of one, which was the case in Figure 2-6). Hence, the execution time of the customers mainly depends on the ability of the DSoS components to handle such an increased traffic without problems. To demonstrate this, we simulated again scenarios 1 and 2 using additional sources of requests to the DSoS components besides the customers of the TA DSoS (Figure 2-9, ta+xtraffic, hr+xtraffic, fr+xtraffic, cr+xtraffic, hc+xtraffic, fc+xtraffic, cc+xtraffic). In this case, we see that the difference between the component response times in scenarios 2 and 1 is much larger.

.

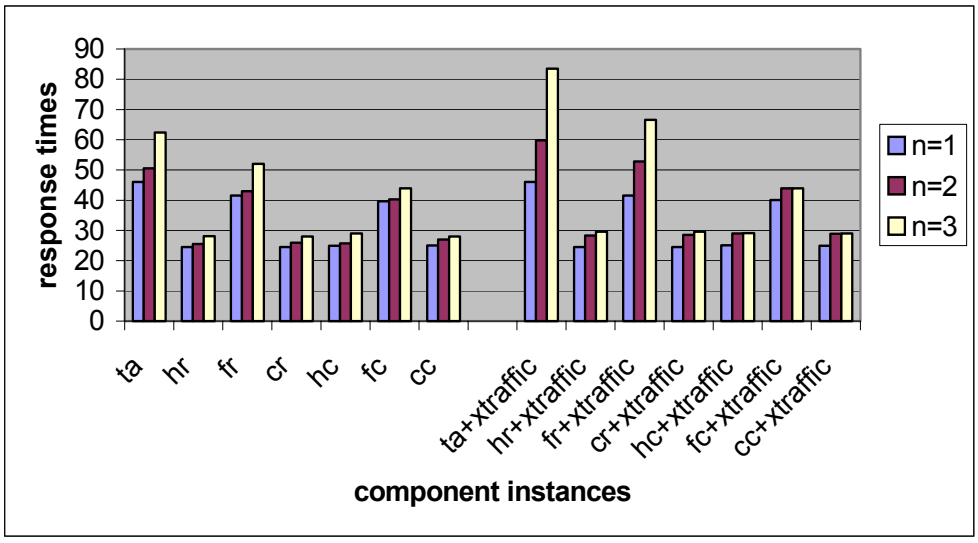

Figure 2-9: Results produced by the QNAP2 tool.

2.5 Summary

This chapter has presented a preliminary version of a developer-oriented, UML-based environment supporting architecture-based development of DSoSs, whose use has been illustrated using the Travel Agent case study [DMS3]. The current version of the environment actually focuses on base support for assisting the development of SoSs from the information system application domain. The environment offers a number of tools for assessing the SoS architecture at the various stages of the SoS development. These tools include the ones already offered by the UML environment on which ours build (i.e., the Rational Rose environment although our extension could be integrated in any environment processing XMI files). Our contribution comes from the definition of an ADL profile, defining an extensible, UML-based ADL, which may be specialised for the description of specific architectural styles as well as of complementary formal architectural models that may be thoroughly analysed. Regarding the latter issue, our preliminary work focuses on assisting the development of SoSs from the standpoint of the quality (also referred to as non-functional or extra-functional properties) they provide, while not requiring extensive expertise in formal methods from developers.

We have introduced specialisations of the ADL that enable analysing SoSs from both a qualitative and a quantitative standpoint. More specifically, the resulting architectural descriptions are automatically translated into system models that may be analysed by existing tools (i.e., SPIN model checker for qualitative analysis, QNAP2 for performance analysis and SURE-ASSIST for reliability analysis). We are building a prototype environment over the Rational Rose tool. Our prototype includes support for base architecture description. We are integrating the UML extensions aimed at qualitative and quantitative analysis together with the associated add-ins for the automatic generation of models processed by the underlying tools.

Up to this point, we have not addressed specific support for the development of **Dependable** SoSs, although the methods and associated tools for quality analysis that have been discussed in this chapter partly serve this purpose. However, as shown by complementary work done in this WP (see next chapters) as well as the other WPs [DMS1, IC1], specific solutions need to be offered regarding the architecting of Dependable SoSs and their validation. It is part of WP2 future work to integrate results offered in the proposed development environment. Our first step towards this objective will be to examine the architecting of DSoSs for the sake of dependability. A preliminary study in this area, which addresses exception handling at the architectural level [Issarny & Banâtre 2001], shows that novel architectural styles need to be elaborated for enhanced dependability to be achieved. Collaboration among the project's partners is therefore ongoing to devise such styles, as further discussed in Chapter 5.

Chapter 3 – Application-Specific Fault Tolerance Mechanisms

3.1 Introduction

The objective of this chapter is to discuss the initial results towards the development of fault tolerance mechanisms to be applied at the application level during the integration of complex systems of systems. These developments put in practice our conclusions presented in Chapters 2 and 3 of the DSoS State of the Art Survey [BC2]. Section 3.2 proposes a disciplined and systematic way of introducing component-level error detection and exception handling. In Section 3.3, we introduce an advanced Coordinated Atomic (CA) action scheme based on exception handling and present a distributed algorithm supporting its execution. This scheme makes it possible for system integrators to apply CA actions in situations typical of SoSs when it is impossible to impose tight synchronisation on action entry and exit required by conventional schemes. Section 3.4 presents our initial view on developing a general architecture to be used in constructing structured fault tolerant SoSs. In Section 3.5, we summarise the results achieved so far and discuss the topics of our future research, which is intended to build a comprehensive set of application-level fault tolerance mechanisms.

3.2 Exception Handling for Individual Application Component Systems[26]

In this section, we show how application-specific exception handling is to be systematically applied during SoS integration at the level of individual component systems. Complex SoSs are built of systems of two types: application-specific and general ones (a similar view is expressed in [Powell *et al.* 2000]). The examples of the latter are middleware services, OS, standard libraries. There are several differences in the way components of these two types are integrated, which stem from the fact that the behaviour of general components is better specified and understood, and they are usually employed by a number of programmers who share their experience so that they can avoid using erroneous services. Further, these components are often smaller, their behaviour is more predictable and, besides, more trust can be put in them compared with the application-specific (including newly-developed) component systems. General systems often have standardised specifications and are delivered with user's guides. Application systems, on the other hand, are usually more complex, less specified, less debugged; most of the time their linking interfaces have to be developed during SoS integration. These systems may not be developed as services to be used by other systems or as component systems to be integrated in a SoS. Their specifications are not fixed, they are more likely to evolve; ways in which they are employed in a SoS often change when integrators and users achieve a better understanding of their services and their functionalities. Application component systems are dealt with at the highest level of SoS integration as they serve as the building blocks for the SoS. These component systems are more likely to damage other component systems or/and to be misused by them.

---

[26] A more complete report of the results presented in Section 3.2 can be found in A. Romanovsky, "Exception Handling in Component-Based System Development", To be presented at the 15th Int. Computer Software and Application Conference, COMPSAC 2001, October 8-12, Illinois, USA.

Integrating application component systems means integrating and accommodating their normal and abnormal behaviour. There is a need to employ disciplined exception handling during SoS integration for many well-known reasons, but, in our opinion, developing SoSs needs more discipline and rigour in handling exceptions principally because component systems are usually very complex and SoS integrators have to deal with a wide variety of abnormalities without having sufficient knowledge about the internal structure or behaviour of component systems.

There is clearly a need for applying *enhanced application-specific error detection* at the level of individual component systems. As integrators are usually reluctant to put high trust in component systems or their specification, it is necessary to develop powerful error detection features and to employ some sort of defensive programming. Moreover, integrators usually do not have a complete specification of the component systems (both their normal and abnormal behaviour). It is a well-known fact that the exceptional behaviour of components is always under-specified or, even, not specified (see, for example, [Szyperski 1998], [Koopman & DeVale 2000]). In addition, we need error detection of this type because it is very likely that there are mistakes in component systems and their specifications, error detection inside these systems is not perfect and they are not used exactly in the contexts they were intended for.

Moreover, there is a need of *additional exception handling* that is local to each component system. It allows integrators to recognise damage and find out the reason for the detected error, to put the component system into a known consistent state, to try local error recovery and to deal with possible mismatches when component systems have different rules for informing the environment about exceptions raised or errors found.

The choice of the right approach to incorporating such local error detection and exception handling features is vital. Unfortunately, even though some component technologies offer structuring techniques for developing wrappers, there has not been enough attention paid to the problems of developing wrappers that provide general error detection and exception handling features suitable for dealing with application-specific component systems.

Local error detection and exception handling are performed at the level of the standard component system interface; they are "local" as they are developed for each component system and do not involve other components. Although we call it "local", we consider it to be a higher-level context than the internal context of the component system, within which their developers might handle or might try to handle exceptions before returning information through the component system interface.

3.2.1 Component Level Error Detection

Component-level error detection is performed by checking predicates and by catching all exceptions and error return codes. These predicates are to be developed in the course of wrapper development as an important part of SoS integration: they describe the correct or expected behaviour of the component system and are made executable. They include known restrictions in the way the component system is used in the integrated SoS. For example, if we build an Internet travel agency system using an existing flight reservation service, we might decide not to use APEX flight tickets at all as these cannot be cancelled. Another important restriction, which the SoS developer might decide to impose on a component system, is that it is only allowed to use the standard component system

interface: very often providers of a component system offer undocumented or non-standard functionalities but the integrators might decide against using them, to improve compatibility.

The wrapper protects the component system from misuse by checking that calls and input parameters are correct, that some parts of the interface are not used (this is application-specific) and that non-standard parts of the interface are not used, and by intercepting calls that can cause known component faults (this information is collected using fault injection, testing, bug reports). At the moment of output the wrapper checks correctness of outputs and, if possible, makes sure that the component system is in a known correct state.

To express what "correct state" means here, SoS integrators develop and formalise their views on the component system behaviour and specification. We totally agree with the general view on containing errors by means of wrapping expressed in [Voas & Miller 1997]: the authors believe that wrappers should limit what the components can do to the environment and what the environment can do to them. But we feel that this needs further development; this paper does not offer any detailed approach to developing such wrappers. Moreover, the only approach proposed relies on the use of results obtained by fault injection during implementation of wrappers.

There is clearly a need for a much more general and rigorous approach. Wrapper development is a complex engineering process, which has to be supported by precise techniques and by a clear description of the engineering steps to be undertaken. Ad hoc development is not acceptable here. Wrapper design incorporates the SoS integrator's view on what the component system can do, should do and should not do in the SoS. This development uses the existing (but often incomplete and unreliable) knowledge of the component functionality and the application-specific knowledge about the context in which the component system is to be incorporated (e.g., restrictions on the use depending on the application profiles). SoS integrators design contracts between the environment and the component system in the form of predicates on component inputs and outputs, and, possibly, on the component state (when it is assessable through the standard interface) used for detecting latent errors. These predicates are incorporated into the component system wrapper in the form of executable assertions. From our point of view, it is crucial to try to develop a detailed specification of the correct component system behaviour but to keep it reasonably small, to allow for cost-effective run-time checking. It is clear that the majority of SoS integrators will be willing to accept some run-time overheads if they use a ready-made system in an integrated SoS.

3.2.2 Component Level Exception Handling

Local exception handling starts when either the component system signals an exception (or, an error return code) or an assertion detects a violation of the specified system or environment behaviour. This handling can include another attempt to provide the required service, search for more information about the exception and the reasons for it, checks of the state in which the component system has been left, its recovery or operations putting it into a correct known state. The wrapped component system should have a set of interface exceptions, which it can signal in such a way that it can give guarantees (or attempts to give them) that the component system is left in a state that corresponds to the exception being signalled. It is vital never to leave the component system in a state not known to be consistent but experience shows that this is not always achieved (see, for example, [Salles *et al.* 1997]). It is the responsibility of the wrapper to check that this has been done properly and, if not, to execute appropriate operations. Guaranteeing the "nothing" semantics is the most useful approach; a

number of the interface exceptions can have such semantics (e.g., Service_Cannot_Be_Used or Illegal_Input_Parameters). It is important to introduce a special *failure* interface exception to be used when the state in which the component system has been left is unknown, to advise the SoS not to employ it without appropriate recovery.

By bracketing each operation on the component system with the software performing this additional functionality the wrapper turns it into a well-defined building block, which SoS integrators can use. The interface exceptions are a very important part of the wrapped component system: the wrapper informs the higher SoS level about abnormal behaviour (providing additional information about the state of the component to allow for compensation at the higher level) and passes the responsibility for recovery to the higher level if local recovery is not possible.

Integration of a complex SoS requires additional activity for developing a unified exception handling policy at the SoS level. *Mismatches* are possible between different exception handling models [Hansen & Fredholm 2001]. For example, in COM, all interface methods should return a status (HRESULT) indicating success or exception/failure of the method execution, which is quite different from catch and throw in Java or C++. SoS integrators should define such a policy and each wrapper should follow it when signalling exceptions.

Another possible way of handling exceptions at the wrapper level is to signal an interface exception and initiate an off-line recovery (e.g., involving operators). This requires a special functionality, which the wrapper can provide: delaying all requests until the component is repaired or replaced.

Local exception handling:

- Incorporates damage assessment (e.g., by calling component methods and analysing information about the detected error).
- Tries to handle an exception locally through the standard interface of the component system.
- Signals an exception to the environment without executing the requested function (if the request is erroneous).
- Uses a unified way of signalling exceptions augmenting the exceptional outcomes with additional information (e.g., component system name, function name, name of the illegal parameter, etc.).

These are some of the ways to handle exceptions locally: re-try the operation; send a message to the operator; redirect the message to an alternative destination; perform a simpler version of the component function; compensate (to guarantee the “nothing” semantics); perform damage assessment; put the component system into a consistent known state (e.g., using standard abort, initialise interface operations); replace a failed component system with a new one.

3.2.3 Discussion

The approach proposed makes error recovery cheaper because it promotes early error detection by executing assertions each time the component system is called, local exception handling, unification of exception propagation to the SoS level, leaving component systems in a known consistent state when an exception is propagated. To conclude this section, we would like to stress again that we

believe that errors should ideally be detected at the level of component systems by wrappers, and when they are detected, an attempt should be made to handle them locally. If this is not possible, an exception should be signalled to the environment in such a way that the component system is left in a known and consistent state to facilitate the subsequent handling at a higher level.

Component-level error detection and exception handling (implemented in component wrappers) makes each individual request addressed to a component system behave as an atomic operation with multiple outcomes. This facilitates SoS integration, allows SoS developers to build a recursive SoS structure and to employ SoS-level fault tolerance using this structure. Besides, this approach promotes protection of each component system against errors outside it, as well as of the rest of the SoS against errors inside it.

We believe that it is important at this stage of the Project to be general and to understand the main underlying principles behind building fault tolerant SoSs. The ideas presented in this section are preliminary, and it is clear for us that more work has to be done to develop concrete practical solutions. Our future research will focus on incorporating local error detection and exception handling into SoS integration process and, more generally, into the whole life cycle of SoS development. Wrappers performing such functionalities have to be introduced at the earlier phases of the life cycle, starting from developing SoS architecture. There is a need for introducing rigorous ways for specifying, analysing and implementing wrappers performing local error detection and exception handling for individual application component systems. To progress in this direction, we have to clearly understand the whole SoS development process, typical characteristics of SoSs from different application domains, assumptions characteristic of SoSs of different types, characteristics typical of component systems to be used in SoSs (including information about the systems that SoS integrators have at their disposal, the “colour” of boxes, system fault assumptions). At this stage, we do not consider the development of advanced ways of wrapping (see Section 3.2.2 in [BC2] on how to wrap components) to be an urgent matter since all standard component technologies provide features for doing this (e.g., adapters, proxies, filters, interceptors, etc.) and, as our survey in [BC2] suggests, there are a number of well-developed techniques for doing this. However, there is a need for a disciplined way of generating wrappers from the specification of the intended behaviour of the component systems. One possible solution to this issue is presented in the next chapter, which introduces a generic wrapping framework together with one of its specialisations based on specifications in temporal logic.

