

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Robust Development of Dependable Software Systems

Titos Saridakis

N ° 3712

Juin 1999

THÈME 1



ISSN 0249-6399

INRIA
RENNES

Robust Development of Dependable Software Systems

Titos Saridakis

Thème 1 — Réseaux et systèmes
Projet Solidor

Rapport de recherche n???? — Juin 1999 — 88 pages

Abstract: The indissoluble bonds of computers and failures have produced a plurality of fault tolerant techniques to satisfy, potentially, any dependability requirement. As a consequence, the development of dependable systems is not based on inventing the mechanism that provides the desired dependability guarantees. Rather, it is based on selecting from the existing techniques the one that best meets the system's dependability requirements. Then, some codification of the selected technique can be used as the search-key for retrieving from a repository of fault tolerant mechanisms the one that implements the selected technique. Hence, the development of dependable systems becomes a process that transforms a set of dependability constraints into a fault tolerant mechanism that meets them. The focus of our work is to ensure the rigorous development of dependable systems by creating a formal basis for the aforementioned selection process. More precisely, this formal basis consists of: a system model in terms of states and actions, which permits the specification of dependability properties in temporal predicate logic; a refinement relation for dependability properties, a refinement relation for system specifications, and their correlation. a means that derives automatically the architectural impact of a dependability property to a system specification. a classification scheme, which captures the property and specification refinement relations. This scheme organizes the contents of a repository of fault tolerant mechanisms, which results in accelerating their retrieval.

The practical contribution of this work is assessed in the Aster development environment. The formal basis is adapted to Aster specificities. In addition, a CASE tool is presented that uses the classification scheme to perform the analysis of dependability requirements. The CASE tool is integrated with the Aster toolkit, replacing the standard Aster tool in the retrieval of fault tolerant mechanisms. The analysis results are also used to provide guidelines for incorporating the selected fault tolerant mechanism in the system structure. While not innovative in its parts since it employs well known techniques from the fields of formal specifications, requirements analysis, software architecture, and software reuse, the originality of this work lies in the combination of existing software technologies for the benefit of system development. The development robustness is guaranteed by the rigorous analysis and the correct refinement of the system's dependability requirements.

Keywords: Architectural refinement, Automated software retrieval, Dependable system, Software architecture, Specification refinement.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Construction robuste de systèmes logiciels sûrs de fonctionnement

Résumé : Les liens indissolubles entre les ordinateurs et les défaillances ont conduit à l'invention d'une variété de techniques de tolérance aux fautes, pour satisfaire, potentiellement, toutes les exigences concernant la sûreté de fonctionnement. En conséquence, la construction de systèmes sûrs de fonctionnement n'est plus basée sur l'invention des mécanismes qui fournissent les garanties de sûreté de fonctionnement désirées, mais plutôt sur la sélection parmi des techniques existantes, de celle qui convient le mieux pour les exigences du système. Ensuite, une codification de la technique sélectionnée peut servir comme index pour extraire d'un répertoire contenant des mécanismes de tolérance aux fautes, celui qui met en oeuvre la technique sélectionnée. Il en résulte que la construction de systèmes sûrs de fonctionnement devient un processus de transformation d'un ensemble de contraintes de sûreté de fonctionnement, en un mécanisme de tolérance aux fautes qui les satisfait. Le but de notre travail est d'assurer la nature robuste de la construction de systèmes sûrs de fonctionnement, en fondant le processus de sélection sur une base formelle. Plus précisément, cette base formelle se compose de :

- Un modèle de système exprimé en termes d'états et d'actions, qui permet la spécification des propriétés de sûreté de fonctionnement dans une logique temporelle de prédicats.
- Une relation de raffinement de propriétés de sûreté de fonctionnement, une relation de raffinement de spécifications de systèmes, et leur corrélation.
- Des moyens pour dériver automatiquement l'impact architectural d'une propriété de sûreté de fonctionnement sur les spécifications d'un système.
- Un schéma de classification, qui capture les relations de raffinement des propriétés et des spécifications. Ce schéma structure le contenu d'un répertoire de mécanismes de tolérance aux fautes, et facilite le processus de sélection susmentionné.

La contribution pratique de notre travail est évaluée dans le cadre de l'environnement de développement Aster. La base formelle est adaptée aux spécificités d'Aster, et un outil est présenté, qui exploite le schéma de classification pour effectuer l'analyse des exigences concernant la sûreté de fonctionnement. Cet outil est intégré dans l'environnement Aster, et remplace l'outil standard pour l'extraction de mécanismes de tolérance aux fautes. Les résultats de l'analyse des exigences fournissent aussi les instructions pour l'intégration des mécanismes de tolérance aux fautes sélectionnés dans la structure du système. Bien qu'elle ne soit pas innovatrice, puisqu'elle utilise des techniques bien connues relevant des domaines des spécifications formelles, de l'analyse des exigences, des architectures logicielles, et de la ré-utilisation du logiciel, l'originalité de notre travail repose sur la combinaison des technologies existantes pour le bénéfice de la construction des systèmes. La nature robuste de cette construction est garantie par l'analyse rigoureuse et le raffinement correct des exigences d'un système concernant la sûreté de fonctionnement.

Mots-clés: Architecture logicielle, Raffinement d'architectures, Raffinement de spécifications Recherche systématique de logiciel, Système sûr de fonctionnement.

Contents

1 Introduction **1**
1.1 Motivations 1
1.2 Problems with the State of the Art 2
1.2.1 Software Fault Tolerance 3
1.2.2 Development Environments for Fault Tolerant Systems 3
1.2.3 Software Architecture 4
1.2.4 Summary 5
1.3 Objectives and Contribution 5
1.4 Structure of the document 7

2 The Aster Framework **9**
2.1 The Aster Development Environment 9
2.1.1 The Aster Prototype 10
2.1.2 Formalization and Specification Matching 11
2.1.3 Software Retrieval 12
2.1.4 CORBA compliance 13
2.2 The Evolution of Aster 13
2.2.1 Multiple Viewpoints on System Properties 14
2.2.2 Systematic and Robust Customization 14
2.2.3 Software Architecture 15
2.3 Discussion 16

3 Formalizing Dependability Properties **19**
3.1 The System Model 19
3.1.1 Basic Notations 20
3.1.2 Basic Definitions 20
3.1.3 Describing System Behavior 22
3.2 The Dependability Analysis Framework 24
3.2.1 Recursive System Structure 24
3.2.2 Specification Composition 25
3.2.3 Capturing Failure Events 26
3.3 Analyzing Dependability Properties 27
3.3.1 Abstract Dependability Properties 27

3.3.2 Property Refinement . . . 31
3.4 Evaluation . . . 33
3.4.1 Minimal Formal Framework . . . 33
3.4.2 Related Work . . . 34
3.4.3 Epilogue . . . 36

4 Systematic Dependability Analysis **37**
4.1 Dependable Software Architecture . . . 37
4.1.1 Specification Refinement . . . 38
4.1.2 Architectural Impact of Specification Refinement . . . 40
4.1.3 Integrating Property and Specification Refinement with their Architectural Impact. . . . 45
4.2 A Database of Refinement Patterns . . . 46
4.2.1 Classification of Refinement Patterns . . . 46
4.2.2 Updating the Database . . . 47
4.2.3 Guided Software Retrieval . . . 51
4.2.4 Deriving the Dependable System Architecture . . . 53
4.3 Evaluation . . . 55
4.3.1 Assessments . . . 55
4.3.2 Related Work . . . 56
4.3.3 Epilogue . . . 59

5 A Case Study **61**
5.1 A Distributed File System . . . 61
5.1.1 The CORBA Implementation . . . 62
5.1.2 The Dependability Requirements . . . 63
5.1.3 Fault Tolerant File Access . . . 65
5.2 Dependability Analysis . . . 67
5.2.1 Searching the Property Database . . . 68
5.2.2 Software Retrieval . . . 70
5.3 Deriving the Overall Dependable Architecture . . . 71
5.4 Summary . . . 74

6 Conclusions **76**
6.1 Summary . . . 76
6.2 Contributions . . . 77
6.3 Open Issues . . . 77
6.4 Epilogue . . . 79

A Glossary **81**

Chapter 1

Introduction

During the last four decades, the development of dependable software systems has been considered of paramount importance by many research and industrial organizations. An intuitive perception of a dependable system is that of a system that correctly provides the functionalities described in its specifications. At the same time, a dependable system harmlessly confronts a set of anticipated failures, where a failure is an abnormal event that can be interpreted as erroneous behavior with respect to the expected system functionality. Hence, the development of dependable systems has two aspects: the correct implementation of system functionalities, and the selection of the appropriate fault tolerant mechanism to deal with the anticipated failures. The focus of this thesis is to ensure the robustness of the latter aspect. More precisely, it suggests a sound framework for transforming a set of dependability requirements into the fault tolerant mechanism that satisfies them, and providing the guidelines for incorporating it in the architectural structure of the system. The remainder of the introduction presents the motivations of this thesis, discusses the problems with the current state of the art in the development of dependable systems, and sets the objectives of the thesis.