3.3 Advanced Atomic Actions Based on Exception Handling

The general application-specific fault tolerance approach that is being developed makes use of the strongest traits of Davies' spheres of control (as the conceptual framework) [Davies 1979] and CA actions [Xu *et al.* 1995]. As is pointed out in [BC2], CA actions serve as a solid foundation in developing structuring techniques suitable for complex SoSs and for achieving their fault tolerance in a disciplined and structured way, due to their ability to:

- Allow system developers to design, structure and provide fault tolerance in complex SoSs in which component systems cooperate and compete.
- Provide support for exception handling, which is vital for component systems 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 full support for maintaining consistency and achieving fault tolerance.
- Allow tolerance of environmental faults, software design faults and crashes of component systems with transactional behaviour.

In addition to these advantages, we have identified that CA actions allow us to address typical problems of SoS integration better than conventional workflow systems [Romanovsky 2001a] because they:

- Are built on a much richer concept of atomicity than traditional workflow systems and allow for a more general way of achieving fault tolerance.
- Offer a powerful solution for building long-lived activities.
- Can serve as a general approach to building SoSs of different types and in different application domains (including real time, the Internet, etc.).
- Are equally applicable at different phases of SoS integration.
- Allow integrators to reason about complex SoSs in terms of participating component systems rather than activities (for example, this makes it easier for them to express recursive SoS structure).
- Are more suitable for the existing standard component technologies (such as CORBA, EJB, and DCOM) as they support system development in terms of components and processes rather than activities.
- Allow the same paradigm to be used at all levels of system development and integration.
- Are more suitable for the conventional paradigms used in system development (including those supported by process-oriented concurrent languages, such as Java and Ada, and by process-oriented formalisms, such as CSP and CCS).

At the same time, we realise that, in order to make CA actions more suitable for building complex SoSs, they need to be enriched by features that are more typical for workflows, such as dealing with activities involving organisations and people. The defining characteristics of component systems that the CA schemes have to deal with are as follows: component systems may be unaware of their participation in a SoS, and due to their nature they may not allow any additional synchronisation to be imposed on their execution from outside.

The main result reported in this section consists of the development of an advanced CA action scheme for incorporating autonomous component systems that are not willing to be, or cannot be, tightly synchronised with other component systems during SoS execution.

3.3.1 Look Ahead CA actions

Decreasing the level of additional synchronisation can be beneficial in many areas: complex systems of systems; distributed applications; systems involving people, organisations, external devices, documents, etc. In fact, in some applications, CA actions imposing tight entry and exit

synchronisation cannot be used at all as they slow down the normal execution of all participants. It would be very advantageous, therefore, to let individual participants leave an action without synchronisation. This would necessitate knowing how to deal with the situation when an exception is raised in an action from which some participants have exited. Participants leaving an action without waiting for all participants at the action exit are *looking ahead*, the term introduced in [Kim & Yang 1989] for the conversation scheme [Randell 1976] (here recovery is based on rolling processes back to recovery points set at the conversation entry).

Our suggestion is to reinterpret the mechanism of looking ahead in the context of CA actions in the following way. If a participant reaches the end of an action and is not aware of any exception inside it, it leaves the action and continues its execution (see Figure 3-1). If an exception is raised by a component system in an action and this action has a participant that has looked ahead from it, then it is clearly not possible to handle this exception at the level of the action. Our idea is to employ the CA action structure, which we have in place, to find a containing CA action that contains all possible erroneous information and to perform cooperative recovery at its level. To do this, we need all component systems to be in this action to provide cooperative error recovery and to guarantee the absence of information smuggling.

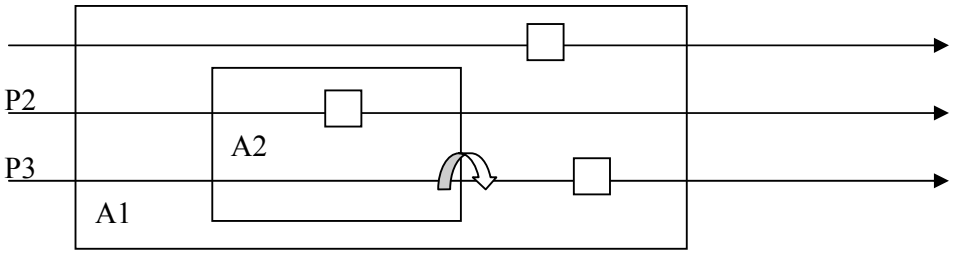

Figure 3-1: Component system P3 has looked ahead from CA action A2. Small squares show the current execution states of the component systems[27]

There are two approaches to developing CA action schemes without entry and exit synchronisation. In the first approach, each participant multicasts service messages to all action participants on entering and successfully exiting an action to allow them to keep locally information about the state of all action participants. Using this information, they can find look ahead component systems and the action that has to be recovered if an exception is raised. Alternatively, a component system can keep its own history and continue its execution without multicasting any service messages. In either case, when a component system raises an exception it multicasts a service message to action participants, and waits until the resolved exception and the action to be recovered are found (it may not necessarily be the actual action the component system is in now, because a concurrent exception can be concurrently raised in a containing action). In our scheme, we exploit the second scenario, as it involves no additional service messages unless exceptions are raised. This approach agrees with the main requirement for fault tolerance features, which is to keep additional overheads low (or even avoid them when possible) if there are no faults in the system.

[27] This way of representing the SoS state allows us to illustrate past, present and the future of a component system in an action. The current SoS state is defined by the current states of all component systems.

Lookahead (LA) component systems. A LA system cannot be involved in recovery at the level of the action that it left as it may have been involved in other actions since then, which may have caused it to lose the action context. It might have smuggled erroneous information outside; moreover, the underlying idea of the atomic action scheme is that we should assume, defensively, that erroneous information has been propagated. Our approach is to find a containing action that includes all such LA systems and to involve all of its participants in cooperative handling to guarantee the consistency of recovery. Let us consider, for example, the SoS shown in Figure 3-2. It is not possible to perform recovery at the level of action A1 in which component system Pk has raised exception En because Pi is not in Al and because Pi has smuggled erroneous information to action A. We have to handle this situation at the level of A.

Predefined LA_system exception. An internal exception raised in a nested action cannot be seen at the level of a containing action. This is why we assume that each action has a predefined interface exception LA_system. This exception is used to inform an action of the fact that its nested action has an internal exception but cooperative handling at the level of this action is not possible because of a LA system. Note that when this exception is raised, it is assumed that the nested action is left in an inconsistent state because not all participants were in it when an exception was raised, so not all of them can ensure their results and some of them could have smuggled erroneous results outside.

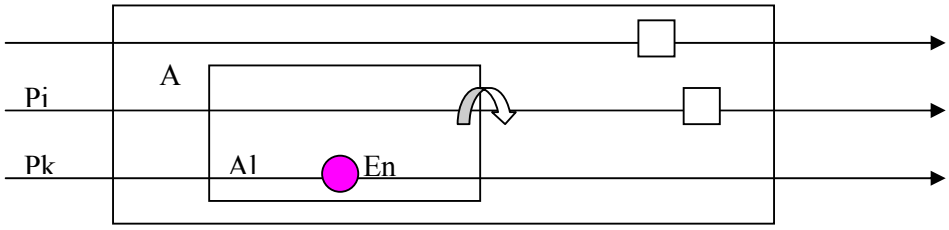

Figure 3-2: Component system Pi is a LA system for action A1. The small shadowed circle shows an exception raised in a system (in this case, En in Pk)

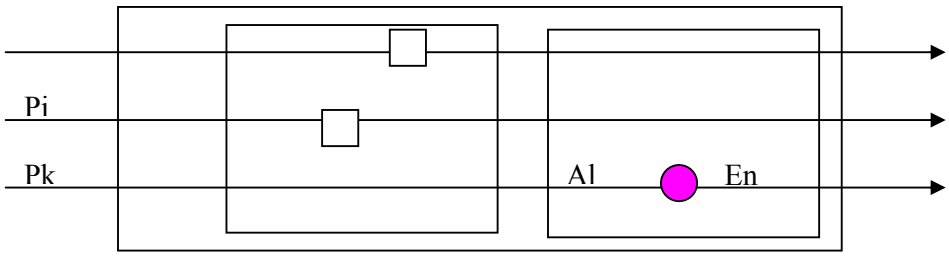

Figure 3-3: Component system Pi is belated for CA action A1

Belated systems. If component systems are not synchronised at the action entry, it is possible that some of them intend to enter the action but are outside it when an exception is raised in it. We call such systems belated systems (see Figure 3-3). In our scheme, as well as in some other schemes [Xu *et al.* 2000], the support waits for belated systems, so as to involve them in cooperative handling if an exception is raised. An alternative solution would be to assume that the fact of a component system being late indicates an error and to find an action of a higher level in which the belated participant is currently involved to initiate recovery at its level. We deliberately separate the issues of error detection and systems being late, and consider that each system detects its errors independently and that it is not the responsibility of our scheme to detect such errors. We assume that each action

participant watchdogs (using timeouts) its participation in actions, so each system will eventually either enter each action it intends to enter or raise an exception.

Nested actions. Dealing with nested actions when an exception is raised in a containing one requires special care. There are two ways of involving action participants in action recovery: asynchronous and synchronous schemes [Mitchell *et al*. 1998]. In asynchronous schemes, action participants are interrupted and because of this all nested actions have to be aborted. Synchronous schemes assume that all participants complete their execution in an action (either by raising exceptions or normally). Developing asynchronous schemes is more challenging, but they provide faster recovery and allow complex situations involving belated systems to be dealt with. Figure 3-4 gives an example of the latter: action Aj has to be aborted to allow cooperative handling at the level of A1. The nested action abort is one of the functionalities of our scheme: when such an action is aborted, all participants that are currently in it are interrupted and asked to execute a special local abortion handler [Xu *et al.* 2000]. We found this approach very useful in our previous study and, although it does not allow for a complete action abort, in many practical situations action participants can perform very effective cleanup and finalising activities to make the subsequent recovery at the higher level simpler. Employing this kind of nested action abort can facilitate the interactions between components [Xu *et al.* 2000].

Minimum containing actions. Our protocol has to deal with situations when several exceptions are raised concurrently in different actions [Xu *et al.* 2000]. Let us consider a SoS with two concurrent exceptions shown in Figure 3-5. Note that exception Em can be caused by erroneous information that a LA system Pk passed from action Al to Aj. We need recovery at the level of A (we call such action a *minimum containing* action for actions Al and Aj): this recovery should start with aborting actions Al and Aj and propagating a LA_system exception to A.

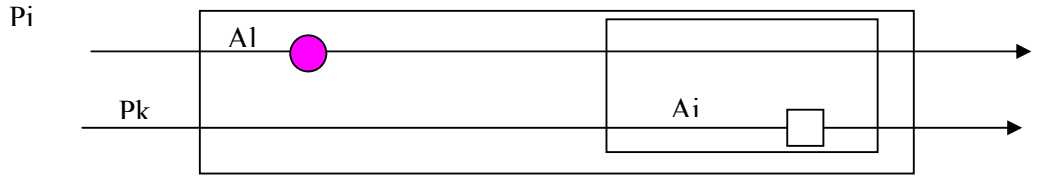

Figure 3-4: Nested CA action Aj has to be aborted

Our approach can be summarised in the following way: we wait for belated systems, interrupt nested actions and chase the LA systems to recover the action they are in.

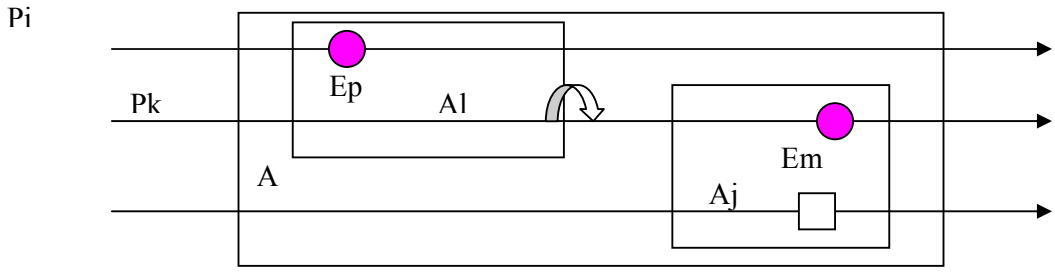

Figure 3-5: Two concurrent exceptions. Action A is the minimum containing action for Al and Aj

Our LA CA action scheme provides resolution of the concurrent exceptions raised within an action ([Campbell & Randell 1986], [Xu *et al.* 2000]) and our protocol incorporates this functionality. Generally speaking, our intention is to keep all benefits of the distributed CA action scheme in [Xu *et al.* 2000] but allow for asynchronous action exit. In particular, we use the same approach to the resolution of concurrent exceptions in an action after all participants have stopped. In this situation, each participant knows the states of all component systems and the system with the "highest" name among all component systems that have raised exceptions resolves the exceptions using the action resolution graph and multicasts it to all action participants to initiate cooperative exception handling.

3.3.2 Distributed Protocol for Exception Handling with Looking Ahead

The protocol works in the following way. Each participant executes action entry/exit operations without any synchronisation with other participants and keeps the whole history of its participation in actions. To raise an exception, it sends a service message with its history attached to all participants of the (current) active action. After receiving this message, each component system analyses its own history and the sender's history to find out whether it has looked ahead from the action with the exception or not. In the former case, it finds the minimum action containing all actions that have to be aborted, aborts all these actions and signals a LA_system exception at the level of this containing action. In the latter case it saves the message and continues since it is a belated system for the action with the exception. When all component systems of an action with an exception or, possibly, with several concurrent exceptions, have been informed about it, one of them resolves concurrent exceptions and triggers cooperative handling.

We assume that the SoS consists of component systems P1, …, Pd. Each action Al is defined by a set of participating systems, a set of internal exceptions, and a set of external ones. Each action participant has a set of handlers, one for each internal exception. The cooperative exception handling of an internal exception consists of all action participants executing their corresponding handlers. We assume that handlers cannot raise internal exceptions. Any action participant can signal an external exception to be propagated to the containing action. To distinguish between these two types of exceptions, and following [Xu *et al.* 2000], we say that internal exceptions are *raised* and that external exception are *signalled* or *propagated*. Component systems enter action Al by executing operation Enter(Al) and leave it either by signalling an external exception or successfully (maybe, after successful cooperative handling of an internal action exception) by executing Exit(Al).

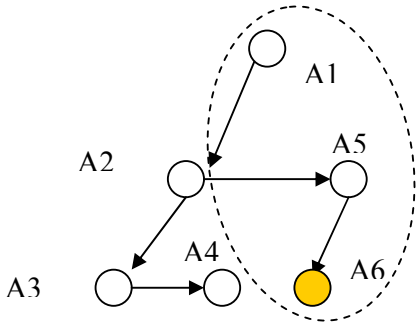

Figure 3-6: Action history Hi for component system Pi. The shadowed node shows *the active action*. The ellipse covers CHi

We shall start with introducing two concepts that are important for the protocol. The *action history* Hi of a system Pi is represented as a binary tree with nodes corresponding to actions. For each node A, the left child is the first action nested in A which Pi entered while in A, and the right child is the first action sibling to A which Pi entered after exiting A. For example, Figure 3-6 shows history Hi of component system Pi. Pi is now in action A6. It has completed its participation in A2, A3 and A4. But its participation in A1, A5 and A6 is not completed: A6 is nested in A5 and A1, and A5 is nested in A1. We call the list of nested non-completed actions *compressed history* (CHi). In our example, CHi=(A1, A5, A6). This is the list of all actions the system is in at the moment. Pi dynamically updates Hi and CHi when it enters and exits actions.

The protocol uses service messages of the following types:

- Exception(Al, Pi, Ep, CHi) is sent by system Pi from active action Al to all of its participants when exception Ep is raised in Pi; the compressed history for Pi is sent as part of the message;
- Suspended(Al, Pi, CHi) is sent by system Pi when it receives Exception or Suspended message from any other participant of Al and stops;
- Commit(Al, Eres) - is sent by a chosen system in action Al to all of its participants after it completes resolution of concurrent exceptions raised in Al, where Eres is the resolved exception (one of the internal exceptions of Al). The corresponding handler for Eres is called by each system once it receives this message.

Each system Pi keeps:

- Hi and CHi;
- CAPi (*the current active action pointer* in Hi) is used because there is a need to keep complete history Hi as it was before a number of actions from CHi have been aborted. In our protocol, nested action abortion is local and some participants of the aborted actions may still be active, so any messages sent by them have to be identified and ignored. For example, in the scenario shown in Figure 3-7 we assume that Pi has aborted its participation in A4 and is currently in A2. When Pi receives Exception(A3, Pk, Em, CHk), it finds out that it has been in A3 before and aborted it, so Pi ignores the message (eventually Pk aborts A3, and A2 becomes the active action for it).
- LBi - a set of Exception and Suspended messages from the actions for which Pi is belated.

Each system can be in N (normal), E (exceptional) or S (suspended) state in its active action. A system goes from state N to state E when it raises an exception; it goes into state S when it receives an Exception or Suspended message within the active action. Any system stops executing the application code when it goes into either state S or state E. In addition, each system keeps list LEi to record all exceptions raised in the active action and the known states (either S or E) of all component systems that have stopped in the action.

Pi signals an external exception Ep to the containing action by performing two steps, one after another: (i) exiting the current action and (ii) raising Ep in the context of the containing action. The first step includes modifying Hi, CHi, CAPi.

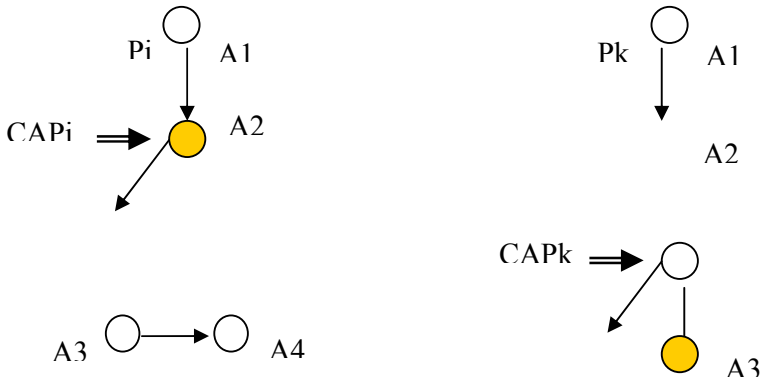

Figure 3-7: The current action pointer; ignoring messages from aborted actions

We assume that each action has a unique name, that all systems have unique names that can be ordered, and that each system knows the lists of all participants of all actions it is taking part in. We further assume that each system has the resolution graphs of all actions it is taking part in.

As we have explained before, in order to deal with an exception at the level of the containing action, we have to abort all active nested actions (from CHi). This should include the abortion of all possible exception resolution protocols and exception handlers. We assume that each action participant has a handler that performs local abortion. The handlers of different participants are not expected to cooperate during abortion, the idea being that each of them is responsible for local abortion (e.g., cleaning up, finalisation) only. If a chain of nested actions for a given system has to be aborted, our support executes the abortion handlers of these actions one by one, starting from the active action.

We assume that an underlying mechanism guarantees FIFO reliable communication between any two component systems; "->" stands for sending message to all participants of the action; "S(Pi)" stands for the state of Pi; "=>" stands for adding information into the list; "message" stands for the current message received by Pi; "abort(Al, A)" stands for aborting all nested actions in CHi starting from Al up to the action nested to A. For any component system Pi, the protocol looks as follows (Al is the active action for Pi):

Protocol for Exception Handling with Looking Ahead

```
loop
if Pi raises Ep then
    Exception(Al, Pi, Ep, CHi) ->; stop; Ep=>LEi; S(Pi):=E end if;

if Pi executes Enter(Al) then
    update Hi, CHi, CAPi;
    read and process messages in LBi that have been sent within Al end if;

if S(Pi) = N and message=(Exception(Aj, Pk, Em, CHk) or Suspended(Aj, Pk, CHk)) then
    stop;
    if Aj=Al then
        Suspended(Al, Pi, CHi)->; Si=>LEi; S(Pi):=S end if;
```

```
    if Al<>Aj then                                                    -- analyse Hi
        if Pi has never been in Aj before then
          message=>LBi; continue end if;                  -- belated system

        if Al is nested in Aj then
          abort (Al, Aj); clean LBi from messages related to all actions in between;
          move CAPi to Aj;                                            -- now Al=Aj
          Si,=>LEi; S(Pi):=S;
          if message=Exception(Aj, Pk, Em, CHk) then Em=>LEi else Sm=>LEi end if; end if;

        if Pi has been in Aj before and has looked ahead from it then
          find Acont for Aj and Al;                                   -- using Hi and CHk
          abort (Al, Acont); clean LBi from messages related to all actions in between;
          move CAPi to Acont;                                         -- now Al=Acont
          Exception(Acont, Pi, LA_system, CHi); Ei=>LEi; S(Pi):=E;    -- Ei is LA_system
          if message=Exception(Aj, Pk, Em, CHk) then Em=>LEi else Sm=>LEi end if; end if;
    end if;
end if;

if S(Pi)<>N and message=(Exception(Aj, Pk, Em, CHk) or Suspended(Aj, Pk, CHk)) then
    if Al=Aj then
        Ej or Sj => LEi;
        if LEi contains states of all systems in Al and S(Pi)=E and Pi is the highest in LEi
        then resolve exceptions from LEi; Commit(Al, Eres)-> end if;
    end if;

    if Al<>Aj then
        if Pi has never been in Aj before then                  -- belated system for Aj
            message=>LBi end if;                                -- for future consumption

        if Pi has been in Aj before and has looked ahead from it then
            find Acont for Aj and Al;
            if Acont=Al then
              ignore message                            -- Pi looked ahead from several actions
            else
              abort(Al, Acont); clean LBi from messages related to the actions in between;
              move CAPi to Acont;                               -- now Al=Acont
              Exception(Acont, Pi, LA_system, CHi)->; Ei=>LEi; -- exception Ei is LA_system
              S(Pi):=E end if;
            end if;

        if Pi was in Aj before but has not left it then         -- Aj is containing for Al
            abort (Al, Aj); clean LBi from messages related to all actions in between;
            move CAPi to Aj;                                    -- now Al=Aj
            Suspended(Aj, Pi, CHi)->; Si=>LEi; S(Pi):=S end if;

        if Pi has been in Aj before but its CAPi is higher than Aj then
            ignore message end if;                              -- from an aborted action
    end if;

if Pi executes Exit(Al) then
```

```
    update Hi, CAPi, CHi end if;

if message=Commit(Aj, Eres) then                          -- S(Pi)<>N
    if Aj=Al then
        discard branches of Hi below CAPi; start handler Eres
    else ignore message end if; end if;                   -- from an aborted action

end loop;
```

Let us consider a simple example to demonstrate how the algorithm works. For the SoS in Figure 3-2, the following scenario is possible: Pk sends Exception(Al, Pk, En, CHk) to Pi and stops; component system Pi receives it and realises that it is a LA system for Al; it finds Acont (it is action A) and raises exception LA_system at the level of A; Pk receives Exception(A, Pi, LA_system, CHi), aborts its participation in Al and moves into state S in action A. Pk is the only component system with exceptions in A, so it sends Commit(A, LA_system) to all action participants and they start cooperative recovery.

3.3.3 Discussion

The correctness of the algorithm presented here can be shown by reducing it to the algorithm presented in [Xu et al., 2000], the correctness of which was formally proved. Comparing with this algorithm our protocol adds only two steps to be taken into account while proving it: (i) searching the action that contains all LA systems known to the system (this step is executed locally), and (ii) raising an LA_system exception in this action. It is not difficult to see that these steps cannot cause deadlocks even if there are several systems executing them at the same time, because (i) they can add only a finite number of additional messages that could be sent (since the number of the component systems and the level of nesting in any SoS are always finite), and (ii) if there are no more exceptions raised, these steps always complete within a restricted period of time. A further analysis, together with more examples and a discussion of such important issues as the benefits of the protocol proposed, its applicability and complexity, can be found in [Romanovsky 2001b].

The results reported in this section constitute the first step in developing structuring techniques for building dependable SoSs. In our future work, we plan to extend the scheme with an ability to deal with component systems of passive nature (including some types of service providers, transactional objects, data, databases, etc.), to do experimental work, and to design a case study using the scheme proposed. Our analysis of the typical characteristics of SoSs shows that there is a need for very flexible structuring techniques. The look ahead scheme presented above supports such flexibility. But, at the same time, we realise that there are different ways in which SoSs, and the structuring techniques used, need to be flexible. Some of the examples of such flexibility that are not supported by conventional CA actions are: allowing outside component systems or actions to access intermediate action states or non-committed results, allowing action participants to fork new participants, allowing action participants to invite other component systems into an action when necessary, allowing flexible dynamic action membership. Our preliminary analysis shows that some of these can be achieved by reducing them to the problem of a looking ahead participant. But, there is still a need for further generalisation of the CA action scheme to make it possible for the SoS integrators to apply action structuring and disciplined exception handling to SoSs with different characteristics.

3.4 General Integration Framework

SoS integrators face two problems related to dealing with abnormal events: developing exception handling at the level of the integrated SoS and accommodating (and adjusting, if necessary) exceptions and exception handling provided by individual component systems. Our ultimate goal is to develop a general exception handling and structuring framework to be applied during SoS development. This framework is to be applied in three main steps, following the initial results reported in this chapter and in [BC2]. Firstly, individual component systems are wrapped in such a way that the wrappers perform activities related to local error detection and exception handling, turning each individual request to a component system into an atomic operation with multiple erroneous outcomes associated with different external exceptions. The wrappers signal, if necessary, exceptions outside the component system to a higher level of SoS structuring. It would be clearly wrong and error-prone to view a SoS as a flat set of all integrated component systems and to leave it with the integrator to decide which of them to involve in handling each abnormal situation because SoSs usually have a much more complex architecture than the client/server one. This is why integrators need general techniques applicable to structuring SoSs of any complexity to support SoS-level error containment and exception handling. At the second step of the framework, the overall SoS is structured as a set of CA actions in which component systems take part. Such actions have important properties, which facilitate exception handling: they are atomic, contain erroneous information and serve as recovery regions; also, these actions can be nested so that SoSs can be developed recursively. At the last step, action-level exception handling is designed to allow each action (i.e., all component systems participating in it) to handle all exceptions signalled by both the wrapped component systems and all nested actions.

By now, we have identified a number of functionalities to be provided at the application level to allow development of fault tolerant SoSs. They include support for:

- Turning an individual (e.g., autonomous) system into a component system to be integrated into a SoS.
- Performing systematic and disciplined local error detection and exception handling at the level of component systems.
- Turning a component system into a participant of CA actions taking part in all action-specific activities such as cooperative exception handling (including resolution of concurrent exceptions at the action level), action entry and exit synchronisation (when necessary), controlling erroneous information smuggling, guaranteeing proper action nesting, assuring consistent access to component systems and any other resources involved in SoSs, etc. [Randell *et al.* 1997].
- Structuring the whole SoS recursively as a number of CA actions with component systems taking part in these actions.
- Turning each component system into either an active CA action participant or a transactional one (this may include, if necessary, advanced features that allow the same system to change this character dynamically).

3.5 Summary

In this chapter, we have first discussed a systematic way of developing local error detection and exception handling features for each application-level component system needed to protect it from a malfunctioning environment and vice versa, to assure its smooth integration into the action level activities and to guarantee known and consistent fault assumptions at the component system level. In the next part of the chapter, we have presented an advanced CA action scheme that does not need entry or exit synchronisation but still keeps action atomicity, exception and error propagation under control. So, when an exception occurs, our support finds an action containing all erroneous information and involves all of its participants into cooperative recovery. Our analysis shows that this scheme is particularly useful for building fault tolerant SoSs out of general autonomous component systems. The following part of the chapter outlined the general framework to be used by SoS integrators at the application level and the functionalities of the DSoS fault tolerance support.

In our future research, apart from further work on the topics mentioned in Sections 3.2.3, 3.3.3 and 3.4, we intend to apply the ideas presented in the chapter in a case study, to do some experimental work (e.g., using one of the standard component technologies), to try to develop design patterns that turn autonomous systems into SoS component systems, to investigate, in detail, possibilities of run-time handling of property mismatches to prevent them from causing SoS failures.

Chapter 4 – Wrapping Mechanisms for DSoSs

4.1 Introduction

The main objective of developing wrapping technology in the DSoS context relates to the need to build error confinement areas (i.e., *Fault Containment Regions*, FCR – see. DSoS Report – *Conceptual Model* [IC1]) around systems and components. Since we consider composition of systems in DSoSs, it is mandatory to ensure that component systems (including legacy systems and off-the-shelf components) behave as specified and that internal errors do not propagate to the entire system of systems. This means that we must define wrapping mechanisms to be placed around them in such a way that their internal errors and failures are correctly reported to the outside world, when they cannot be recovered inside the component itself. This is a crucial issue to design, at a later stage, error recovery strategies in a larger context, based on the cooperation of other systems.

So far, our work on wrapping technology has concentrated on the definition of a generic framework and on its specialisation using formal description techniques for generating and implementing wrappers. The basic framework relies on modelling system requirements to derive expected properties. The properties of a given system component are described preferably in some formal syntax that gathers both behavioural and temporal aspects. For instance, temporal logic can be used to define these properties and wrappers can be automatically produced by compiling the formal specifications. Resulting wrappers can thus account both for timing and functional constraints. The error detection relies on the runtime verification of the properties by executing the wrappers on-line. The implementation framework that has been developed to run the wrappers requires extended observability and controllability from the target system components. In particular, several observability levels can be considered, depending on the system component and the kind of wrappers that must be executed. Reflection can be used here as an enabling technology for improving both observability and controllability. Some initial experimental results obtained with a real time application running on a COTS real-time microkernel illustrate the benefits of the approach in terms of error detection coverage in both the time and the value domain.

Nevertheless, several variants of this framework can be defined. The range of possible instances starts from basic filtering assertions to specifications in a formal language, which shows the various possibilities of specialising this framework in the context of DSoS. Tradeoffs between error detection efficiency, the required observability of the considered target system or component, and overheads are key parameters in this respect.

The remainder of this chapter is structured as follows. Section 4.2 describes the basic principle of the wrapping technology proposed together with related work. In Section 4.3, we provide an overview of the proposed wrapping framework. An instance of this framework based on specifications in temporal logic and the automatic translation into wrapping code is described in Section 4.4. Section 4.5 presents some experimental result obtained with the specialised framework described in section 4.4. Section 4.6 elaborates on various alternatives for implementing this framework that can be useful in the context of DSoS. They mainly depend on the target system component, the wrapping objectives and related performance overhead. Finally, Section 4.7 provides some preliminary conclusions about this step of our work.

4.2 Principles and Related Work

A wrapper can be defined in general terms as a software component that sits around a target component or system (i.e., that can be included within a *Connection System*, see DSoS Report – *Conceptual Model* [IC1]). Traditionally, wrappers have been used in the security domain (e.g., [Cheswick & Bellovin 1994) to enforce security policies through the isolation of software (i.e., filtering techniques as in Firewalls).

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 two parts: an *adapter*, providing additional services to applications, and an *encapsulation mechanism*, responsible for linking components. This definition is mostly related to interfacing heterogeneous systems. Some examples of this kind of wrapper are the so-called *interceptors* provided by CORBA and DCOM+, and the *proxies* in CORBA3 and Enterprise Java Beans.

In this work, we are mainly concerned with wrappers for error detection and error containment. The notion of *error confinement wrapper* was introduced by Voas [Voas 1998] in relation with the use of COTS (*Commercial Off-The-Shelf*) components in the design and implementation of dependable systems. In this work, the author distinguishes input and output wrappers. Input wrappers can be seen as *filters* preventing syntactically incorrect inputs from reaching the component. Output wrappers submit outputs to an acceptance test before being released. For example, the work reported in [Ghosh *et al.* 1999] provides robustness wrappers for filtering erroneous inputs to off-the-shelf software applications in Windows-NT based systems.

Wrapping can also be achieved using *executable assertions* [Mahmood *et al.* 1984, Rabéjac *et al.* 1996, Hiller 2000]. Executable assertions usually consist of checks inserted into the source code of the target component, which use predefined rules to test the validity of a given condition (e.g., a post-condition that checks a variable with respect to its expected value at the end of a block of instructions). Executable assertions can be used during software development to aid developers in finding faults in the system, but also in operation as part of fault-tolerance mechanisms. As an example of the latter, the work reported in [Salles *et al.* 1999] proposes a platform to efficiently implement wrappers for COTS microkernels from executable assertions.