1.1 Motivations

The development of dependable systems is a difficult task. The number of the assorted parameters influencing the quality of dependability, combined with the miscellaneous cases of failure events that should be considered, increase enormously the complexity of system development. This complexity can no longer be successfully faced by humans without substantial computer aid. To provide the required systematic assistance to the development process, one needs a common interpretation of the dependability properties along all development phases (requirement specification and analysis, system design and implementation, retrieval and assembly of constituent software components). This common interpretation should be rigorously employed throughout the development process, because the slightest misinterpretation of some requirement may lead to incorrect implementation and compromise the dependability guarantees provided by the final product. Such a common interpretation may have the form of a commonly accepted vocabulary of dependability

properties, or the form of a set of different vocabularies with a precise *one-to-one* and *onto* mapping of terms from one vocabulary to another.

Using the common interpretation of dependability properties, the dependability requirements express the constraints associated to structural elements of the system (*i.e.* objects and interconnections), without sacrificing the clarity of the architectural description. The developer has to transform the architectural description of a system and its associated dependability requirements, into the architectural description of an equivalent dependable system. This raises a need for means that transform the architectural description annotated with the dependability properties of the participating entities, into an elaborated blueprint of the structure of the dependable system. In addition, it is highly desirable that such blueprints contain the guidelines for incorporating fault tolerant mechanism into the initial system structure, in order to relieve the developer from deducing them by hand.

The elaborated blueprint of a dependable system contains many details concerning the interconnections of the fault tolerant mechanism with the rest of the system, and the constraints of their utilization. This fact discourages the developers from taking into consideration every single detail of the dependable system blueprint. As a result, some details judged negligible by developers' intuition are omitted in order to achieve the construction of the system in reasonable time and with reasonable effort. This has a fatal repercussion on the correctness of the system implementation and, subsequently, on the robustness of the development process. To avoid it, significant machine-aid is required, which would permit to consider every detail of the system blueprint. Otherwise, the whole effort to express in a consistent manner the dependability requirements across the development process, and to produce a detailed structure that correctly reflects that system dependability qualities, is void. If a thorough analysis of dependability requirements cannot be translated into an equally correct system implementation, then it lacks the *raison d'être*. Consequently, there is an obvious need to provide the developer with the means to facilitate the translation from an architectural description to its precise implementation.

1.2 Problems with the State of the Art

While the problems related to the development of dependable systems have been recognized long ago, the solutions proposed through the last decades were mostly *ad hoc* and unorganized, or focused on specific cases and thus too restrictive. One can differentiate among three research fields that have been (and still are) occupied with the development of dependable systems: the software fault tolerance[1] field, the development environments for fault tolerant systems and the field of software architecture. In the remainder of this section, their limitations are stressed in order to justify why the author considers that they fall short from satisfying the needs raised in the robust development of dependable

[1] In the context of this report, software fault tolerance refers to the research domain that deals at the software level with failure of a computing system, as opposite to the domain of hardware fault tolerance which focuses on the structure and redundancy of the computing material (*e.g.* processors, memories, peripheral devices, network links, *etc*) necessary to cope with failures. The reader should not confuse this separation with the one that distinguishes software fault tolerance as the domain that deals with software faults (algorithmic errors or bugs).

systems. Inspired by Spector and Gifford's perspective [69] and to render the reasoning more intelligible for the reader, software engineering is compared with civil engineering and the development of software systems is juxtaposed with the construction of buildings.

1.2.1 Software Fault Tolerance

Software fault tolerance has been traditionally attacking the practical aspect of a dependable system, *i.e.* the conception, design, and implementation of fault tolerant mechanisms that can be used to confront a variety of failure events. As a separate field of research, software fault tolerance was dissociated from hardware-level fault tolerance at the late 60's and its coming to age was certified as early as in the mid-70's (*e.g.* see [58]). Since then, the field has been greatly evolved, revolving around two axes: *(i)* the definition of the foundations of software fault tolerance (including efforts like [1] and [41]), and *(ii)* the conception of new techniques (*e.g.* see [14]), the design of new protocols (*e.g.* see [55] and [12]) and the implementation of new mechanisms (*e.g.* see [65]) to deal with the ever increasing needs for dependability. As a result, the field has proposed a wide spectrum of fault tolerant mechanisms, that potentially cover every dependability requirement known today.

While the contribution of this field in the development of dependable systems is incontestable, providing the means to deal with failures does not imply the possession of the essential knowledge for assembling software components together and building dependable systems. Considering the symmetrical case in the civil engineering field, the knowledge of building rigid, practically unbreakable bricks does not imply the knowledge of putting them together to obtain a solid building construction. Although it is indispensable to have the materials providing the appropriate properties, it is equally important to know how to structure these materials to obtain a set of properties in the overall construction. In the software engineering domain, software fault tolerance does not provide this indispensable structuring knowledge. Hence, when the scale of the software system is small and simple, it is manageable to use the product of software fault tolerance to obtain the desired dependability properties for the system. But in the case of large scale distributed systems, the direct integration of a fault tolerance mechanism to obtain the desired dependability guarantees, is no longer attainable.

1.2.2 Development Environments for Fault Tolerant Systems

To deal with the problems of plugging the products of software fault tolerance into a prefixed system structure, a number of specialized development environments appeared. Such environments adopt a specific fault tolerant mechanism or a small set of such, and provide the corresponding functionalities as execution primitives. The utilization of these execution primitives by the software system ranges from completely transparent access to the fault tolerant mechanism (*e.g.* see Alphorn [3]), to a complete set of programming primitives that allow the developer to integrate a set of fault tolerant capabilities in a custom-made manner in the system (*e.g.* see Arjuna [52]). In any case, the developer does not have to search the entire domain of existing fault tolerant mechanisms to find which

one to use. Moreover, there is no need to study and understand the chosen mechanism and modify it to make it compatible with the system under construction.

Such development environments build on the results of the software fault tolerance domain, and provide a significant support to the developer. The latter is now relieved from the implementation details of the fault tolerant mechanism in use, and focuses on its integration into the system structure. Nevertheless, there are two main limitations with the use of such environments in the development of dependable systems: *(i)* they do not support the analysis of dependability requirements of the systems, and *(ii)* their final product is a system adapted to the fault tolerant capabilities offered by the environment, and not an execution platform customized to meet the system dependability needs. To make a parallel with civil engineering, these environments look like fixed architectural designs which indicate how to structure prefabricated materials to achieve specific characteristics. These specific characteristic cannot be modified. It is the form of the resulting building that is adapted to them. Hence, these development environments do not produce all kinds of dependable system, but rather a single family of such.

1.2.3 Software Architecture

The fundamental objective of software architecture is to capture in the design of a system the rationale of the architectural choices in order to provide *(i)* a set of constraints that guide the system implementation, and *(ii)* a documentation for the system that explains *why* implementation's obscure properties conform with design requirements. This is an up to date interpretation of the constitutional definition of the software architecture objective, given in the early 90s by Perry and Wolf [54]:

> "The primary focus in architecture is the identification of important properties and relationships – constraints on the kinds of components that are necessary for the architecture, design, and implementation of a system."

Software architecture is a complementary domain concerning the undertaken approach for the development of dependable systems. The focus of this domain is more on the robustness of the development process and the rigorous specification of the architectural structure of a system. Some of the most prominent work in the application of software architecture for the development of software systems has been coupled with the production of some executable form of the system (*e.g.* see work on Darwin [44], Rapide [43], and Unicon [68]). However, the principal contribution of software architecture remains the precise description of system structure. In the author's opinion, software architecture forms nowadays one of the most suitable research fields for producing an accurate blueprint of the system implementation that correctly complies with the system requirements concerning its algorithmic constraints, and its quality properties (*e.g.* dependability, security, timeliness, *etc*).