When the set of predefined rules used by assertions are based on a formal specification of the target component, we talk of *runtime verification*. Runtime verification [Diaz *et al.* 1994, Jahanian *et al.* 1994, Mok & Liu 1997, Savor & Seviora 1997, Schneider 1998] can be seen as another kind of wrapping, where a monitor checks system constraints at runtime against an executable formal description of the system. All works on runtime verification have in common:

- A specific formalism, used to describe the system.
- A monitor, corresponding to executable code of the formal description to check constraints at runtime.
- A characterisation/abstraction of the system behaviour, that defines the way the system is viewed by the monitor.

Runtime verification allows impacts of faults to be detected at runtime. Wrappers can thus be used to harden the system against faults (namely, *hardware faults* and *software bugs*). In a SoS, wrapping techniques can be used following an *onion-like* model, to tackle different types of faults and considering nested systems. Each wrapped component (or system) can later be considered as a self-checking component (or system), a notion that has many benefits for the definition of error recovery strategies.

Significant research has been carried out in the field of runtime verification. The work described in [Diaz *et al.* 1994] introduces the concept of *observer* for designing self-checking distributed systems. The observer is an on-line monitor that checks the system behaviour against an executable model of the system. In the paper, the observer concept is developed for formal models based on Petri nets and LAN-based distributed systems built on a broadcast service. The approach is applied to the OSI layering of an open system architecture, to the Link and Transport layers of an industrial LAN, and to a virtual ring MAC protocol.

Many other studies have used Real Time Logic (RTL) to monitor timing constraints of real-time tasks at runtime (e.g., see [Jahanian *et al.* 1994, Mok & Liu 1997]). Timing properties of tasks are modelled in RTL and an efficient runtime monitor is derived from the defined set of constraints. The objective is to detect timing violations as early as possible. The system is viewed as a sequence of event occurrences triggered by tasks and sent to the monitor. The latter detects timing violations by resolving constraints with the actual timestamps of events.

The work reported in [Savor & Seviora 1997] defines an *out-of-time* supervisor for programs whose requirements specifications consist of non-deterministic SDL models. The system is viewed as a set of input and output signals that are processed by the supervisor an arbitrary amount of time after their occurrence (i.e., out-of-time). The approach is exemplified in this reference work with a telecom application.

Based on the principles discussed above, our objective is to define a generic wrapping framework that can be specialised for different types of systems or software components in the context of DSoS. The proposed wrapping framework essentially provides means to improve error detection but also, to some extent, means to trigger recovery actions.

4.3 Proposed Wrapping Framework

The target component to be wrapped is referred to as the *Target System/Software Component* (TSC), i.e., either a software component or a component system in the DSoS context. Our framework is composed of several elements resulting from *Modelling* of the TSC requirements, *Generating wrappers* and requirements for the *Implementation* of the wrapping software on the actual TSC.

These elements are the following: (i) the reference, which is the formal description of the system requirements, (ii) the wrapping, which comprises the wrappers and the required runtime software, and (iii) the observability but also controllability (including system clock), which characterises how the behaviour of the system is perceived by the wrappers. Figure 4-1 provides an overall description of the framework proposed.

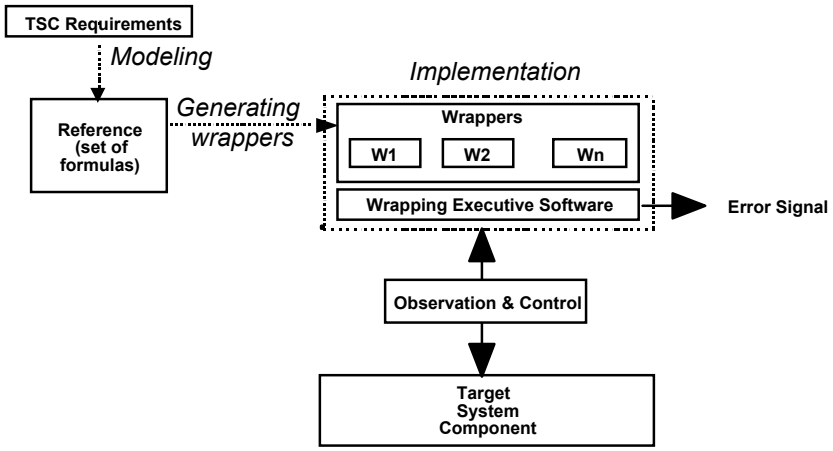

Figure 4-1: Overall framework

The **Modelling** process involves deriving the reference model that corresponds to the TSC specification. The reference formally expresses the correct behaviour of the system as perceived from outside, i.e., by components interacting with the TSC. The resulting formal expressions correspond to a set of formulas derived from the TSC requirements. The set of formulas precisely defines what is expected from the TSC, at least from a non-functional viewpoint.

Wrappers generated from the formal expressions must be able to verify the corresponding properties on-line. A wrapper consists of an executable code dependent on the formula, and hence different specifications produce different wrappers. In practice, the formulas must be translated into some executable code and mapped through an executive software layer to the real TSC. Basically, each formula of the specification produces a single wrapper.

The **Implementation** of the wrappers requires observation and control of the target system in operation. The specification formally exhibits the parts of the TSC behaviour that must be observed and controlled to perform the verification and to act on the TSC for recovery actions. It is worth noting that, in general, it is not useful to observe the complete behaviour of the TSC. There are several objective reasons for this, for instance:

- Too many items should have to be monitored at runtime (too much performance overhead).
- Some behaviours of the TSC are of little interest (e.g., the behaviour can be expressed by a restricted set of variables).

Observability and controllability issues concerning the TSC are crucial in the proposed approach. They characterise how the TSC behaviour is perceived or viewed by the wrappers. The behaviour of the TSC might be described by a sequence (or a set) of messages [Diaz *et al.* 1994], event occurrences [Mok & Liu 1997], signals [Savor & Seviora 1997], states [Schneider 1998], etc. Indeed, it depends very much on the formalism used to describe the TSC requirements, but also on the target system that is considered. In other words, the TSC can be seen as an *Abstract Finite State Machine* (AFSM) that defines, at a given abstraction level depending on the observability level, the perception of the TSC in term of states and transitions (clock triggers and event occurrences). The verification of the specification is carried out on this model.

4.4. Wrapping and Temporal Logic

This section is devoted to the specialisation of the general framework (see Figure 4-2) proposed in Section 4.3 using temporal logic to express TSC specifications [Rodriguez *et al.* 2001]. We have defined a formal language based on a temporal logic to specify system requirements. A salient feature of this work is that such formulas can be compiled automatically to generate the wrappers. A runtime checker executes the wrappers.

The main contribution of our approach with respect to prior work in the field is the automatic generation of the monitoring code (i.e., the wrappers) by compiling both timing and functional constraints of the system expressed in a temporal logic.

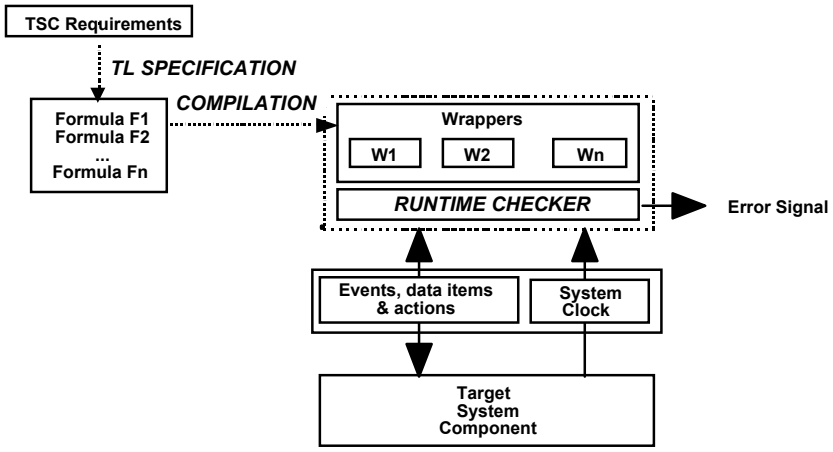

Figure 4-2: Temporal Logic-based framework

The main difference between the formal language we have defined and standard temporal logics is that it specifically introduces the notions of *discrete time* (in the form of *clock triggers*) and *event*, and defines accordingly extended temporal operators. Wrappers are automatically generated by compiling temporal logic specifications to code that contains calls to a runtime checker. The latter is independent from TSC specifications and is responsible for the execution of the wrappers. It detects timing and value failures of the TSC operation with respect to its specifications. Indeed, the runtime checker is an interpreter of temporal logic formulas and raises error signals whenever a statement of the specification is evaluated to false. The main advantage of having a runtime checker is that it makes the wrappers independent from the actual TSC. Indeed, the runtime checker can be seen as a

sort of *virtual machine* for the execution of wrappers. Therefore, porting the wrappers to different systems means porting just the runtime checker. Regarding observability issues, the TSC is viewed as a set of variables that describe its internal state between different *clock triggers* and *event occurrences*. Indeed, the logic is built from predicates that describe the state of the system variables, which are checked by the temporal operators at different clock triggers and event occurrences. This approach has been proved useful and efficient when considering real-time COTS microkernels.

4.4.1 Modelling

This section describes how temporal logic can be used to specify functional and temporal properties of real-time kernels at different levels of abstraction. Temporal logic is a first order logic extended with some temporal operators. The temporal operators are the following: *always* (□), *sometime* (○), and *next* (◆).

As far as dependable systems are concerned, two main classes of properties must be considered, namely, *safety* properties and *liveness* properties. Safety properties state that no matter what inputs are given, and no matter how nondeterministic choices are resolved inside the system, the system will not reach a specific undesirable configuration (e.g., deadlock, emission of undesired outputs, etc.). Liveness properties state that some desired configuration will be visited eventually or infinitely often (e.g., expected response to an input). Safety and liveness properties of executive software packages (e.g., microkernels, operating systems kernels, middleware layers) can be expressed in temporal logic. Examples given here are extracted from experiments conducted on COTS executives (see Section 4.5) and address scheduling aspects.

As an example of a safety property, consider the following formula:

$$\square \neg ((\exists\, s \in [\text{Tasks}]: s \neq \text{idle} \wedge s \in [\text{ReadyQueue}]) \wedge [\text{Running}] = \text{idle}) \quad (1)$$

The formula states that task idle will never be executed if any other task is ready to run. Task idle has the lowest priority in the system and it is always ready to run, so it runs whenever no user task is in the dispatch queue ([ReadyQueue]). Therefore, a situation in which task idle is running while the dispatch queue is not empty leads to the violation of the safety property expressed by formula (1), which may be viewed as the occurrence of a faulty behaviour of the kernel.

A liveness property is expressed by the following formula:

$$\forall i \in [\text{TimeoutQueue}] : \blacklozenge ([\text{Flow}] = \text{CANCEL}(i) \vee i \in [\text{TimeoutQueue}_0]) \quad (2)$$

The formula states that a timer will trigger as long as it is not cancelled. Timers are usually used as alarms by real-time tasks to detect deadline misses. For instance, each time a periodic task is released, it sets a timer ($i \in$ [TimeoutQueue]) with an initial timeout value equal to its deadline. If the task instance finishes by the deadline, it cancels the timer ([Flow] = CANCEL (i), meaning that the execution flow triggered a cancel operation) and suspends until its next period. However, the task might not get to disable the timer by the deadline because of a system overload (too much

computation, task preemptions, or task blockings). In this case, the timer eventually triggers ($i \in [TimeoutQueue_0]$) and an exception is raised, warning about the missed deadline.

Properties can be expressed at different levels of abstraction and complexity. To illustrate this issue, consider formula (3), which is a variant of formula (2):

$$\Box\ ([Flow] = SET\ (i, D_i, 0) \Rightarrow\ \bigcirc^{< D_i - t}\ ([Flow] = CANCEL\ (i)) \vee \blacklozenge^{D_i - t}\ (i \in [TimeoutQueue_0])) \qquad (3)$$

The interpretation of formula (3) is as follows. Predicate $[Flow] = SET\ (i, D_i, 0)$ states that the execution flow triggered a request to set timer i, with initial timeout value D_i, and a null period interval (*one-shot* behaviour). Timeout D_i denotes the absolute instant of time at which timer i will trigger, and is equal to the deadline of the requesting task. Setting a timer implies, then either:

- Timer i is cancelled by the deadline D_i ($\bigcirc^{<D_i - t}\ ([Flow] = CANCEL\ (i))$, where t is the current system time, and $D_i - t$ is thus the relative deadline with respect to the current time), or
- Deadline D_i is eventually missed ($\blacklozenge^{D_i - t}\ (i \in [TimeoutQueue_0])$).

Clearly, formula (3) is less abstract than formula (2), even though both of them refer to the same liveness property. The level of complexity and abstraction of the formulas can thus be tuned so as to reduce the time overhead induced by the verification process (less complex and more abstract formulas), or to increase the efficiency of the verification by performing more exhaustive analyses of the microkernel (more complex and less abstract formulas).

A comprehensive specification of microkernel requirements in temporal logic is provided in [Rodriguez *et al.* 2000]. We extracted from these kernel specification, a single temporal logic formula, namely, formula Create (Figure 4-3) which describes the creation of higher priority tasks.

$$\Box\ (\ [Flow] = \uparrow Create\ (th_b) \wedge th_a = [Running] \wedge$$
$$\bigcirc\ (\ [Flow] = \uparrow signal\ (th_b) \wedge [Running] = th_a \wedge prio\ (th_b) > prio\ (th_a)\ ) \Rightarrow$$
$$\blacklozenge_{event}\ (\ [Flow] = \downarrow context\_switch \wedge [Running] = th_b \wedge th_a \in [ReadyQueue_{prio(th_a)}]\ )\ )$$

Figure 4-3: Formula Create

The interpretation of formula Create is as follows. Whenever a request for the creation of a task ($\uparrow Create$) is issued by running task th_a ([Running]), some time later the kernel initiates the insertion of a newly created task th_b into the ready queue ($\uparrow signal$). As long as the priority of th_b is higher than the priority of th_a, task th_a is preempted at the completion of the next context switch operation. As a result, th_b is elected as the newly running task, whereas th_a is inserted back into the ready queue. This formula is used for illustration later in the subsequent paragraph of Section 4.

4.4.2 Generating Wrappers

The wrappers are implemented by translating each formula into standard C code. The translation process can be automated by means of a *compiler* of temporal logic into the C language. Execution of

the resulting wrappers relies on a specific interface for observability (and controllability) and on the runtime checker services. Consider again formula Create (see Figure 4-3). Figure 4-4a provides the plain text expression of this formula, while Figure 4-4b shows how the formula is translated into standard C code. The runtime checker services for running this code are indicated in capital letters (e.g., NEXT_EVENT, ASSERT, etc.). The interface of the runtime checker is partially described in Section 4.4.3. The needed services for observing TSC internal data and events are those identified with prefix tsc- (e.g., tsc-getPrio, tsc-getRunning, etc.).

```
Formula : [Create]

Always ( [Flow] = begin_create(thb) & tha = [Running] &
     Sometime ([Flow] = begin_signal(thb) & [Running] = tha & prio(thb) > prio(tha))
=>
     Next_event ([Flow] = end_ ctxswt & [Running] = thb & tha in [ReadyQueue(prio(tha))]) )
```

a) Plain text of formula Create

```
int Create_start () {

        return ALWAYS (ev_begin_create, (void*)Create_always) ;
}
int Create_always (Context* context) {

        context = NEW_CONTEXT ();
        CONTEXT_PUT (1, tsc-getFlowParam (1), “thb”, context);
        CONTEXT_PUT (2, tsc-getRunning (), “tha”, context);

        return SOMETIME (ev_begin_signal, (void*)Create_sometime, context);
}
int Create_sometime (Context* context) {

        int thb = CONTEXT_GET (1, context);
        int tha = CONTEXT_GET (2, context);

        CONDITION (tsc-getFlowParam (1) == thb && tsc-getRunning () == tha &&
                    Tsc-getPrio (thb) > tsc-getPrio (tha) );

        return NEXT_EVENT (1, (void*)Create_next_event, context);
}
int Create_next_event (Context* context) {

        int thb = CONTEXT_GET (1, context);
        int tha = CONTEXT_GET (2, context);

        ASSERT (tsc-getFlow () == ev_end_ctxswt && tsc-getRunning () == thb &&
                tsc-isThreadInReadyQueue (tha, tsc-getPrio(tha)) == TRUE );
}
```

b) C code of formula Create

Figure 4-4: Wrapper implementation

In Figure 4-4b, the original formula is divided into a start function (`Create_start`), and three main routines, one for each temporal logic operator used (`Create_always`, `Create_sometime`, and `Create_next_event`). The internal structure of these three routines is very similar:

- State data is either saved to or restored from memory (`CONTEXT_PUT`, `CONTEXT_GET`). Service `NEW_CONTEXT` allocates memory (from a static memory pool) for each new instance of the wrapper.
- The system state is checked, either to evaluate the antecedent (`CONDITION`) or the consequent (`ASSERT`) of the formula.
- Unless the end of the formula is reached, an event (e.g., `ev_begin_signal)` is scheduled by a temporal logic operator (`SOMETIME`, `NEXT_EVENT`).

Function `Create_start` requests the runtime checker to activate the wrapper (`ALWAYS`). The execution flow of the system is diverted to one of the three main routines when the corresponding scheduled event occurs. In Figure 4-4, event `ev_begin_create` makes execution divert towards `Create_always`, event `ev_begin_signal` towards `Create_sometime`, and whatever event occurring right after event `ev_begin_signal` towards `Create_next_event`.

Error detection mainly relies on the evaluation of the consequent of the formula by using service `ASSERT`. It accepts a Boolean expression as an input parameter, which is assessed by the runtime checker. An error is signalled if such an expression evaluates to false (see. example in Section 4.5).

4.4.3 Implementation

The Runtime Checker

The runtime checker is responsible for the assessment of the temporal logic formulas. The evaluation of a formula is explicitly requested, by a wrapper, of the runtime checker. Accordingly, the runtime checker provides to the wrappers the interface shown in Figure 4-5.

Temporal Logic Formulas	Interface services (C language)
$\Box$ ([Flow] = event $\wedge$ p)	`int Always (int event, void* p);`
$\bigcirc$ ([Flow] = event $\wedge$ p)	`int Sometime (int event, void* p, void* context);`
$\blacklozenge_{event}^{i}$ (p)	`int Next_event (int i, void* p, void* context);`
$\blacklozenge_{event}^{\{grp\}}$ (p)	`int Next_event_grp (void* grp, void* p, void* context);`
$\blacklozenge_{\downarrow tick}^{i}$ (p)	`int Next_tick (int i, void* p, void* context);`
$\blacklozenge_{\uparrow tick}^{i}$ (p)	`int Tick_next (int i, void* p, void* context);`

Figure 4-5: Runtime checker interface

The first column contains the type of temporal logic formulas accepted by the runtime checker, while the second column lists the corresponding interface's services. Parameter `p` refers to a temporal logic

formula, whereas parameter `context` points to a structure storing state data. For example, formula Create (Figure 4-3) accesses services `Always`, `Sometime` and `Next_event`.

As previously explained, the wrapping executive software (i.e., the runtime runtime checker in this Section) must be informed of the occurrence of TSC events and clock ticks and also be able to access TSC state information (observation and control box in Figure 4-1). A communication channel is established for this purpose between the TSC and the wrapping executive software. This channel is controlled by the runtime checker using some special services. The services must be provided to map the runtime checker to a given TSC. The related services are listed in Figure 4-6.

Services interface in C language
`void connectEventHdl (void* evHdl)`
`void connectTickHdl (void* tickHdl)`
`void switchonEvent (int eventId)`
`void switchoffEvent (int eventId)`

Figure 4-6: Services used by the runtime checker

The runtime checker defines two handlers, one for managing kernel events (`evHdl`), and a second one for dealing with clock ticks (`tickHdl`). The runtime checker connects them to the TSC using services `connectEventHdl` and `connectTickHdl`, respectively. Moreover, events can be switched on and off at runtime, so as to optimise the execution of the runtime checker and reduce its overhead. This is provided by services `switchonEvent` and `switchoffEvent`, respectively.

Observability of Events and Data Items

Observability concerns the way the target software component is perceived by the wrappers. In practice, it is related to the ability to observe TSC events and data at runtime. This section discusses the possible approaches for the provision of observability.

The observability of the TSC can be provided in different ways, depending on whether the TSC cooperates or not with the wrappers in supplying the needed information [Diaz *et al.* 1994]. We shall distinguish three different cases:

1. *No cooperation from TSC:* The wrappers must spy the behaviour of the TSC to detect the occurrence of events and obtain data.

2. *Cooperation from TSC:* The TSC informs the wrappers when a significant event occurs, sending the corresponding data at the same time.

3. *Cooperation from TSC and wrappers:* The TSC informs of the occurrence of events, and the wrappers subsequently request the necessary data.

Concerning the first point, the behaviour of the TSC is obtained from information that is already accessible to the wrappers. The main advantage of this approach is that neither the design nor the software of the TSC needs to be modified. The main drawback is that the internal behaviour of the TSC cannot be observed, but only its external activity. This type of observability can be readily

applied to monitor communication protocols in distributed systems, as reported in [Diaz *et al.* 1994], where an observer obtains TSC information by directly snooping the buses. Supervision of telecom applications [Savor & Seviora 1997] also benefits from this type of observability, where checks are made with respect to the input and output signal activity generated in the system.

Concerning the second point, the TSC must be modified with the insertion of interceptors that catch and deliver the necessary events and data. An interceptor may, for example, be a call to the wrapping executive layer. The main advantage of this approach is that the internal behaviour of the TSC can be observed and controlled. The main drawback is that documents and software of the TSC must be available in order to identify the precise locations where interceptors must be inserted. As an example of this approach we cite the work in [Mok & Liu 1997], where Java programs are annotated with a special class method, called *trigger*, that sends both the name and the occurrence time of events over a socket connection to the monitor.

Concerning the third point, first, the TSC informs the wrappers of the occurrence of an event, and then, the wrappers explicitly get the needed data from the TSC. This approach presents the same advantages and disadvantages as the previous one. However, the main benefit of this approach with respect to the cooperative observability is that it easily allows different sets of information to be selected and obtained at each event occurrence. As an example, the work reported in [Salles *et al.* 1999] uses the notion of *reflection* (see [Maes 1997]) to provide this kind of observability to microkernels. The reflective approach is further described in Section 4.6. It is worth noting that the wrapping framework proposed in Section 4.3 is, *a priori*, independent from the approach used for supplying observability and controllability, and thus can be applied to a broad range of TSCs in the context of DSoS.

4.5 Experimental results: examples

In operational life, errors due to design and physical faults might be revealed and activated as the result of a particular execution of the TSC, a COTS microkernel in our case study (an early version of the Chorus microkernel). Such errors can be successfully detected thanks to the runtime verification of properties. Following the framework described in Section 4.4, a runtime checker is in charge of performing the verification of the properties expressed by the temporal logic formulas. The runtime checker, in conjunction with the temporal logic formulas, can be viewed as an extended wrapper, which checks whether the behaviour of the microkernel matches the specification.

Error detection is thus based on the on-line verification of kernel properties. The runtime checker detects the occurrence of an error whenever a property is violated at runtime, i.e., whenever a temporal logic formula is evaluated to false. Such a violation means that the property expressed by the formula does not hold for a particular execution context of the kernel. This can be regarded as the detection of an error, since the behaviour of the kernel no longer matches its specification. The runtime checker can then perform some actions so as to put the system into a controlled state, ranging from a crash (*fail-silent* behaviour) to a graceful degraded mode of the system.

4.5.1 Scheduling example

Let us consider again formula Create (Figure 4-3). If several tasks request the creation of a higher priority task, formula Create is then evaluated concurrently for different instances of the task's

identifiers (namely, th_a and th_b). This means that different instances of the same temporal logic formula are concurrently evaluated under various kernel contexts during execution of the system. As long as the kernel behaves correctly, formula Create is true in any execution interval of the kernel for whatever values of the task's identifiers.

Assume that a given task th_a, with id 11 and priority 7, initiates the creation of a higher priority task th_b, which is given id 10 and priority 9. Consider that an error in the kernel makes the term $th_a \in [ReadyQueue_{prio(th_a)}]$ evaluate to false. This means that either task th_a was not inserted into the ready queue of the corresponding priority, or it was not inserted at all. The runtime checker offers a report on the status of any violated formula. For the current example, the corresponding report is shown in Figure 4-7.

```
1. Violation at [7379126 us]
2.      -- [Formula = Create]
3.      -- [tha = 11]
4.       -- [thb = 10]
5.      -- [prio (tha) = 7]
6.      -- [prio (thb) = 9]
7.      -- [tha in ReadyQueue (prio (tha)) == TRUE] : [0 == 1]
```

- *Line 1 shows the system time at which the violation is detected*
- *Lines 3 and 4 show the actual values for the task's identifiers, while task's priorities are given in lines 5 and 6*
- *Line 7 contains the term of the formula which was evaluated to false*

Figure 4-7: Example of a violation report

The runtime checker also accounts for *pending formulas*. A pending formula is a formula that is not completely evaluated by the end of the execution interval for which the kernel is verified. This feature is related to the use of operator $\bigcirc$ in formulas like $\bigcirc$ ([Flow] = ev $\wedge$ p), where formula p is never evaluated unless event ev occurs, i.e., as long as predicate [Flow] = ev is false in the execution interval considered. Let us look again at the description of formula Create in Figure 4-3. Although task th_a requests the creation of a task, the kernel might never serve such a request due to higher priority tasks constantly preempting task th_a and saturating the CPU. Under these conditions, event $\uparrow$signal (th_b) will never occur because task th_a is never given access to the CPU again. At the end of the execution interval, the runtime checker will warn that event $\uparrow$signal (th_b) is pending. This feature can thus be used to detect faulty tasks that never release a resource (namely, the CPU in this case).

To summarise, the verification of properties can assist system designers in detecting design errors in early versions of the microkernel. Moreover, even if the kernel is well designed and satisfies the required properties beforehand, transient faults in the hardware might impact the kernel during operational life. In other words, even though some properties might be redundant for correct kernels (i.e., kernels without design errors), it is however useful to consider checking them so as to harden the

kernel against the various sources of errors that may occur at runtime. The set of properties to be verified can thus be viewed as a kernel wrapper based on formal specifications. Therefore, our approach encompasses errors due to both design faults and physical faults impacting the kernel in operation. As a result, wrappers based on formal specifications help in developing correct kernels as well as in hardening COTS kernels at runtime, which is of high interest for both system designers and integrators.

4.5.2 Kernel Timers Example

We illustrate here how design and implementation errors of the kernel can successfully be detected by wrappers. The timer service of the Chorus microkernel was encapsulated by the wrapper corresponding to a formula expressing the correct behaviour of timers. This formula, which specifies the correct behaviour of kernel service `set_timer`, involves kernel calls (such as *SetTime*r) and events (such as ↓TimeoutSet corresponding to the completion of a *SetTimer* call within the kernel). A full account of this formula (named timer_1) can be found in [Rodríguez *et al.* 2000]. The workload used was a periodic task, τ_A, whose pseudo-code is shown in Figure 4-8

```
1.  task body Thread is
2.  begin
3.     Initialize ();
4.     set_timer (tm, tABS, T);
5.     loop
6.          wait_next_release ();
7.          Periodic_Code ();
8.     end loop;
9.  end Thread;
```

Figure 4-8: Periodic task

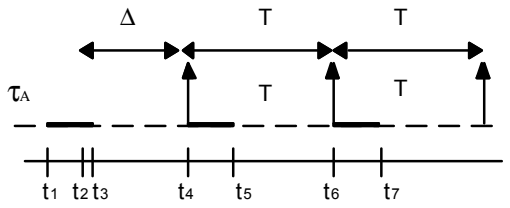

Figure 4-9: Nominal behavior

The task first initialises (line 3) and executes the `set_timer` system call, which requests the kernel to set a periodic timer `tm`, with absolute start time t_{ABS}, and period `T` (line 4). Next, it enters a loop where the task first suspends until its next release (line 6) and then executes its periodic code (line 7). Each release of the task is referred to as *instance* (I_i). The nominal behaviour of task τ_A for the execution of instances I_1 and I_2 is shown in Figure 4-9.

The initialisation routine is executed in interval $[t_1, t_2)$, whereas interval $[t_2, t_3)$ corresponds to the execution of service `set_timer` by the kernel. At time t_2, the kernel computes the first release time of τ_A, namely, Δ, as the difference between t_{ABS} and the current time t_2. At time t_3, the task enters the loop and suspends on `wait_next_release`. At time t_4, τ_A is released and executes until time t_5 (instance I_1), and then it executes from time t_6 to time t_7 (instance I_2).

Figure 4-10 shows the reports delivered by the runtime checker for the verification of service `set_timer` when task τ_A is executed. The kernel fails to compute the correct value of Δ (`delta`, in line 9), because it introduces an error of 1 tick. Indeed, in line 9, the kernel assigns 70 ticks to Δ, while the value computed by the wrapper is 69 ticks.

```
1. Violation at [9166093 us]
2.      -- [Property = Timer_1]
3.      -- [tha = 11]
4.      -- [tma = 30961816]
5.      -- [abs = 986 ticks]
6.      -- [period = 10 ticks]
7.      -- [offset = 0 ticks]
8.      -- [SysTicks = 916]
9.      -- [delta == abs + offset - [SysTicks] - 1]:[70 == 69]
10. Violation at [9166639 us]
11.     -- [Property = Timer_1]
12.     -- [tha = 11]
13.     -- [tma = 30961816]
14.     -- [abs = 986 ticks]
15.     -- [period = 10 ticks]
16.     -- [ticks = 69]
17.     -- [tma in [TimeoutQueue(ticks)] == TRUE]:[0 == 1]
```

Figure 4-10: Reports for the verification of `set_timer`

Formula timer_1 calculates the value of Δ as the number of *full* tick intervals (i.e., number of entire intervals between two consecutive ticks) between the next clock tick and tick t_{ABS}. Indeed, since only full intervals are considered, the current interval must be discarded, and Δ is computed starting from the next clock tick. For example, if t_{ABS} was 917 ticks, since current time t_2 is 916 ticks (`SysTicks`, in line 8), Δ should be assigned value 0, so that task τ_A can be released at the next tick interrupt. However, since the actual value of t_{ABS} is 986 (`abs`, in line 5), Δ should be assigned value 69 (computed as 986+0-916-1 by timer_1, as indicated in line 9). Nonetheless, the kernel assigns value 70 to Δ, thus delaying τ_A's first release until tick 987.

The wrapper corresponding to formula timer_1 successfully detects this behaviour. At time t_2, it detects that an incorrect value is given to Δ (in line 9), whereas in line 17, it detects (some microseconds later) that the related timer object is placed into an incorrect timeout queue, i.e., that corresponding to a timeout of 70 ticks instead of 69 ticks. From a performance point of view, the overhead introduced by the runtime checker for the verification of service `set_timer` was 232 μs.

4.6 Alternative Framework Instances

4.6.1 Modelling Alternatives

A TSC can be viewed as an abstract finite state machine (AFSM). However, the level of complexity of a TSC may make it difficult to: (i) obtain this AFSM, and (ii) practically verify properties at runtime, because of the state explosion problem. The definition of the AFSM can be induced from the requirements, ideally from a set of some logic formulas corresponding to the specification. The abstract view of the TSC machine's state is given by a set of *state variables*. State variables are related to objects handled by the TSC, not all visible to external systems or components. Accordingly, the AFSM derives from the specification of the expected behaviour of the TSC.

Transitions among states of the AFSM are triggered by a number of *events*. Basically, these events correspond to both external stimuli and the start or termination of actions. Clock interrupts and entering or leaving a TSC function are some examples of TSC events. From a practical viewpoint, TSC events can be activated by the passage of time, the TSC execution flow and stimuli from the environment. For executive software packages, the TSC environment can be viewed as being composed of the *real world*, which generates asynchronous inputs, the *hardware*, which raises interrupts and exceptions (e.g., internal error detection), and the *interacting components*, which issue service calls to the TSC.