The essential drawback of software architecture in the current state of the art, is that it does not integrate the requirements for quality properties (and hence the dependability requirements) of a system, in a convenient way for the developer. To deal with this, the developer may produce an architectural description that does not consider dependability

requirements, and then try to incorporate the appropriate fault tolerant mechanism during the implementation phase. This would render void all verification processes about system consistency that where performed during the design phase. Alternatively, the developer may include the dependability concerns in the system design, and produce an elaborated architectural description that contains explicit entities, which satisfy dependability requirements. Without systematic computer aid, this alternative requires an enormous effort from the developer, who is obliged to deal with multiple, interwoven system aspects at the same time.

1.2.4 Summary

The conclusion drawn from the above presentation of the state of the art is that existing fields contributing to the development of dependable systems do not offer robust solutions for the general case of a dependable system. Development environments are shown to offer a restricted set of fault tolerant mechanisms, and to enforce the system modification to meet the mechanism characteristics. Software fault tolerance products do not provide any indication on how they should be integrated with a system, or whether they successfully meet all system's dependability requirements. Finally, software architecture does not provide the means to facilitate the analysis of the system's dependability requirements in order to identify the corresponding fault tolerant mechanism, and to integrate it with the system architecture. These facts give raise to an emerging need to provide support for the correct system design, which integrates dependability considerations in the system architecture. This support should transform a software system and the dependability requirements placed on it, into a single and thorough description of the dependable system architecture.

1.3 Objectives and Contribution

Following the above discussion, the problem in the development of dependable systems is localized in the selection of the appropriate fault tolerant mechanism and its correct incorporation in the system structure. This thesis proposes a solution for the rigorous analysis of the system's dependability requirements, the identification of the constituent components of the corresponding fault tolerant mechanism, and the delivery of the guidelines for their integration in the system structure. The objectives of this thesis are summarized in the following.

- The specification of a minimal logic vehicle consisting of a system model based on states and actions and a frugal temporal logic basis. This logic vehicle is used for expressing the system properties related to dependability requirements.
- The definition of a refinement relation for system specifications, based on the refinement relation of dependability properties. The refinement of system specifications is used to produce patterns that show how a dependability requirement can be analyzed to its constituent properties.

- The construction of a classification scheme that captures the above refinement patterns, and forms the basis of a CASE tool that performs the systematic analysis of the system's dependability requirements.
- The correlation of the dependability analysis results with a repository of fault tolerant mechanisms, to allow the guided software retrieval of the components that provided the required dependability properties.
- The decuction of means for automatically deriving the architectural impact of the application of a dependability property on a system specification, in order to integrated the selected components into the system architecture.
- The integration with the Aster development environment [32], which forms the framework that fostered this thesis.
- The last, but not least, objective concerns the assessment of the thesis' practical benefits, through the development of the dependable version of a distributed file system simulation.

Besides the primal contribution regarding the robustness of the development of dependable systems, this thesis also contributes to a number of related research fields. More analytically, these contributions are presented in the following.

Software Fault Tolerance: The formalization of well-known dependability concepts under a common logic vehicle, contributes to the rigorous documentation of the implied properties. In addition, the classification scheme organizes the refinement relations among the known dependability properties.

Software Architecture and Dependability Analysis: Coupling the dependability analysis with the software architecture of a system, promotes the rigorous and methodical integration of dependability concerns in the system architecture.

Design Reuse: Refinement patterns capture the dependability analysis process, and help deriving the architectural impact of the dependability properties when applied to system specifications. This knowledge can be reused in the development of any software system, having dependability requirements for which a refinement patterns exists.

Software Reuse: The proposed dependability analysis presents a systematic manner for transforming system requirements into search-keys for the retrieval of the appropriate fault tolerant mechanism that satisfies them.

Configurable System Customization: Transforming the system's dependability requirements into a fault tolerant mechanism that satisfies them provides a way for selecting the components that customize an execution environment to meet the application needs.

Although the proposed solution is tailored for Aster, it can be easily adapted to other configuration based environments, that customize the execution platform by reusing existing software components.

1.4 Structure of the document

Given the fact that this thesis is fostered in the Aster framework and in order to provide a legible understanding of what the thesis appends to the standard Aster capabilities, this report starts by presenting the Aster framework. After presenting the conceptual context in which the thesis was developed, Chapter 3 focuses on the definition of the logic vehicle, which consists of a system model and a minimal temporal predicate logic. The logic vehicle is used for the formal documentation of dependability properties and system specifications, and for the definition of a property refinement relation. Chapter 4 exploits the capabilities of the logic vehicle in offering a methodology and the associated means to transform the system's dependability requirements into a fault tolerant mechanism that satisfies them, and to provide the instructions for incorporating it in the system architecture. The benefits of this thesis are assessed in Chapter 5. The dependability requirements of a simulation of a distributed file system are analyzed using the proposed methodology, and the resulting fault tolerant mechanism selected from the Aster software repository is integrated with the system architecture. The report concludes with a summary of the suggested approach, a reminder of the contributions, and an overview of some important issues that are raised but not addressed in this thesis and which form the field of future work. At the end of this report, the reader may find and appendix containing a vocabulary of terms used in this report. It is aimed at serving as a quick reference for the reader who gets confused by the terminology used in this document.

Chapter 2

The Aster Framework

The goal of this chapter is to present the Aster context that fostered this thesis. The comprehension of this context will help the reader to understand the rationale behind this thesis' objectives, and to draw a clear line between the existing work in Aster and the contributions of this thesis. The chapter is organized in three sections. The first section presents the departure point and the initial objectives of the Aster project. The second one presents its evolution, which came as a natural consequence to the experience gained by the employment of the Aster environment in the development of distributed systems. The last section discusses the issues that this thesis attacks and the sets reference points in Aster from which this thesis departs. In the remainder of this chapter, the employment of the pronouns "we" and "our" aims at expressing the fact that the work and the opinions presented hereafter, reflect the mind of the whole Aster team.[1]

2.1 The Aster Development Environment

Started in 1995, the Aster activity aimed at contributing in the development of configuration-based distributed systems. In configuration-based programming, the application provides instructions to the development environment on how to configure a base execution platform so as to meet application needs. Pioneer projects in configuration-based programming like Conic [45], Polylith [57] and Durra [5], laid the foundations of the Aster framework [31]. From the Aster perspective, a distributed system consists of three parts: an application, a middleware layer, and a base execution and communication platform. Aster customizes middleware to meet the application requirements, as demonstrated by one of the early Aster papers [28]. In Aster we distinguish two types of application requirements: those concerning the *algorithmic* properties of the system, and those concerning the *quality* properties of the system. The algorithmic properties are related to the functionality of the services offered in the system (*e.g.* a queue should not be empty to perform a *pop()* operation on it), and the quality properties are related to the quality attributes characterizing the system functionalities (*e.g.* despite transient queue failures, a *push()* operation should be always possible). Aster appends to the contributions of the aforementioned pioneers in

[1] http://www.irisa.fr/solidor/work/aster.html

two ways. First, it separates the declaration of the algorithmic from the quality system properties, and uses formal specifications to describe the latter. Second, based on these formal specifications, it performs a specification matching process to identify the software components that should be used to customize the middleware layer.

2.1.1 The Aster Prototype

A prototype of the Aster development environment has been built for customizing middleware compatible with the OMG's ORB execution and communication platform (*e.g.* see [32]). This prototype implementation consists of three salient parts, which are graphically illustrates in Figure 2.1 and briefly presented below.

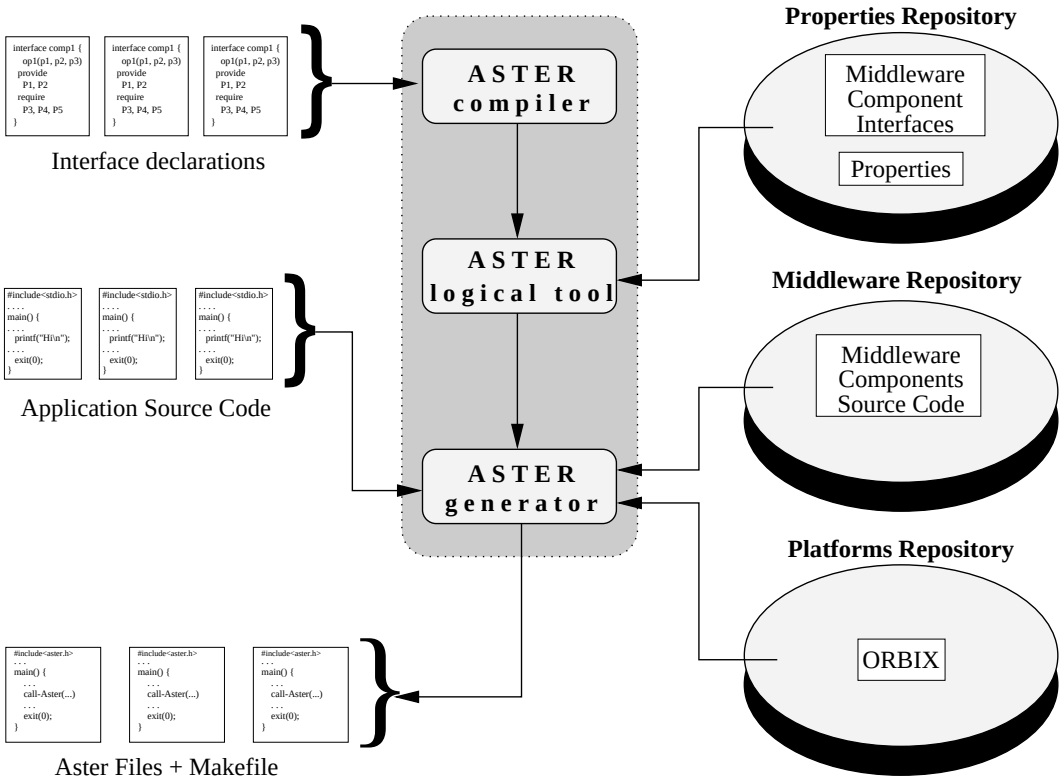


Figure 2.1: Salient parts of the Aster development environment.

1. The compiler parses the declarations of the application interfaces, where the requirements on quality properties are given explicitly. The result is a translation of these interfaces to CORBA-IDL interfaces and the extraction of the quality requirements which are used as input to the next part, which is the logical tool.

2. The logical tool performs a specification matching process, which takes as input the application's quality requirements and searches a database that contains the properties implemented by the reusable components found in the Aster software repository. The result is a set of software components that customize the middleware layer to meet the application requirements.

3. The generator takes the CORBA-IDL interfaces of the application components and of the software components selected by the logical tool, and produces the code that interconnects them to compose an integrated system. The result is an ensemble of C++ files and CORBA-IDL files, and a makefile that assembles them together to produce the executable representation of the system.

According to the above discussion, the Aster prototype has three characteristic features: *(i)* the employment of specification matching techniques based on the formal specifications of system constraints, *(ii)* the retrieval of software components for *automatic* middleware customization, and *(iii)* the compliance with the CORBA architecture. In the remainder of this section, a brief overview of these features is presented, assessing the initial choices and revealing their limitations.

2.1.2 Formalization and Specification Matching

One need early recognized in the Aster activity, was the one of rigorous specification of the application requirements and the properties provided by the software components used to customize the middleware layer. This was the result of the observation that the slightest misinterpretation of what is required and what is provided has severe consequences on the correctness of the system construction. To guarantee a unique and precise interpretation of the required and provided properties, Aster proposed their formal specification using first-order calculus and a small set of base predicates expressing system events like the exchange of messages and the occurrence of failures (see [29] for more information). Given the formal specification of the properties required by an application and the properties provided by components in the Aster software repository, the middleware customization process is transformed into a search of properties among the provided ones, that are at least as strong as the required ones. This search process is based on the *plug-in match* of specifications, which is inspired by well-known specification matching techniques proposed in the related literature (*e.g.* see [47, 53, 75]). Given the definition of the plug-in match of the set of properties P to the set of properties Q in formula (2.1), and for $P_{\mathcal{X}}$, $P_{\mathcal{MW}}$, and $P_{\mathcal{A}}$ being the properties provided by the base execution platform, the middleware, and the application respectively and $R_{\mathcal{X}}$, $R_{\mathcal{MW}}$, and $R_{\mathcal{A}}$ being the properties required by the execution platform, the middleware, and the application respectively, the middleware customization in Aster is captured by formula (2.2).

$$P \lhd_{plug-in} Q \;\equiv\; \forall Q_i \in Q | (\exists P_j \in P | P_j \Rightarrow Q_i) \tag{2.1}$$

$$(P_{\mathcal{X}} \cup P_{\mathcal{MW}} \cup P_{\mathcal{A}}) \lhd_{plug-in} (R_{\mathcal{X}} \cup R_{\mathcal{MW}} \cup R_{\mathcal{A}}) \tag{2.2}$$

The Aster logical tool, which performs the specification matching, is a combination of a simple theorem prover and a database linear search program. The input of the tool is a predicate-logic formula expressing one application requirement and the output is the first formula, or conjunction of formulas, in the Aster property database that is found equivalent or stronger than the input formula. The utilization of the logical tool gave some encouraging results for a first set of cases studied in the Aster prototype. However, a number of limitations were soon revealed, concerning both the formal basis and the implementation of the specification matching process. The logic vehicle was not adequate for describing properties related to the execution of the system, *i.e.* properties that include the notion of a sequence of system events. In addition, the time complexity of the theorem prover, albeit its frugal implementation, rendered prohibitive its use for formulas of medium size (*e.g.* ten terms connected with predicate logic operators). As a conclusion, the initial phase of the Aster activity has convinced us that the specification matching of the formally specified application requirements is an appealing way for the correct middleware customization. But the means employed in this process should be carefully designed and implemented to be of practical interest.

2.1.3 Software Retrieval

To perform middleware customization, Aster possesses a software repository of reusable components from which it selects those that customize the middleware to meet application needs. Initially, the software retrieval process based on the logical tool previously described, had not received much attention. A brute-force linear search was employed to find a set of software components whose properties form a plug-in match to the system requirements. This is acceptable for a small-sized property database, but when its size raises to few hundreds of formulas a necessity becomes evident for means to ameliorate the search process. On the other hand, we observed that the application requirements usually refer to high level properties, which are satisfied by combinations (conjunctions) of the properties implemented by software components. Studying deeper the application requirements concerning the qualities of security and dependability, we realized the existence of a conceptual hierarchy among the properties of each system quality. For example, an application may require a property as abstract as *failure masking*. This property can be analyzed to message *diffusion* and *active* replication. The latter can be further analyzed to *replication, atomic delivery,* and *voting*.

While it is rather difficult to find a software component satisfying directly the initial requirement, a plug-in match can be found by following the above analysis and combining the set of components implementing the properties *diffusion, replication, atomic delivery,* and *voting*. Hence, a need raised to replace the brute-force software retrieval with the employment of a rigorous codification of the properties conceptual hierarchy, to guide the selection of the software components that satisfy application requirements. Subsequently, the software repository needed to be organized accordingly so as to permit the efficient retrieval of the set of components corresponding to a search key (*i.e.* an application requirement). In addition, appropriate means had to be invented to describe the structure of the compound component satisfying the application requirement.

2.1.4 CORBA compliance

Although the Aster approach is conceptually independent from any middleware or execution and communication platform specificities, in the Aster prototype the tools related, in some way, to the system implementation, are specific to the OMG's CORBA[2] model [8]. The prototype is aimed at customizing the Orbix[3] implementation of OMG's ORB [4], and the components in the software repository are implemented as CORBA servers for Orbix. Besides, the system model is implicitly assumed to conform with the CORBA model, *i.e.* application is made of client objects invoking the methods exported by server objects, or the methods exported by OMG's *Common Object Services*[4] and *Common Facilities*[5]. Consequently, the Aster prototype is aimed for systems using RPC-like communication.

Following the CORBA model was an advantageous departure point, since a number of issues were automatically resolved by this compliance. For example, the only constraint for developing components to populate the software repository, was to implement them as CORBA servers. This relieved us from dealing with issues related to the design and implementation of reusable components, which are not directly related to the research interests in the Aster activity. In a similar way, the interaction patterns considered by the Aster prototype were limited to RPC and one-way communication (asynchronous message passing) patterns, which are the only ones supported by CORBA. Execution platform heterogeneity and executability issues were not even considered since they were neatly masked by the utilization of CORBA. However, the ambitions of the Aster project went beyond the development of CORBA compliant distributed systems. Soon, the attention in the Aster activity was also directed to the development of distributed systems using other communication patterns like event based and tuple space based interactions (see [33]), and to the quality properties related to interoperability issues (see [60]) raised by the execution platform heterogeneity. Consequently, we started searching for means to describe the quality properties of other system architectures besides CORBA-compliant ones.