Transitions of the automata are triggered by either the start or the completion of an operation. Accordingly, the various machine's states are distinguished by the different actions performed on the set of state variables defined. The various possible transitions between states refer to service calls to the TSC, namely ↑ServiceCalls(parameters), or the execution of some specific internal functions which are needed to animate the AFSM at runtime, namely ↑InternalFunctionStart(parameters), ↓InternalFunctionEnd(). It is worth noting that the various internal actions must be made visible to enable the verification of properties at runtime.

However, depending on the TSC considered, in particular in DSoSs (e.g., legacy systems or components), the required observability may be difficult to obtain. In this case, the inputs that govern the definition of wrappers are: (i) the TSC requirements, and (ii) the failure modes observed, either from the field when sufficient failure information is available or from fault injection experiments that exhibits:

- Lack of error detection mechanisms within the TSC.
- The manifestation of software faults in the TSC.

This approach is driven by the identification of input patterns and sequences of invocation requests or messages (protocol) that lead to a wrong behaviour of the TSC. This approach is conventional and simple to understand but suitable for complex TSCs whose internal activity is difficult to observe and master. This approach will be followed for defining wrappers for CORBA middleware packages according to characterisation results obtained in WP3 (see. DSoS Report – *CORBA Failure Analysis* [IC3]).

The use of conventional filtering and protocol analysis (using an AFSM) fit into the generic framework defined in Section 4.2. In this case, formulas rely on simple expressions to prevent unexpected situations (result of the modelling process), wrappers are generated in an *ad-hoc* manner to obtain executable assertions, and the implementation requires input/output requests to be intercepted. Some audit information that would be available from outside the TSC would also be of interest to improve the external checking of properties. This point refers to observability issues that are discussed in more details in the next section.

4.6.2 Observability Alternatives

Observability is an important feature of the implementation of the framework. This means that when no other option is possible, all the available information reported by the TSC can be of interest to adjust the verification of properties. It is obviously of prime importance to intercept input/output requests and collect at runtime possible information about the internal state of the TSC considered (error reports, message logs, audit trails, etc.). The amount of information that can be observed is very much dependent of the TSC considered. From this viewpoint, we classify TSCs in DSoS into three categories:

- **Legacy TSCs**: In the context of DSoS, considering legacy TSCs may introduce important constraints to the framework proposed. In this case, the TSC can be a black box and only information available from outside (delivered by the TSC or easily intercepted) should be considered in the modelling phase of our framework to define and set up the wrappers.

- **Open TSCs**: the TSC includes built-in features that enable some internal activity to be observed and controlled. This is the case for CORBA packages that provide the class? *Portable Interceptor* to track, intercept, and customise all interactions between CORBA objects. This is also the case for a *Java Virtual Machine* that enable the internal state of Java objects to be observed and serialised (*Serialisation Interface*) for recovery.

- **Reflective TSCs**: the reflective approach generalises and extends the two previous cases and provides an implementation framework for the wrappers. The required observability, but also controllability features are provided to the wrapping executive software (see Figure 4-1) through *interception* mechanisms and through a clear interface for *introspection* and *intercession*, called *metainterface*. The implementation of these features is provided by an additional software package, the *reflection package*, which can take advantage of and extend existing reflective features of the TSC. The reflection package implements the “observation & control” box of Figure 4-1.

In the next section we provide a detailed description of the reflective implementation framework that provides the observation and control features.

4.6.3 Implementation Alternatives

As already stated, the implementation of the abstract finite state machine (AFSM) can rely on a reflective approach. The AFSM is an image of the TSC in operation, what in reflective terms is referred to as the *metamodel*. In practice, the TSC specification is verified against the metamodel of the TSC. The metamodel is animated by *TSC events*, and its states are defined by *state variables* (see

Section 4.4). This means that TSC events must be intercepted and some data structures observed. In reflective terms, the former relates to the notion of *reification*, and the latter to the notion of *introspection*. In addition, as far as fault tolerance is concerned, some other features are required to act upon the TSC to reflect some changes within when needed, e.g., when an error is detected. Here comes the notion of *intercession* that covers all aspects dealing with controlling the behaviour of the target system.

In a reflective system [Maes 1987, Kickzales 1991, Fabre & Pérennou 1998] a clear distinction is made between the so-called *base-level*, running the target system, and the *metalevel*, responsible for controlling and (possibly) updating the behaviour of the target system. Information is provided from the base-level to the metalevel, that becomes *metalevel data* or *metainformation*. Any change in the metainformation is reflected to the base-level. Metalevel data can be viewed as an abstraction — a model — of the base-level behaviour and structure. Indeed, such a model corresponds to the *metamodel*, mentioned above. The distinction made between the base-level and the metalevel provides a clear separation of concerns between the functional aspects handled at the base-level and the non-functional aspects (here, error detection and confinement) handled at the metalevel.

Figure 4-11 illustrates the various layers, components and mechanisms that make up the reflective implementation framework. This framework complies with and extends the principles introduced in [Salles *et al.* 1999].

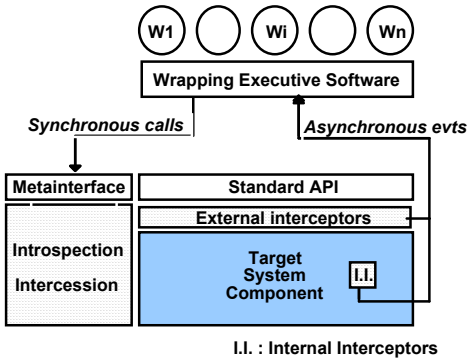

Figure 4-11: Reflective implementation framework

The base-level of our reflective system is the TSC, while the metalevel is part of the Wrapping Executive Software. The metalevel obtains the necessary events through interception facilities, can request information through the so-called metainterface, and can request actions through this interface as well. The *reflection package* corresponds to:

- *External and Internal Interceptors* that reify service calls of the TSC and internal function calls and events, respectively.
- *Introspection* facilities that can be accessed trough the metainterface to obtain internal TSC state information.
- *Intercession* facilities that can also be accessed through the metainterface to adjust the behaviour of the TSC at runtime and also perform some recovery actions.

Reification allows the AFSM (i.e., the metamodel) to be animated by triggering state transitions. Introspection allows the wrapping executive software to access the required data items, those appearing in the formulas. Intercession allows the wrappers to perform actions on the TSC. This facility can be used for recovery.

The definition of the metainterface can be directly derived from the TSC specification (formulas Fi, in Figure 4-1). Indeed, the specification points out the necessary events, data structures and functions of the TSC that must be observed and controlled. The implementation of the metainterface consists of a custom component added to the target TSC. For example, the metainterface for real-time microkernels includes services identified in Figure 4-12.

Temporal logic	**Metainterface (C language)**
[Flow]	`int getFlow (int* event, void* params);`
[Running]	`int getRunning ();`
[ReadyQueue$_i$]	`int isThreadInReadyQueue (int th, int i);`
prio (th)	`int getPrio (int th);`

- *`getFlow` returns the current kernel event occurred (`event`) and the related parameters (`params`)*
- *`getRunning` returns the identifier of the currently running task*
- *`isThreadInReadyQueue` returns true as long as task `th` is found in the ready queue of level `i`*
- *`getPrio(th)` is a state function that returns the priority of task `th`*

Figure 4-12: Metainterface derived from formula Create

The reflective framework can be used, not only for the implementation of extended error detection mechanisms based on formal specifications, but also, to some extent, for the implementation of recovery actions, the latter being outside the scope of present work. Nevertheless, it is worth pointing out that the intercession mechanisms enable the state of the TSC to be updated, corrected or stopped when an error is detected. The latter is similar to the notion of exception handling using the termination model [Miller & Tripathi 1997].

4.7 Summary

The major contribution of this work is the provision of a generic wrapping framework that can be specialised for various DSoS components. Depending on the type of TSC and its related observability features, an instance of this framework can be defined and used to implement error confinement wrappers. This framework can be the basis for building *Connection Systems* that prevent both timing and value faults to propagate to the entire system of systems.

When specification can be expressed in temporal logic, one key feature of this work is to provide automatic generation of error confinement wrappers by compiling system specifications. Such wrappers account for both timing and functional requirements of the target system. The on-line detection of errors is achieved by a runtime checker, in charge of executing the wrappers. Porting the wrappers to different systems means porting just the runtime checker. Indeed, the runtime checker can

be seen as a virtual machine for the execution of wrappers. The proposed wrappers can accommodate several observability approaches depending on the target software component. For instance, various techniques can be used to implement interceptors, including the use of TSC built-in interceptors (e.g., CORBA *Portable Interceptors*).

The provided case study illustrated how this wrapping approach can be used to improve error detection and error containment of real-time systems. However, the proposed wrapping framework is sufficiently general to be applied to various kinds of TSC in DSoS, like CORBA middleware systems, real-time applications and real-time communication systems.

Chapter 5 - Conclusion

This deliverable has presented the latest project results in the area of architecture and design of DSoSs, whose activities relate to the three following topics: (i) architecture-based development of DSoSs, (ii) mechanisms for enforcing dependability of SoSs, and (iii) wrapping technology for adapting and protecting component systems composing a DSoS.

Regarding the first topic, Chapter 2 has presented a developer-oriented, architecture-based development environment. This environment builds around an extensible ADL, which may be later specialised to describe DSoS-specific architectural styles as well as to enable the exploitation of DSoS-specific methods and tools for the system's design, analysis and implementation. Specialisation of the ADL has been discussed from the standpoint of supporting quality analysis of the DSoS from both a qualitative and a quantitative perspective. Illustration of the environment usage has further been sketched, using the Travel Agent case study. One contribution of our work comes from the concern of making the task of the developer easier, ultimately aiming at the actual use of our environment. While the importance of system architecting, and hence the need for precise architecture description, is now widely acknowledged, as e.g., exemplified by the "*IEEE Recommended Practice for Architecture Description"*[28], solutions to this issue that have been proposed by the software architecture community have hardly been exploited. The main reason for this is that system developers prefer using standard notations for the description of system architectures (typically, UML) and are rarely willing to invest on specifying architectures using formal notations, although this enables more thorough analysis of their systems, as demonstrated by ADLs that have been proposed (e.g., see [BC2] for a survey). Our environment promotes the actual use of ADLs and associated methods and tools, by defining an extensible ADL that is based on UML, and addressing its specialisation for rigorous system analyses while minimising the needed expertise in formal methods. The latter issue is addressed through the integration of quality attributes within the architectural elements, and the automatic generation of formal system models from the resulting architecture descriptions.

The definition of mechanisms for enforcing dependability of SoSs has been addressed in Chapter 3, focusing on application-specific fault tolerance mechanisms. A systematic way of developing local error detection and exception handling features for each application-level component system has first been presented. This allows component systems to be protected from malfunctioning environment and vice versa, to assure their smooth integration into DSoS and to guarantee known and consistent fault assumptions at the component system level. An advanced CA action scheme has then been presented. Compared to past solutions, this scheme does not need entry or exit synchronisation, but still keeps action atomicity, exception and error propagation under control. Thus, when an exception occurs, the CA Action support finds an action containing all erroneous information and involves all of its participants in cooperative recovery. Analysis of the proposed solution shows that this scheme is particularly useful for building fault tolerant SoSs out of general autonomous component systems. The following part of the chapter has outlined the general framework to be used by SoS integrators at the application level and the functionalities of the DSoS fault tolerance support.

[28] IEEE Std 1471-2000, October 2000.

Chapter 4 has concentrated on aiding the development of wrappers for the integration of component systems. The major contribution of this work is the provision of a generic wrapping framework that can be specialised for various DSoS components. Depending on the type of target component system and its related observability features, an instance of this framework can be defined and used to implement error confinement wrappers. This framework can be the basis for building Connection Systems that prevent both timing and value faults to propagate to the entire system of systems. When specification can be expressed in temporal logic, one key feature of this work is to provide automatic generation of error confinement wrappers by compiling system specifications. Such wrappers account for both timing and functional requirements of the target system. The on-line detection of errors is achieved by a runtime checker, in charge of executing the wrappers. Porting the wrappers to different systems means porting just the runtime checker. Indeed, the runtime checker can be seen as a virtual machine for the execution of wrappers. The proposed wrappers can accommodate several observability approaches depending on the target software component. For instance, various techniques can be used to implement interceptors, including the use of built-in interceptors (e.g., CORBA *Portable Interceptors*). The provided case study has illustrated how this wrapping approach can be used to improve error detection and error containment of real-time systems. However, the proposed wrapping framework is sufficiently general to be applied to various kinds of system components in DSoSs, like CORBA middleware systems, real-time applications and real-time communication systems.

The project's future work in the AD workpackage will be on further elaborating the proposed solutions to architecture-based development, dependability mechanisms and wrapping technologies, which will address the specific requirements of DSoSs. Part of our future work is on the definition of novel architectural styles, following results from the CM and AD WPs. In this context, we have started examining the definition of an architectural style, based on the CA Action scheme, for the development of fault-tolerant Internet-based DSoSs. At the most abstract level, the architectural style is structured around the composition of CA Actions, while at more concrete levels, each CA Action refines into a composite component, which comprise component systems interacting via a multi-party connector that realises the CA Action coordination protocol.

References

[Aggarwal *et al.* 1990] S. Aggarwal and C. Courcoubetis and P. Wolper. "Adding Liveness Properties to Coupled Finite-State Machines", *ACM Transactions on Programming Languages and Systems*, 12(2), pp 303-339, 1990.

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

[Bjorner *et al.* 2000] N. Bjorner and A. Browne and M. A. Colon and B. Finkbeiner and Z. Manna and H. B. Sipma and T. E. Uribe, "Verifying Temporal Properties of Reactive Systems: A SteP Tutorial", *Formal Methods in System Design*, 16(3), pp. 227-270, 2000.

[Browne *et al.* 2000] A. Browne and H. Sipma and T. Zhang, "Linking SteP with SPIN", in Proc. *7th International SPIN Workshop*, pp. 181-186, 2000.

[Butler 1992] R. Butler, "The SURE Approach to Reliability Analysis", *IEEE Transactions on Reliability*, 41(2), pp. 210-218, 1992.

[Butler & Johnson 1995] R. Butler and S. Johnson, "Techniques for Modeling the Reliability of Fault-Tolerant Systems With the Markov State-Space Approach", *NASA Reference Publication 1348*, 1995.

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

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

[Cimatti *et al.* 1999] A. Cimatti and E. Clarke and F. Giunchiglia and M. Rovery, "NuSMV: A New Symbolic Model Verifier", in Proc. *11th International Computed Aided Verification Conference*, pp. 495-499, 1999.

[Dashofy *et al.* 2001] E. dashofy and A. van der Hoek and R. Taylor, "A Highly Extensible, XML-based Architecture Description Language", in Proc. *2nd Working IEEE/IFIP Conference on Software Architecture (WICSA-2),* pp 103-112, 2001.

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

[Diaz *et al.* 1994] M. Diaz, G. Juanole and J.-P. Courtiat, "Observer - A Concept for Formal On-Line Validation of Distributed Systems", *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 900-913, 1994.

[Emerson & Halpern 1986] E. A. Emerson and J. Y. Halpern, "Sometimes and Not Never Revisited" On Branching versus Linear Time Temporal Logic", *Journal of the ACM*, 33(1), pp. 151-178, 1986.

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

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

[Formal Methods group, CMU 2001] Formal Method Group, "Formal Methods – Model Checking", CMU, Available at http://www.cs.cmu.edu/~modelcheck, 2001.

[Formal Systems (Europe) 2001] Formal Systems (Europe) Ltd, "Failure Divergence Refinement: FDR2 User Manual", Available at http://www.formal.demon.co.uk/fdr2manual/fdr2manual_toc.html, 2001.

[Garlan *et al.* 1997] D. Garlan and R. Monroe and D. Wile, "ACME: An architecture interchange language", Technical Report, Department of Computer Science, CMU, 1997.

[Garlan *et al.* 2000] D. Garlan and J. Kompanec and P. Pinto, "Reconciling the needs of architectural description with object-modeling notations", in Proc. *3rd International Conference on the Unified Modeling Language (UML-00),* 2000.

[Geist & Trivedi 1990] R. Geist and K. Trivedi, "Reliability Estimation of Fault Tolerant Systems : Tools and Techniques", in *IEEE Computer*, 23(7), pp. 52-61, 1990.