2.2 The Evolution of Aster

The issues raised during the implementation and the utilization of the Aster prototype [11, 30, 60], stimulated the evolution of Aster. Beyond the limitations of the Aster prototype mentioned in the previous section, a significant observation caused the revision of the initial objectives. This observation concerned the fact that the customization results depend heavily on the design decisions, regarding how the system provides the qualities required in its specifications. The conclusion drawn was that one way to accomplish a successful customization is to guarantee that the system structure is correct with respect to the application requirements. And this correctness concerns the algorithmic constraints of the system as much as the constraints on system's qualities (*e.g.* dependability, security, timeliness, *etc*). In the remainder of this section, the main axes of the Aster evolution are

[2] http://www.omg.org/corba/c2indx.htm
[3] http://www.iona.com/products/orbix/index.html
[4] http://www.omg.org/corba/csindx.htm
[5] http://www.omg.org/corba/cfindx.htm

presented, emphasizing on those related to the system design and its systematic transformation into a correct system implementation.

2.2.1 Multiple Viewpoints on System Properties

The first issue that became clear in our involvement with the Aster prototype was the fact that the algorithmic and the various quality system properties, interweave to influence the system structure. Still, we could analyze the properties concerning the same system quality (*e.g.* dependability), independently from properties concerning other system qualities (*e.g.* security) or the algorithmic system properties. For example, it is possible to study the algorithmic constraints concerning the file accesses in a file system without having to consider at the same time the constraints concerning the availability and reliability of the constituent file servers, and vice versa. This observation led us in employing the *"separation of concerns"* principle, to study the aspects related to different system requirements. As a result, we adopted a multi-view description of the system structure [34], where each viewpoint expresses the impact of a quality property on the system architecture.

To accommodate the aforementioned viewpoints in the description of a single system, our attention was focused on the definition of the frameworks that would permit us to analyze the various quality properties and describe their influence on the system structure. Our primary goals were: *(i)* to understand the properties related to each viewpoint, *(ii)* to define the adequate framework for expressing them and their impact on system behavior, and *(iii)* to identify their impact on the system architecture. Among the first viewpoints to be studied in the Aster project are those related to security [9] and dependability qualities [61]. The latter viewpoint laid the foundations for the in-depth study of the robust development of dependable systems, which is the subject of the thesis presented in this report.

2.2.2 Systematic and Robust Customization

Another issue that became clear in our involvement with the Aster prototype, was the difficulty to perform the middleware customization automatically. The difficulty concerned the incorporation of the selected software components in the architectural structure of the initial system. Initially, the customized interaction of two objects was achieved by interposing between them a single middleware component, or a small number of such, sequentially interconnected. However, the Aster experience with the middleware customization for various types of software systems (*e.g.* see [11, 28, 60, 63]) revealed the fact that the middleware of a customized interaction may have a complex structure. Consequently, the developer's intervention is necessary to assure that the middleware components are correctly interconnected among them and with the application. For example, this is the case of a middleware consisting of the components implementing the properties *diffusion, replication, atomic delivery,* and *voting*, in order to meet the needs of a software system that requires *failure masking*.

The aforementioned human intervention has to be accomplished according to a set of guidelines, whose correctness is previously verified in order to avoid the possible introduc-

tion of errors in the system structure. Such errors would endanger the correctness of the customization process, and thus the robustness of the system development. As a solution to this problem, we chose to associate to the analysis of the system requirements concerning quality properties the corresponding architectural impact. Hence, the search process for finding the software components for middleware customization with respect to a given requirement would also reveal the way that these components should be incorporated in the system architecture.

Another issue that influence the customization robustness, is the interference of the different middleware customizations, each with respect to application requirements concerning a different quality property. Although we are aware of the problems raised by the compatibility of the different customizations, we have not addressed it in the Aster context yet. Finally, to guarantee the customization robustness, one should first guarantee that the selected software components implement correctly the properties that they claim to provide. This is strongly related to the validation of the implementation of a software component with respect to its specifications, and it is not addressed in Aster. Instead, we assume that the implementation of the reusable components populating the software repository is validated before introducing them into the repository.

2.2.3 Software Architecture

The experience with the Aster prototype showed that design correctness plays a very important role in the robustness of the development process. If the blueprints of the software system are not correct, then no matter how rigorous the implementation phase is, the final product will doubtfully be a correct incarnation of the initial specifications. The development robustness is guaranteed if the elaborated system structure correctly reflects the system requirements. Based on this observation, the focus of the research in Aster has been oriented towards the field of software architecture [68], which studies the system structure in terms of participating architectural entities and their interconnections. More precisely, in the Aster project we search for an appropriate way to incorporate the system requirements regarding quality properties into architectural descriptions which, by default, reflect the algorithmic and interaction system properties. The employment of software architecture in Aster is based on:

- The formalization of quality properties used to assure the correct interpretation of system requirements, and their correct refinement into an elaborated architectural description of the middleware integration with the software system.
- The utilization of the elaborated system architecture as guidelines for the middleware customization process.
- Multiple viewpoints on the system characteristics, which are defined depending on the system property being specified in each case.

The first two points are common for all quality properties. The last point is property-specific. Each viewpoint may describe differently the architectural impact of a quality

property on the system structure. The Aster logic vehicle, commonly used by all viewpoints, permits to test the compatibility of the outcome of the various viewpoint analyses, and to provide useful input to the middleware customization process.

2.3 Discussion

The goal of this section is to summarize the Aster background that fostered the work presented in this thesis, to define the problems that this thesis attacks, and to delineate the paths followed for solving them. In addition, a clear line is drawn between what existed in the Aster project and what this thesis adds to it.

Aster contributes to the development of distributed system by supporting the systematic middleware customization, which adapts the execution platform to the application requirements. The means used to achieve the systematic middleware customization are the formalization of the quality system properties (*e.g.* dependability, security, timeliness, *etc*), the automatic software retrieval based on a specification matching technique, and the systematic integration of the retrieved middleware components in the system architecture. Aster proposes a multi-view description of the system requirements concerning quality properties, where each viewpoint corresponds to a different quality property. The prototype implementation of the Aster development environment consists of:

- A compiler, which parses the interfaces of the application objects; an interface contains the operations imported and exported by the object, along with the object's quality requirements on each operation.
- A logical tool, which searches in the Aster software repository for reusable middleware components; selected software components implement properties, which form a plug-in match to some application requirement.
- A generator, which takes the outcome of the parsing and the software retrieval processes, and produces the code that glues together the application objects with the customized middleware.

The contribution of this thesis to Aster is twofold. First, it provides an accurate definition of a generic system model. Second, it proposes a framework for the analysis of the dependability requirements, which results in the systematic selection of the fault tolerant mechanisms to be used for middleware customization. In addition, this framework provides the means to derive the guidelines for integrating the selected fault tolerant mechanism with the system architecture. The approach undertaken for the specification of this framework, evolves around the following axes:

- the profound study of the dependability properties and their formal specifications,
- the architectural impact of the dependability properties on the system specification, and

- the organization of the dependability properties in a classification scheme, which captures a refinement relation in terms of both the dependability properties and the associated software architectures.

It has been already mentioned that requirements concerning different quality properties are analyzed in the framework specified by different viewpoints. For example, the first quality property to be methodically studied in the Aster project is security [9]. The study has shown that in the security domain, the analysis needs are focused on the definition of higher level security policies that determine the result of the combinations of lower level security policies. Consequently, the framework for the requirements analysis from the security viewpoint, provides a logic vehicle for describing how to legally combine different security policies, and what are the policies resulting from these combinations. The framework for the dependability viewpoint, proposed by this thesis, is radically different. It is based on the refinement of the abstract dependability requirements into the concrete properties provided by existing fault tolerant mechanisms, and the provision of guidelines for incorporating the obtained fault tolerant mechanism into the system architecture. Hence, dependability framework focuses on the identification of the means that satisfy specific dependability requirements. In contrast, the security framework focuses on the results of the combination of the security constraints associated with different system entities.

The framework proposed by this thesis for the dependability viewpoint is not the *only* possible approach to the robust development of dependable systems. What the author asserts is that it is a correct approach offering a realistic, practical, and efficient solution for systems with a wide variety of dependability requirements. In addition, it is expandable in that the knowledge concerning the analysis of new dependability requirements is shown to be easily incorporated in the proposed classification scheme and used thereafter.