[Ghosk *et al.* 1999] K. Ghosh, M. Schmid and F. Hill, "Wrapping Windows NT Sofware for Robustness", in Proc. *29th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-29),* Madison, WI (USA), pp. 344-347, 1999

[Hansen & Fredholm 2001] B. E. Hansen and H. Fredholm, "Adapting C++ Exception Handling to an Extended COM Exception Model", in *Advances in Exception Handling Techniques*, A. Romanovsky, C. Dony, J. Knudsen and A. Tripathi (Eds.) LNCS-2022, Springer- Verlag, 2001.

[Hiller 2000] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems", in Proc. *IEEE Int. Conference on Dependable Systems and Networks (DSN-2000),* New York, NY (USA), pp. 24-33, 2000

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

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

[Issarny & Banâtre 2001] V. Issarny and J-P. Banâtre. "Architecture-based Exception Handling", in Proc. *34th Annual Hawaii International Conference on System Sciences (HICSS'34)*, 2001.

[Jahanian *et al.* 1994] F. Jahanian, R. Rajkumar and S. Raju, "Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems", *Real-Time Systems*, vol. 7, no. 3, pp. 247-273, 1994.

[Johnson 1988] S. C. Johnson, " Reliability Analysis of Large Complex Systems Using ASSIST ", in Proc. *8th AIAA/IEEE Digital Avionics Systems Conference*, pp. 227-234, 1988.

[Kazman *et al.* 1999b] R. Kazman and M. Klein and P. Clements, " Evaluating Software Architectures for Real -Time Systems ", *Annals of Software Engineering*, 7, pp. 71-93, 1999.

[Kazman *et al.* 1999c] R. Kazman and M. Barbacci and M. Klein and S. J. Carriere and S. G. Woods, " Experience with Performing Architecture Tradeoff Analysis ", in Proc *21st ACM-SIGSOFT-IEEE International Conference on Software Engineering (ICSE-99)*, pp. 54-63, 1999.

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

[Kiczales 1991] G. Kiczales, J. d. Rivières and D. G. Bobrow, *The Art of the Metaobject Protocol,* MIT Press, 1991.

[Kim & Yang Y89] K. H. Kim and S. M. Yang, "Performance Impacts on Look-Ahead Execution in the Conversation Scheme", *IEEE Transactions on Computer*, C-38, 8, pp.1188-202, 1989.

[Klein *et al.* 1999] M. Klein and R. Kazman and L. Bass and S. J. Carriereand M. Barbacci and H. Lipson, " Attribute-Based Architectural Styles ", in Proc. *1st IFIP Working Conference on Software Architecture (WICSA-1),* pp. 225-243, 1999.

[Kloukinas & Issarny 2001] C. Kloukinas and V. issarny, "SPIN-ning Software Architectures", in Proc. *2nd Working IEEE/IFIP Conference on Software Architecture (WICSA-2),* pp 67-76. 2001.

[Kobayashi 1978] H. Kobayashi, *Modeling and Analysis : An Introduction to System Performance Evaluation Methodology*, Addison-Wesley, 1978.

[Koopman & DeVale 2000] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems*", IEEE Transactions on Software Engineering,* SE-26, 9, pp.837-48, 2000.

[Laprie 1985] J. C. Laprie, "Dependable Computing and Fault Tolerance : Concepts and Terminology ", in Proc. *15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, 1985.

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

[Lilius & Paltor 1999] J. Lilius and I. Paltor, "vUML, A Tool for Verifying UML Models", in Proc. *14h International Conference on Automated Software Engineering (ASE'99)*, pp. 255-258, 1999.

[Maes 1987] P. Maes, "Concepts and Experiments in Computational Reflection", in Proc. *OOPSLA'87,* Orlando, FL (USA), pp. 147-155, 1987

[Magee *et al.* 1999] J. Magee, J. Kramer and D. Giannakopoulou, "Behavior Analysis of Software Architectures", in Proc. *1st IFIP Working Conference on Software Architecture (WICSA-1),* pp.35-49, 1999.

[Mahmood *et al.* 1984] A. Mahmood, D. M. Andrews and E. J. McCluskey, "Executable Assertions and Flight Software", in Proc. *6th Digital Avionics Systems Conf.,* Baltimore, Maryland (USA), pp. 346-351, 1984.

[Medvidovic & Rosenblum 1999] N. Medvidovic and D.S. Rosenblum, "Assessing the Suitability of a Standard Design Method ", in Proc. *1st IFIP Working Conference on Software Architecture (WICSA-1)*, pp. 161-182, 1999.

[Medvidovic & Taylor 2000] N. Medvidovic and R. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages",IEEE Transactions on Software Engineering, 26(1), pp. 70-93, 2000.

[Medvidovic *et al.* 1999] N. Medvidovic and 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-21)*, pp. 44-53, 1999.

[Miller & Tripathi 1997] R. Miller and A. Tripathi, "Issues with Exception Handling in Object-Oriented Systems", in Proc. *11th European Conference on Object-Oriented Programming (ECOOP'97),* Jyväskylä, Finland, 1997.

[Mitchell *et al.* 1998] S. E. Mitchell, A. J. Wellings and A. Romanovsky, "Distributed Atomic Actions in Ada 95", *Computer Journal*, 41, 7, pp.486-502, 1998.

[Mok & Liu 1997] A. K. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints", in Proc. *3rd IEEE Real-Time Technology and Applications Symposium,* Montreal, Canada, pp. 252-262, 1997.

[Powell *et al.* 2000] D. Powell, J.-P. Blanquart, Y. Crouzet and J.-C. Fabre. "Architectural Approaches for using COTS Components in Critical Applications", in *11th European Workshop on Dependable Computing* (EWDC-11). Budapest, Hungary, May 11-13, 2000. http://domino.inf.mit.bme.hu/EWDC-11.nsf

[PVS 2001] "The PVS Specification and Verification System", Available at http://pvs.csl.sri.com/, 2001.

[Rabéjac *et al.* 1996] C. Rabéjac, J.-P. Blanquart and J.-P. Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection", in Proc. *26th Int. Symp. on Fault-Tolerant Computing (FTCS-26),* Sendai, Japan, pp. 138-147, 1996.

[Randell 1975] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, SE-1, 2, pp.220-321975.

[Randell *et al.* 1997] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu and A. F. Zorzo. "Coordinated Atomic Actions: from Concept to Implementation", Computing Dept., University of Newcastle upon Tyne, TR 595, 1997.

[Richters *et al.* 2000] M. Richters and M. Gogolla, "Validating UML Models and OCL Constraints", in Proc. *3rd International Conference on the Unified Modeling Language (UML-00),* 2000.

[Rodriguez *et al.* 2000] M. Rodríguez, J.-C. Fabre and J. Arlat, "Formal Specification for Building Robust Real-time Microkernels", in Proc. *21st IEEE Real-Time Systems Symposium (RTSS 2000),* Orlando, Florida (USA), pp. 119-128, 2000.

[Rodriguez *et al.* 2001] M. Rodríguez, J.-C. Fabre and J. Arlat, "From temporal logic specifications to error confinement wrappers", *LAAS Research Report n°01171*, May 2001, 19p.

[Romanovsky 2001a] A. Romanovsky "Coordinated Atomic Actions: How to Remain ACID in the Modern World*", ACM Software Eng. Notes*, 26, 2, pp 66-8, 2001.

[Romanovsky 2001b] A. Romanovsky. "Looking Ahead in Atomic Actions with Exception Handling", to be presented at the *20th Symposium on Reliable Distributed Systems*, New Orleans, USA, October, 2001.

[Roscoe & Broadfoot 1999] A. W. Roscoe and P. J. Broadfoot, "Proving Security Protocols with Model Checkers by Data Independence Techniques", *Journal of Computer Security, Special Issue CSFW11*, 1999.

[Rushby 1999] J. Rushby, "Integrated Formal Verification: Using Model Checking with Automated Abstraction, Invariant Generation, and Theorem Proving", *Dams et al*, pp. 1-11, 1999.

[Salles *et al.* 1997] F. Salles, J. Arlat and J.-C. Fabre, "Can We Rely on COTS Microkernals for Building Fault-Tolerant Systems", in *the 6th Workshop on Future Trends in Distributed Computing Systems*, Tunisia, 1997.

[Salles *et al.* 1999] F. Salles, M. Rodríguez, J.-C. Fabre and J. Arlat, "Metakernels and Fault Containment Wrappers", in Proc. *29th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-29),* Madison, WI (USA), pp. 22-29, 1999.

[Savor & Seviora 1997] T. Savor and R. E. Seviora, "An Approach to Automatic Detection of Software Failures in Real-Time Systems", in Proc. *3rd IEEE Real-Time Technology and Applications Symposium,* Montreal, Canada, pp. 136-146, 1997.

[Schneider 1997] F. Schneider, "Enforceable Security Policies", Report no. TR98-1664, Department of Computer Science, Cornell University, Ithaca, NY (USA), 1998

[SMV 2001] "SMV Model Checker", Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/, 2001.

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

[UML v1.3] OMG, *UML Semantics* 1.3, 1997.

[Voas & Miller 1997] J. Voas and K. Miller, "Interface Robustness for COTS-based Systems", in *the Colloquium COTS and Safety Critical Systems*, IEE Group C1, January 1997.

[Voas 1998] J. M. Voas, "Certifying Off-the-Shelf Software Components", *Computer*, pp. 53-59, 1998.

[Wile 1999] D. Wile. "AML: An architecture meta-language", in *Proc. of the 14th IEEE Conference on Automated Software Engineering (ASE-99)*, 1999.

[Wolper 1986] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic", Extended Abstract, in Proc. *13th ACM Symposium on Principles of Programming Languages*, pp. 184-193, 1986.

[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, IEE CS Press,1995.

[Xu *et al.* 2000] J. Xu, A. Romanovsky and B. Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems", *IEEE Transactions on Parallel and Distributed Systems*, PDS-11, 10, pp.1019-32, 2000.

[Zarras & Issarny 2001] A. Zarras and V. Issarny, "Automating the Performance and Reliability Analysis of Enterprise Information Systems", To appear in Proc. *16th IEEE International Conference on Automated Software Engineering (ASE)*, 2001.

[Zarras *et al.* 2001] A. Zarras and V. Issarny and C. Kloukinas and V. K. Nguyen. "Towards a base UML Profile for Architecture Description", in Proc. *ICSE Workshop on Describing Software Architecture with UML,* pp 22-26, 2001.

DSoS Reports References

[BC2] *State of the Art Survey*, DSoS Deliverable, September 2000.

[DMS1] *Preliminary Dependability Modelling Framework*, DSoS Deliverable, April 2001.

[DMS3] *Case Study Problem Analysis*, DSoS Deliverable, April 2001.

[IC1] *Revised version of conceptual model*, DSoS Deliverable, September 2001.

[IC3] *Failure analysis of an ORB in the presence of faults*, DSoS Deliverable, September 2001.

Appendix 1 - Architecture-based Development Environment

A1.1 OCAML Expression for Checking Well-formedness of ADLconnector Elements

The following gives the OCAML expression that is generated from the OCL constraints specified in the definition of the ADLConnector element, which is used for checking the well-formedness of the instances of this element.

```
let eval =
((self#interfaces#isNotEmpty#oclAnd
  (self#ExtendedElement#allConnection#oclForAll
    (new oclInteger 0)
    (new oclBoolExpression
      function l ->
      let ae = (List.nth l ((List.length l) - 1)) in
      (ae#type#requires#oclExists
        (new oclInteger 0)
          (new oclBoolExpression
            function l ->
            let i = (List.nth l ((List.length l) - 1))
            in
            ((self#interfaces#includes i)#oclImplies
            (self#ExtendedElement#allConnection#oclExists
             (new oclInteger 0)
             (new oclBoolExpression
               function l ->
               let ae2 = (List.nth l ((List.length l) - 1)) in
               (ae#type#provides#includes i) ) ) ) ) ) ) )#oclAnd
 (self#ExtendedElement#allConnection#oclForAll
    (new oclInteger 0)
    (new oclBoolExpression
      function l ->
      let ae = (List.nth l ((List.length l) - 1)) in
      (ae#type#provides#oclExists
        (new oclInteger 0)
          (new oclBoolExpression
            function l ->
            let i = (List.nth l ((List.length l) - 1))
            in
            ((self#interfaces#includes i)#oclImplies
            (self#ExtendedElement#allConnection#oclExists
             (new oclInteger 0)
             (new oclBoolExpression
               function l ->
               let ae2 = (List.nth l ((List.length l) - 1)) in
               (ae#type#requires#includes i) ) ) ) ) ) ) );;
if eval#value = true
   then print_string "—Constraint holds --"
   else print_string "—Constraint does not  hold --"
```