Chapter 3

Formalizing Dependability Properties

This chapter focuses on the specification of the generic system model adopted in the Aster project, and the formal framework employed for the analysis of the dependability requirements. The latter provides means to express the system behavior in the presence of failures and to identify the system entities that assure a set of given dependability properties. Using this formal framework the developer can transform a formal description of the system behavior in terms of abstract system states before and after a failure into a precise description of the set of software components and their interconnections, that are employed to enforce the given system behavior.

3.1 The System Model

To be practically beneficial for system development, the system model should satisfy two conditions: (*i*) be easy to understand and use, and (*ii*) be expressive enough to capture all of the properties related to the various system behaviors. The approach proposed by the author is founded on the classical state-based system, where system actions are modeled as state transitions. The logic vehicle chosen to describe the system behaviors is an extension of predicate logic with the *precedence* relation [38] specifying a partial order in which predicates are verified. Notice that the use of temporal logic relations is not indispensable for modeling the temporal precedence of the predicate. Indeed, means have been invented like *history lists*, which are employed by a number of approaches (*e.g.* see [15] and [71]) in order to avoid temporal relations for ordering the occurrences of events in a system and to remain purely first order logic. However, the author believes that temporal operators render the formulas more legible.

The remainder of this section is structured as follows. First the mathematical notations used in this report are presented. Then, the formal definitions of a system and the associated concepts (*e.g.* state, action, execution, *etc*) are given. The base predicates used to express system behaviors come after, followed by the formalization of the equivalence of system specifications. The definition of the refinement rule completes the presentation of the system model.

3.1.1 Basic Notations

Algebraic notations.

- Expression $P \times Q$ denotes the *Cartesian product* of sets P and Q.
- Expression $\mathcal{P}^Q$ denotes the *powerset* of set Q (*i.e.* $\mathcal{P}^Q = \{Q_i | Q_i \subseteq Q\}$).
- Expression $r : P \times Q$ denotes a relation r which associates variables from a set Q to values from a set P.
- Expression $f : P \to Q$ denotes a function f that assigns values from the set Q to variables from the set P. Expression $dom(f)$ denotes the *domain* of f (*i.e.* $dom(f) = P$), and expression $rang(f)$ denotes the *range* of f (*i.e.* $rang(f) = Q$).
- Expression $\langle a_\nu \rangle$ *of* X denotes a *sequence* of elements of type X, which is a function that maps the natural numbers (denoted by the symbol $\mathbb{N}$) to elements of the set X (*i.e.* $\langle a_\nu \rangle$ *of* $X \equiv a_\nu : \mathbb{N} \to X$).

Temporal logic notations.

- Symbol "$\prec$" denotes the binary predicate operator that defines the *temporal precedence* relation. For P and Q being first-order predicates, the expression $P \prec Q$ expresses the fact that there exists a point on the temporal axis where P is verified and until this point Q has never been verified.
- Symbol "$\Diamond$" denotes the unary predicate operator called *eventually*, which expresses the fact that a predicate will be verified some time in the future.
- Symbol "$\Diamond\!\!\!\!-$" denotes the unary predicate operator called *in the past*, which expresses the fact that a predicate was verified some time in the past.

3.1.2 Basic Definitions

- A *system* $\mathcal{S}$ is defined as the triplet (V, U, R), where V is a set of variables, U is the set of values including the undefined value denoted by the symbol "$\perp$", and R is the function that assigns to each variable the set of its legal values.

$$\mathcal{S} = (V, U, R),\ where\ (V \neq \emptyset) \wedge (\perp \in U) \wedge (R : V \to \mathcal{P}^U),\ i.e.\ \forall v \in V | R(v) \subseteq U$$

- For a system $\mathcal{S} = (V, U, R)$, the *legal mappings* is the set $\mathcal{M}(\mathcal{S})$ of functions that map each system variable to one of its legal values. A legal mapping is also called a system *state*, and is denoted by the greek letter σ. When more than one system appear in the same context, the name of the system is prefixed to the state, in order to distinguish among states belonging to different systems. For example, the notations $\alpha.\sigma$ and $\beta.\sigma$ refer to two distinct states belonging respectively to system α and β.

$$\mathcal{M}(\mathcal{S}) = \{M_i | (M_i : V \to U) \wedge (\forall v \in V | M_i(v) \in R(v))\}$$

- For a system $\mathcal{S}$, the *legal transitions* is the set $\mathcal{T}(\mathcal{S})$ of all functions that associate one system state to some other system state. A legal transition is also called a system *action*.

$$\mathcal{T}(\mathcal{S}) = \{T_i | T_i : M \rightarrow \mathcal{M}(\mathcal{S}) \wedge M \subseteq \mathcal{M}(\mathcal{S})\}$$

  There are two categories of system actions: the I/O actions which represent the exchange of information between the system and its surrounding environment, and the *internal* actions which correspond to intrinsic system transitions. Only the I/O actions are perceivable by an observer that belongs to the system's surrounding environment. There are two types of I/O actions, expressing the export and the import of information. For a system S interacting with the environment E by exchanging data D, $S.xport(E, D)$ denotes the export action and $S.mport(E, D)$ denotes the import action. The prefix $S.$ is omitted when the system in question is obvious in a given context.

- A *specification* of a system $\mathcal{S}$ is the triplet (M, M_0, T), where M is a subset of states, M_0 is the initial system state in M, and T is a subset of system actions that includes at least one action defined on the initial state and that the union of all states resulting from these actions equals the set M.

$$\Sigma(\mathcal{S}) = (M, M_0, T) \equiv M \subseteq \mathcal{M}(\mathcal{S}) \wedge M_0 \in M \wedge T \subseteq \mathcal{T}(\mathcal{S}) \wedge \exists T_i \in T | M_0 \in dom(T_i) \wedge \bigcup_{T_i \in T} rang(T_i) = M$$

- For the specification $\Sigma(\mathcal{S}) = (M, M_0, T)$, a state $M_x \in M$ is said to be *reachable by* an action $T_x \in T$, *iff* there exists a sequence of actions starting by T_x which brings the system to the state M_x.

$$T_x \rightsquigarrow M_x \equiv \exists M_1 \in M | M_1 = dom(T_x) \wedge \exists \{T_1, ..., T_n | T_i \in T\} | (T_x(M_1) = dom(T_1) \wedge \forall (i < n) | rang(T_i) = dom(T_{i+1}) \wedge T_n(...T_1(T_x(M_1))...) = M_x)$$

- An *execution* of a system $\mathcal{S}$ according to the specification $\Sigma(\mathcal{S})$ is a sequence of subsets of state transitions. All of the transitions in the first element of the sequence are defined for M_0, and all the domains of actions corresponding to some sequence element a_j are reachable by all actions corresponding to some preceding sequence element a_i where $i < j$.

$$\mathcal{X}(\Sigma(\mathcal{S}) = (M, M_0, T)) \equiv \langle a_\nu \rangle of \mathcal{T}(\mathcal{S}) | (\forall T_i \in a_0 | dom(T_i) = M_0) \wedge (\forall T_i \in a_i \ , \ \forall T_j \in a_j | (i < j \Rightarrow T_i \rightsquigarrow dom(T_j)))$$

  Notice that the above transitions are not necessarily system actions. Hence, an execution may or may not conform with the specification $\Sigma(\mathcal{S})$. The latter case covers the execution of a system that have experienced some failure event. A *correct*

execution is the one that conforms with the system specification $\Sigma(\mathcal{S})$ (*i.e.* where each transition is a system action in T).

$$\mathcal{X}^C(\Sigma(\mathcal{S}) = (M, M_0, T)) \equiv (\mathcal{X}(\Sigma(\mathcal{S})) = \langle a_\nu \rangle of \mathcal{T}(\mathcal{S})) \wedge (\forall a_i \in \mathcal{X}(\Sigma(\mathcal{S}))|a_i \subseteq T)$$

Conceptually, different sequence elements represent different steps in the system execution, where at each step all corresponding actions are performed but in an undefined order. System actions in different subsets are performed in the same order as defined by the indexes of the corresponding sequence elements. Sometimes, is it useful to refer to the first n steps of an execution. Given an execution $\mathcal{X} = \langle a_\nu \rangle$, its first n step are noted $\mathcal{X}_{\rightarrow n}$.

Notice that, by definition, system actions are deterministic, *i.e.* for each system state and system action defined on that state, the resulting system state is uniquely defined. However, the choice of the system action among possible alternatives for a given system state is not uniquely defined. Hence, the execution of a system is non-deterministic.

- The *trace* of the execution $\mathcal{X}(\Sigma(\mathcal{S}))$ is the sequence of subsets of system states, where the first element is the singleton $\{M_0\}$ and each of the other sequence elements is formed by the union of the ranges corresponding to system actions of the execution $\mathcal{X}$.

$$\mathcal{R}(\mathcal{X} = \langle a_\nu \rangle of \mathcal{T}(\mathcal{S})) \equiv \langle b_\nu \rangle of \mathcal{M}(\mathcal{S})|(b_0 = \{M_0\}) \wedge (\forall i > 0|b_i = \bigcup_{T_x \in a_{\nu-1}} rang(T_x))$$

3.1.3 Describing System Behavior

The behavior of a system that can be observed by its surrounding environment, is a function of the system's I/O actions, since these are the only interaction points with the system. Hence, a generic framework employed to describe the system behavior, should provide means to express at least the system's I/O actions. Such a framework can be based on the previously described system model, and the axiomatic definition of a predicate capturing the fact that the system has reached a given state. Based on this predicate, the predicates expressing the I/O system actions are defined. This generic framework can be extended with the definition of other predicates that facilitate the desired system analysis (*e.g.* predicates *init*, *exit*, *begin*, *commit*, and *abort* can be defined to facilitate the description of system initialization and termination, and the behaviors related to transactional properties). More formally, the predicates of the generic framework used in this report, are:

- The predicate expressing that a system is in state σ is introduced by the unary operator [ ], *i.e.* $[\sigma]$ is true when the system is in state σ.
- The predicate *export* expressing the I/O action performed by a system α when it sends to a system β the data D. The data D sent are a function of some α's

state preceding the I/O action, and if some β's state is a function of data D, the export action precedes this state. More formally, for Z being the set of all messages exchanged between systems α and β, and $M(\alpha)$ and $M(\beta)$ being the sets of states of systems α and β respectively, the *export* predicate is defined as:

$$export(\alpha, \beta, D) \equiv (\exists \alpha.\sigma | \Diamond [\alpha.\sigma]) \wedge (\exists f : M(\alpha) \rightarrow Z | D = f(\alpha.\sigma)) \wedge ((\exists \beta.\sigma | (\exists g : Z \rightarrow M(\beta) | \beta.\sigma = g(D))) \Rightarrow ([\alpha.\sigma] \prec [\beta.\sigma]))$$

NOTICE: $export(\alpha, \beta, D)$ is verified when the action $\alpha.xport(\beta, D)$ is being executed.

- The predicate *import* expressing the I/O action performed by a system β, when it receives the data D sent by a system α. The export of data D from α to β has been preceded, and some state of β after receiving D is a function of the received data. More formally, for Z being the set of all messages exchanged between systems α and β, and $M(\beta)$ being the set of states of system β, the *import* predicate is defined as:

$$import(\alpha, \beta, D) \equiv \Diamond export(\alpha, \beta, D) \wedge \exists \beta.\sigma | ((\exists g : Z \rightarrow M(\beta) | \beta.\sigma = g(D)) \wedge \Diamond [\beta.\sigma])$$

NOTICE: $import(\alpha, \beta, D)$ is verified when the action $\beta.mport(\alpha, D)$ is being executed.

This framework allows the description of various facts concerning the behaviors of systems. An interesting example is the definition of the equivalence of system specifications. Intuitively, for two system specifications to be considered equivalent they should have the same behaviors under the same conditions. Since the behavior of a system perceived by an external observer, can be uniquely characterized by its I/O actions, *"same behavior"* can be interpreted as *"same sequence of I/O actions"*. This has the impact that the executions of the two systems according to the specifications in question should be equivalent, since an external observer is not able to differentiate between them. More formally, the specification equivalence based on the execution equivalence is given below:

- Two executions $\mathcal{X}(\Sigma(\mathcal{S})) = \langle a_\nu \rangle$ and $\mathcal{X}'(\Sigma'(\mathcal{S}')) = \langle b_\nu \rangle$ are said to be *equivalent iff* the corresponding subsequences of I/O actions are identical. In other words, each I/O action that belongs to one execution, also belongs to the other, and more precisely, to the sequence element with the same cardinality as in the first execution. For io_x and io_y denoting two I/O actions (mport or xport indifferently), the execution equivalence is captured in the following formula:

$$\mathcal{X}(\Sigma(\mathcal{S})) \equiv \mathcal{X}'(\Sigma'(\mathcal{S}')) \Leftrightarrow ((io_x \in a_i \wedge io_y \in a_i) \Leftrightarrow (io_x \in b_i \wedge io_y \in b_i)) \wedge ((io_x \in a_i \wedge io_y \in a_j \wedge i < j) \Leftrightarrow (io_x \in b_k \wedge io_y \in b_l \wedge k < l))$$

NOTICE: the equivalence of two executions refers only to the system behavior perceived by an observer external to the system. This does not impose any particular relation between the traces of two equivalent executions.

- The specification $\Sigma(\mathcal{S})$ is said to be *subordinate to* the specification $\Sigma'(\mathcal{S}')$ *iff* for every execution $\mathcal{X}(\Sigma(\mathcal{S}))$ exists an equivalent execution $\mathcal{X}'(\Sigma'(\mathcal{S}'))$.

$$\Sigma(\mathcal{S}) \sqsubseteq \Sigma'(\mathcal{S}') \;\equiv\; \forall \mathcal{X}(\Sigma(\mathcal{S}))|(\exists \mathcal{X}'(\Sigma'(\mathcal{S}'))|\mathcal{X}(\Sigma(\mathcal{S})) \equiv \mathcal{X}'(\Sigma'(\mathcal{S}')))$$

- Two specifications $\Sigma(\mathcal{S})$ and $\Sigma'(\mathcal{S}')$ are said to be *equivalent iff* each one is subordinate to the other.

$$\Sigma(\mathcal{S}) \equiv \Sigma'(\mathcal{S}') \quad \Leftrightarrow \quad \Sigma(\mathcal{S}) \sqsubseteq \Sigma'(\mathcal{S}') \;\wedge\; \Sigma'(\mathcal{S}') \sqsubseteq \Sigma(\mathcal{S})$$

3.2 The Dependability Analysis Framework

To study the dependability requirements of a system it is essential to analyze the system into its constituents, identify those constituents where the dependability requirements apply, modify their specifications to meet these requirements, and then putting them back together to obtain the dependable version of the intial system. Then, coupling the system model presented in the previous section with the the capability to analyze a system into its constituents and composing the constituents to obtain the complete system provides the formal framework for the dependability analysis.

3.2.1 Recursive System Structure

In the definition of the system action, it was mentioned that the only points of interaction between a system and its surrounding environment are its I/O actions. During a system execution, these actions permit the surrounding environment to supply the input data to the system and to acquire the outcome of the system functions. The input data supplied by the surrounding environment can be the outcome of another system's execution and the output of a system may serve as the input to another system in the surrounding environment. Hence, by appropriately binding some input operations of a system specification with some output operations of some other systems' specification, a larger system specification is formed. Inversely, the larger system specification can be seen as a composition of the smaller systems specifications. In the remainder of this report, when decomposing a system specification into its constituent, the term *objects* is used to refer to the subsystems corresponding to the constituent specifications. So, intuitively a system consists of a set of objects, where each object is itself a system. Figure 3.1 illustrates graphically this nested structure of a system. From the user's viewpoint, system S is a single entity, which has four I/O actions in the presented execution. A closer examination of S however, reveals that it is composed from objects P and Q, which are in turn composed from objects ({A, B, C} and {D, E} respectively).

The utility of this recursive system structure from the standpoint of the dependability requirements analysis, is that it allows to refine the requirements placed on a given system into a set of constraints associated to its constituents. Then, by iteratively decomposing a system specification into its constituents and refining the constraints placed on each constituent, the specifications of the objects specific to the dependability requirements are revealed and their properties are rigorously specified.