A1.2 Model Checking the TA wrt Message Ordering

The following gives the whole PROMELA model that is used for checking the TA against ordered delivery of reply messages at the client. The given specification embeds two realisations of the HTTP connector (i.e., HTTP/1.0 and HTTP 1.1) and two realisations of the Web servers (i.e., sequential processing of incoming requests and concurrent processing of incoming requests).

```
#undef  PROTOCOL_1_0
#undef  SERIAL_SERVER
#define PROTOCOL_1_0        1
#define SERIAL_SERVER       1
#undef  SERIAL_SERVER
#undef  PROTOCOL_1_0
#undef  DEBUG
#define DEBUG               1
/*************************************************************************
 *                          Data Definitions                             *
 *************************************************************************/
mtype { white, red, blue } ;       /* enumeration types */
typedef Msg {
 mtype m ;                         /* The message itself. */
 int   s_id                        /* Sender's process ID. */
#ifndef PROTOCOL_1_0
 ;
 bit   type
#endif
} ;
#ifdef PROTOCOL_1_0
#          define port_ portRV
#else
#          define port_ portNB
#          define REQUEST        1
#          define REPLY   0
#endif
#define BUFSIZE         3
typedef portRV {chan port = [0] of {Msg}} ; /* Rendez-vous port type */
typedef portNB {chan port = [BUFSIZE] of {Msg}} ; /* Non-blocking port type */

 /* Used to pass ports to connectors */
chan startup_port_RV = [0] of {portRV} ;
chan startup_port_   = [0] of {port_} ;

chan start_verification = [0] of {bit} ;

bit sent_red_p = 0, rcvd_red_p = 0, rcvd_blue_p = 0 ;
/**/
/*************************************************************************
 *                          Connectors                                   *
 *************************************************************************/
proctype RPCconnector (portRV RPC_req ; portRV RPC_rep)
/*
In RPC, we must receive a reply before we can send another request,
so it's rendez-vous communication.
*/
{
 Msg    msg ;
 do
 :: RPC_req.port ? msg ->
```

```
   msg.s_id = _pid ->
   RPC_rep.port ! msg ->
   RPC_rep.port ? msg ->
   msg.s_id = _pid ->
   RPC_req.port ! msg
 od
}
proctype HTTPconnector (port_ http_req ; port_ http_rep)
/*
In HTTP 1.0, the client must first receive a reply before it can make another request.
So, we use  bufferless, i.e., rendez-vous, channels.
In HTTP 1.1, however, the client can pipeline multiple requests and then receive the replies.
So, we use channels with buffers, i.e., send is non-blocking now (unless the buffer is full, of course).
*/
{
 Msg    msg ;
#ifdef PROTOCOL_1_0
 do
 :: http_req.port ? msg ->          /* Receive a request from the client. */
   msg.s_id = _pid ->
   http_rep.port ! msg ->           /* Pass it to the server. */
   http_rep.port ? msg ->           /* Receive a reply from the server. */
   msg.s_id = _pid ->
progress_HTTPconnector:
   http_req.port ! msg              /* Pass it to the client. */
 od
#else
 /* In HTTP 1.1 we must make sure that the message *we* are receiving
   was not placed by us in the channel, in the first place. */
 do
 /* Receive a request from the client. */
 :: http_req.port ?? msg.m,msg.s_id,REQUEST ->
   msg.type = REQUEST ->
   msg.s_id = _pid ->
   /* Pass it to the server. */
progress_A_HTTPconnector: http_rep.port ! msg
#ifdef UNREACHABLE
   ->
   len(http_req.port) < BUFSIZE - 1 /* Leave one place for a reply. */
#endif
 /* Receive a reply from the server. */
 :: http_rep.port ?? msg.m,msg.s_id,REPLY ->
   msg.type = REPLY ->
   msg.s_id = _pid ->
   /* Pass it to the client. */
progress_B_HTTPconnector: http_req.port ! msg
 od
#endif
}
/**/
/************************************************************************
 *                          Components                                  *
 ************************************************************************/
proctype Customer ()
{
 port_ TA_connector ;
```

```
 Msg   msg ;
 bit   sent_blue_p = 0 ;
#ifndef PROTOCOL_1_0
 int   requests_in_channel = 0 ;
#endif
 /* Tell the startup process, which channels you use. */
 startup_port_   ! TA_connector ;
/*  start_verification ? _ ;*/
 printf("MSC: Client %d\n", _pid);
 do
 /* Choose the kind of message to send. */
#ifdef PROTOCOL_1_0
 :: if
#else
 /* Don't fill the channel with requests. */
 :: (BUFSIZE - 1 > requests_in_channel) ->
    if
#endif
    /* Send a red message if you haven't already sent one. */
    :: (! sent_red_p) ->
       msg.m = red -> sent_red_p = 1
    /* Send a blue message if you haven't done so, but only after
       having sent a red one. */
    :: (! sent_blue_p && sent_red_p) ->
       msg.m = blue -> sent_blue_p = 1
    /* Or, just send a white message. */
    :: msg.m = white
    fi ->
#ifndef PROTOCOL_1_0
    msg.type = REQUEST ->
    requests_in_channel = requests_in_channel + 1 ->
#endif
    msg.s_id = _pid ->
    TA_connector.port ! msg ->
 /* Read the reply. */
#ifdef PROTOCOL_1_0
    TA_connector.port ? msg ->
#else
 :: TA_connector.port ?? msg.m,msg.s_id,REPLY ->
    requests_in_channel = requests_in_channel - 1 ->
#endif
#           ifdef DEBUG
    printf(
"MSC: Client received a %d message from %d: rcvd_red_p = %d rcvd_blue_p = %d\n",
    msg.m ,msg.s_id,
    ( rcvd_red_p || (red  == msg.m)),
    (rcvd_blue_p || (blue == msg.m))) ->
#           endif
progress_Client:                    /* Client should always be receiving replies. */
    rcvd_red_p  = ( rcvd_red_p || (red  == msg.m)) ->
    rcvd_blue_p = (rcvd_blue_p || (blue == msg.m))
#           ifdef DEBUG
; printf("MSC: Client %d treated a %d message from %d\n", _pid,msg.m,msg.s_id)
#           endif
 od
}
```

```
/**/
proctype TravelAgentFrontEnd ( )
/*
The TravelAgentFrontEnd Web server implements two different kinds of servers:
 - If SERIAL_SERVER is defined, then each time it reads one request
   and immediately replies to it, before reading the next one.
 - If SERIAL_SERVER is not defined, then it treats requests in parallel.
   That is, the replies are no longer in a FIFO order with respect to the requests.
   We model this, by allowing a non-deterministic choice of particular
   requests (red or blue) when these exist and then immediately
   responding to these, irrespectively of their position in the
   channel's buffer.
*/
{
 port_  TA_connector_a ;          /* Port used by clients. */
 portRV TA_connector_b ;          /* Port used by underlying system. */
 Msg    msg ;
#ifndef SERIAL_SERVER
 port_  Slave_down, Slave_up ;
#endif
 /* Tell the startup process, which channels you use. */
 startup_port_   ! TA_connector_a ;
 startup_port_RV ! TA_connector_b ;

#ifndef SERIAL_SERVER
 run Slave(Slave_down, Slave_up) ;
 run Slave(Slave_down, Slave_up) ;
#endif
 do
 ::                               /* Receive a request. */
#ifdef PROTOCOL_1_0
    TA_connector_a.port ? msg ->
#else
    TA_connector_a.port ?? msg.m,msg.s_id,REQUEST ->
#endif
    msg.s_id = _pid ->
#ifndef SERIAL_SERVER
    /* Send it to a slave. */
#ifndef PROTOCOL_1_0
    msg.type = REQUEST ->
#endif
    Slave_down.port ! msg ->
    /* Receive the answer from the slave. */
#   ifdef PROTOCOL_1_0
 :: Slave_up.port ? msg ->
#   else
 :: Slave_up.port ?? msg.m,msg.s_id,REPLY ->
#   endif
#endif
#ifndef PROTOCOL_1_0
    msg.type = REQUEST ->
#endif
    /* Pass it to the underlying system - RPC. */
    TA_connector_b.port ! msg ->
    TA_connector_b.port ? msg -> /* Receive the reply from the system - RPC. */
    msg.s_id = _pid ->
```

```
#ifndef PROTOCOL_1_0
   msg.type = REPLY ->
#endif
   TA_connector_a.port ! msg    /* Send the reply to the client. */
 od
}
proctype FlightReservation ( )
{
 portRV TA_connector_a ;        /* Port used by clients. */
 port_  TA_connector_b ;        /* Port used by underlying system. */
 Msg    msg ;
 /* Tell the startup process, which channels you use. */
 startup_port_RV ! TA_connector_a ;
 startup_port_   ! TA_connector_b ;
 do
 :: TA_connector_a.port ? msg -> /* Receive a request - RPC. */
   msg.s_id = _pid ->
#ifndef PROTOCOL_1_0
   msg.type = REQUEST ->
#endif
   TA_connector_b.port ! msg -> /* Pass it to the underlying system. */
#ifdef PROTOCOL_1_0                    /* Receive the reply from the system. */
   TA_connector_b.port ? msg ->
#else
   TA_connector_b.port ? msg.m,msg.s_id,REPLY ->
#endif
   msg.s_id = _pid ->
   TA_connector_a.port ! msg    /* Send the reply to the client - RPC. */
 od
}
/**/
proctype AirCompanies ( )
/*
 AirCompanies models the final Web server. As for the TravelAgentFrontEnd component, it
 implements two different kinds of servers:
*/
{
 port_ TA_connector ;               /* Port used by clients. */
 Msg   msg ;
#ifndef SERIAL_SERVER
 port_ Slave_down, Slave_up ;
#endif
 /* Tell the startup process, which channels you use. */
 startup_port_   ! TA_connector ;
#ifdef SERIAL_SERVER
 do
 /* Receive a request. */
#ifdef PROTOCOL_1_0
 :: TA_connector.port ? msg ->
#else
 :: TA_connector.port ? msg.m,msg.s_id,REQUEST ->
   msg.type = REPLY ->
#endif
   msg.s_id = _pid ->
   TA_connector.port ! msg        /* Send the reply to the client. */
 od
```

```
#else  /* The server treats requests concurrently. */
 run Slave(Slave_down, Slave_up) ;
 run Slave(Slave_down, Slave_up) ;
 do
 /* Receive the reply from the system. */
#ifdef PROTOCOL_1_0                    /* Rendez-vous communication. */
 :: TA_connector.port ? msg ->
#else                                  /* Non-blocking communication. */
 :: TA_connector.port ?? msg.m,msg.s_id,REQUEST ->
    msg.type = REQUEST ->
#endif
    /* Pass the request to the slaves. */
    Slave_down.port ! msg
#ifdef PROTOCOL_1_0                    /* Rendez-vous communication. */
 :: Slave_up.port ? msg ->
#else                                  /* Non-blocking communication. */
 :: Slave_up.port ?? msg.m,msg.s_id,REPLY ->
    msg.type = REPLY ->
#endif
    TA_connector.port ! msg
 od
}
proctype Slave( port_ In ; port_ Out )
{
 Msg   msg ;
 do
    /* Receive the reply from the master. */
#ifdef PROTOCOL_1_0                    /* Rendez-vous communication. */
 :: In.port ? msg ->
#else                                  /* Non-blocking communication. */
 :: In.port ?? msg.m,msg.s_id,REQUEST ->
    msg.type = REPLY ->
#endif
    msg.s_id = _pid ->
    Out.port ! msg                     /* Send the reply to the client. */
 od
#endif
}
active proctype startup ()
{
  portRV up_RV, down_RV ;     /* up and down rendez-vous channels. */
  port_  up_, down_ ;         /* up and down unspecified channels. */
  printf("MSC: startup %d\n", _pid) ;
  run Client() ;
  startup_port_   ? up_ ;
  run TravelAgentFrontEnd() ;
  startup_port_   ? down_ ;
  startup_port_RV ? up_RV ;
  run HTTPconnector(up_, down_) ;
  run FlightReservation() ;
  startup_port_RV ? down_RV ;
  startup_port_   ? up_ ;
  run RPCconnector(up_RV, down_RV) ;
  run AirCompanies() ;
  startup_port_   ? down_ ;
  run HTTPconnector(up_, down_) ;
```

```
 startup_port_ ? down_
}
/**/
/*
 We want:
 1)   p = [] (sent_red_p -> <> rcvd_red_p) to hold. The case with blue is symmetrical.
      This stands for not accepting losses of messages.
 2)   q = !(!rcvd_red_p U rcvd_blue_p) to hold. This stands for arrival of messages in order.
      The property inside parentheses describes the case where we have received a blue message,
       but not a red one.
 I.e.:
                    ([] (sr -> <> rr)) && (!(!rr U rb))
#define sr      sent_red_p
#define rr      rcvd_red_p
#define rb      rcvd_blue_p
It should hold when PROTOCOL_1_0 is defined, or when SERIAL_SERVER is defined.
cat > pan.ltl << EOF
#define sr      sent_red_p
#define rr      rcvd_red_p
#define rb      rcvd_blue_p
EOF
cat > TA.ltl << EOF
(([] (sr -> <> rr)) && (!(!rr U rb)))
EOF
spin -F TA.ltl >> pan.ltl
cat > pan_in << EOF
#define PROTOCOL_1_0
#define SERIAL_SERVER
#undef PROTOCOL_1_0
#undef SERIAL_SERVER
EOF
cat TA.spin >> pan_in
spin -a -N pan.ltl pan_in
gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=64 -DXUSAFE -DNOFAIR  pan.c
time ./pan -m10000 -w19  -a -c1
- With PROTOCOL_1_0 defined and SERIAL_SERVER undefined, p /\ q holds
- With PROTOCOL_1_0 defined and SERIAL_SERVER undefined, p /\ q holds.
- With PROTOCOL_1_0 defined and SERIAL_SERVER defined, p /\ q holds.
- With PROTOCOL_1_0 undefined and SERIAL_SERVER undefined, p /\ q
  fails because blue is received before red.
I.e. ! (!PROTOCOL_1_0 && !SERIAL_SERVER) -> (p /\ q)
*/
```

A1.3 Quantitative Analysis of the TA wrt Performance and Reliability

A1.3.1 Part of the Queuing Network Model for Performance Analysis

The following gives part of the queuing network model that is generated from the performance-oriented UML-based architectural modelling of the TA and that is processed by the QNAP2 tool.

```
1    /STATION/
```

```
2   NAME = rpcCon;  CAPACITY = MAXINT;
3   TYPE = INFINITE;
4   SCHED = FIFO;
5   RATE = 1;
6   SERVICE(ta-i) =
7   BEGIN
8           NORMAL(2, 7);
9           TRANSIT(NEW(CUSTOMER), rpcCon, hr-i);
10          TRANSIT(NEW(CUSTOMER), rpcCon, fr-i);
11          TRANSIT(NEW(CUSTOMER), rpcCon, cr-i);
12          JOIN;
13          TRANSIT(OUT);
14  END;
15  SERVICE(hr-i) =
16  BEGIN
17          NORMAL(7, 12);
18          TRANSIT(NEW(CUSTOMER), httpib, ibis);
19          TRANSIT(NEW(CUSTOMER), httpsf, sofitel);
20          JOIN(1);
21          TRANSIT(OUT);
22  END;
23  SERVICE(fr-i) =
24  BEGIN
25          NORMAL(7, 12);
26          TRANSIT(NEW(CUSTOMER), httpaf, afrance);
27          TRANSIT(NEW(CUSTOMER), httpol, olairwa);
28          JOIN(1);
29          TRANSIT(OUT);
30  END;
31  SERVICE(cr-i) =
32  BEGIN
33          NORMAL(7, 12);
34          TRANSIT(NEW(CUSTOMER), httpav, avis);
35          TRANSIT(NEW(CUSTOMER), httphe, hertz);
36          JOIN(1);
37          TRANSIT(OUT);
END;
```

In the model generated for performance analysis given above, for each interface provided by the ta, hr, fr, and cr instances, corresponding services are generated and associated with the generated station (lines 6-14, 15-22, 23-30, 31-38, respectively). Multiple threads are used for providing those services (line 3). Moreover, the work demands required for the generated services are normally distributed. The generated station is associated with a FIFO queue of limited capacity (line 2), representing the RPC connector used for connecting ta, hr, fr, and cr.

Similarly, we generate stations and services for the component instances that constitute the instances of the Hotels, AirCompanies, and CarCompanies components. Finally, a source station hosting a service that represents the customer component instance is generated. This station creates initiation customers according to the POISSON distribution. Each initiation customer is the mapping of message 1 shown in Figure 2-6 of Chapter 2. The initiation customer is transferred to the station that is associated with the services representing interfaces provided by ta. Serving an initiation customer results in the creation of 3 new customers (lines 9-12), which are the mapping of messages 2, 3, 4 shown in Figure 2-6. The initiation customer waits until the 3 new customers are served. Serving the newly created customers results in the creation of three sets of customers which are the mapping of message sets 5, 6, 7 shown in Figure 2-6. Those sets of customers are sent to the stations that are

associated with the services representing interfaces provided by available hotels, air companies and car companies.

A1.3.2 Transition Rules of the State Space Model for Reliability Analysis

The following gives the transition rules of the state space model that is generated for reliability analysis of the TA, using the SURE-ASSIST tool.

1. IF ta = 0PERATIONAL THEN
2. TRANTO ta = FAILED BY Lambda_Lif;
3. ENDIF;
4. IF taNode = OPERATIONAL THEN
5. TRANTO taNode = FAILED BY Lambda_Hrdw;
6. IF ta = OPERATIONAL
7. TRANTO ta = FAILED BY Lambda_Hrdw;
8. IF hr = OPERATIONAL
9. TRANTO hr = FAILED BY Lambda_Hrdw;
10. IF fr = OPERATIONAL
11. TRANTO fr = FAILED BY Lambda_Hrdw;
12. IF cr = OPERATIONAL
13. TRANTO cr = FAILED BY Lambda_Hrdw;
14. ENDIF;
15. IF hotelNodes <= n-1 THEN
16. TRANTO hotelNodes++ BY Lambda_Hrdw;
17. IF hc <= n-1
18. TRANTO hc++ BY Lambda_Hrdw;

ENDIF;

For ta, for instance, the above rules state that if the scenario is in a state where ta is OPERATIONAL, then the scenario may reach a state where ta is FAILED (lines 1-3). The rate of this transition equals to the arrival rate of ta faults (ta.Faults.arrival-rate). Similar are the rules for the hr, fr, and cr component instances and for the instances of the RPC and HTTP connectors. For taNode the rules state that if the scenario is in a state where taNode is OPERATIONAL, then it may reach a state where it is FAILED and ta, hr, fr, and cr are also FAILED (lines 4-14). The rate of those transitions equal to taNode.Faults.arrival-rate. Finally, for one of the hc, fc, and cc redundancy schemas the rules state that if the scenario is in a state where the number of failed redundant component instances is smaller, or equal to the number of component faults that can be tolerated by the schema, then the scenario may reach a state where the number of failed redundant component instances is increased by one. Moreover, for each redundancy schema instance, the rules state that if the scenario is in a state where the number of failed redundant nodes is smaller, or equal to the number of node faults that can be tolerated by the schema, then the scenario may reach a state where the number of failed redundant nodes is increased by one and the number of failed redundant components is also increased by one (lines 15-19).

A1.3.3. Death State Constraint for Reliability Analysis

The following gives the death state constraint that is generated for reliability analysis of the TA using the SURE-ASSIST tool.

1. DEATHIF
    1. (
    2. ta = FAILED OR
    3. rpcCon = FAILED OR
    4. httpCon = FAILED OR
    5. taNode = FAILED OR
    6. hc > n-1 OR
    7. hcNodes > n-1 OR
    8. fc > n-1 OR
    9. fcNodes > n-1 OR
    10. cc > n-1 OR
    11. ccNodes > n-1 OR

);