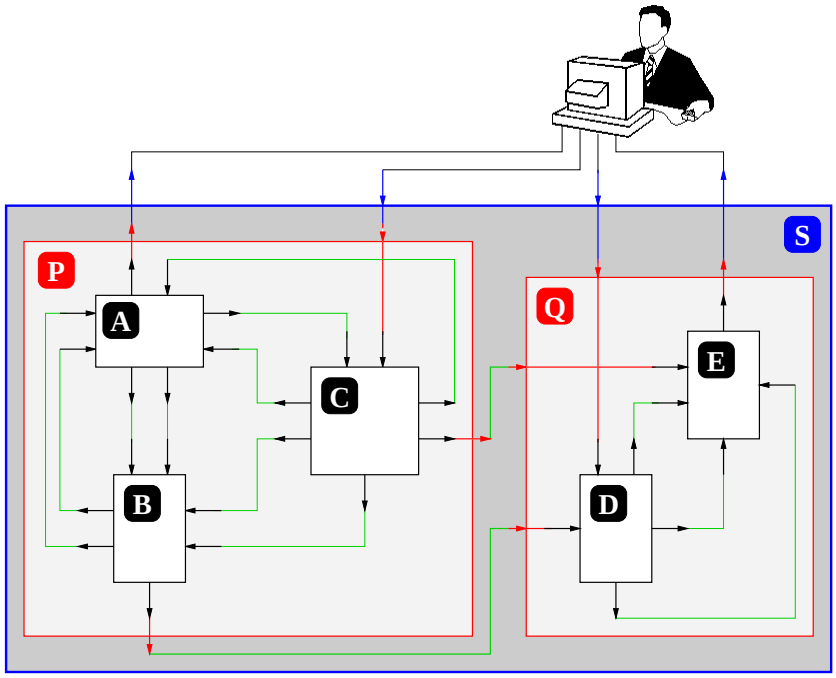


Figure 3.1: The recursive system structure.

3.2.2 Specification Composition

Following the intuitive image given by Figure 3.1, when two system specifications are composed, the bound I/O actions of each constituent specification become internal actions in the large system specification and the unbound I/O actions of the constituent specifications form the set of I/O actions for the larger system specification. In other words, the behavior of the composed system (in terms of I/O actions) as perceived by an observer external to it, should be exactly the same as the aggregate behavior of the two constituent systems, as perceived by an observer external to both systems. More formally, for $\Sigma_a(S_a) = (M^a, M_0^a, T^a)$ and $\Sigma_b(S_b) = (M^b, M_0^b, T_b)$ being two system specifications, their composition (denoted by the symbol "$\bowtie$") is equivalent to the system specification $\Sigma(S) = (M, M_0, T)$, under the following condition:

$$\begin{aligned}\Sigma_a(S_a) \bowtie \Sigma_b(S_b) \equiv \Sigma(S) \quad \textit{iff} \quad & (mport(a, D) \in T^b \Leftrightarrow xport(b, D) \in T^a) \ \wedge \\ & (mport(b, D) \in T^a \Leftrightarrow xport(a, D) \in T^b) \ \wedge \\ (((mport(\epsilon, D) \in T^a \vee mport(\epsilon, D) \in T^b) \wedge \epsilon \notin \{a, b\}) \Leftrightarrow & \ mport(\epsilon, D) \in T) \ \wedge \\ (((xport(\epsilon, D) \in T^a \vee xport(\epsilon, D) \in T^b) \wedge \epsilon \notin \{a, b\}) \Leftrightarrow & \ xport(\epsilon, D) \in T)\end{aligned}$$

System specifications composition is commutative and associative, which means that composition of a set of system specifications gives the same result independently of the composition order. More formally:

$$\Sigma_1(S_1) \bowtie \Sigma_2(S_2) \equiv \Sigma_2(S_2) \bowtie \Sigma_1(S_1), \quad and$$
$$\Sigma_1(S_1) \bowtie (\Sigma_2(S_2) \bowtie \Sigma_3(S_3)) \equiv (\Sigma_1(S_1) \bowtie \Sigma_2(S_2)) \bowtie \Sigma_3(S_3), \quad and$$
$$\bowtie_i \Sigma_i(S_i) \equiv \Sigma_1(S_1) \bowtie ... \bowtie \Sigma_n(S_n)$$

Specifications composition can be used to deduce the specifications of a system composed from a number of objects with known specifications. It can also be used as a criterion for the correct decomposition of a system specification into its constituents: the decomposition of a specification into its constituents is correct, *if and only if* the composition of the constituent specifications is equivalent to the initial system specification. This does not provide a method for the systematic decomposition of system specifications, but only a correctness criterion. The specification decomposition aiming at revealing the constituent sub-systems specific to the dependability properties of the initial system, should be guided by some other means. Such other means include the analysis results concerning the dependability requirements of the initial system, as proposed by this thesis. Once a fault tolerant mechanism appropriate for a given dependability requirement has been identified, and its interconnections with the initial system are fixed, its incorporation within the initial specification can be achieved using the composition rule given above.

The approach proposed in this report for guiding the dependability analysis, is to base the decomposition of the initial specification on the *refinement* of the properties that capture the dependability requirements of the system. A dependability requirement expresses some constraints on the system specification and by refining it, a stronger property is produced. Employing an appropriate iterative refinement process on the dependability requirements results in a set of properties that form an elaborated description of how the initial dependability requirements can be satisfied. These properties serve a double purpose: *(i)* they are the keys used for searching the specifications of the software components composing the desired fault tolerant mechanism, and *(ii)* they describe the interconnections among the constituents of the fault tolerant mechanism. The remainder of this section presents the author's proposition for specifying and refining the dependability properties in the Aster context. The utilization of the refinement results in the identification of the fault tolerant mechanism that corresponds to the dependability requirements and its incorporation in the software architecture of the initial system, are discussed in the next chapter.

3.2.3 Capturing Failure Events

At the definition of the system execution in page 21, the correct execution was distinguished from any system execution in general. It has been mentioned that an execution which does not conform with the system specification is an execution that has experienced some failure event and hence is not a correct one. The predicate *faulty* is introduced to capture

a failure event, and holds for the first state reached by the system which is the result of a transition that does not belong to the set of system actions. Given an execution $\mathcal{X}$ of the system specification $\Sigma = (M, M_0, T)$ and a state σ that belongs to the execution trace $\mathcal{R}(\mathcal{X}) = \langle b_\nu \rangle ofM$, the predicate *faulty* is formally defined as follows:

$$faulty_{\mathcal{X}}(\sigma) \equiv (\exists b_f \in \mathcal{R}(\mathcal{X}) | \sigma \in b_f) \wedge (\exists \mathcal{X}^C | (\mathcal{R}(\mathcal{X}^C) = \langle b_\nu^C \rangle \wedge \forall i < f | b_i = b_i^C)) \wedge \nexists T_x \in T | (\exists \sigma' \in b_{f-1} | T_x(\sigma') = \sigma)$$

Notice that the above predicate definition permits a state to be considered faulty during a given system execution, even if the state belongs to the set of the system states according to the given specification. This is absolutely legal since it permits to identify the failure where a system performs an operation that is defined in its specification but is different from the requested one (*e.g.* a system offering two services, integer addition and integer multiplication, and performs the addition of two integers for which their product has been requested). In addition, the above predicate captures the case of erroneous system mappings, *i.e.* mappings that do not belong at all to the system states designated by the system specification.

3.3 Analyzing Dependability Properties

This section exemplifies the use of the dependability analysis framework presented in the previous section. A number of dependability properties are specified and their refinements are given to reveal the essential constituents of a system that provides these properties. The formal specifications of these properties are an interpretation specific to the author's standpoint rather than a widely acknowledged characterization. The goal is not to impose the given interpretation but to use it as an example for the utilization of the formal framework for the analysis of the dependability properties.

3.3.1 Abstract Dependability Properties

The dependability properties associated to a system specification refer to the general behavior that the designer of the system expects in the presence of failures. These properties are in general too vague to serve as a key for the selection of the software components that should be used to compose the fault tolerant mechanism that guarantees the requested dependability properties. More likely, successive refinements should be performed to constraint the level of abstraction until the dependability properties are transformed into specifications of software components.

Based on the fault tolerance perspective introduced by Laprie [41], *Dependability* captures the quality of a system that allows a user to trust it despite the occurrence of failures. Attributes of *Dependability* are *Safety, Availability,* and *Reliability.*[1] The *Safety* property

[1] According to Laprie [41], dependability has a security attribute too. However, the author focuses mainly on the fault tolerance aspects of dependability. In the Aster context, these aspects are not related to the security attribute of dependability, which forms a domain of separate study (see [9]).

is verified by a system which, after a failure, enters a safe state where failure products are removed. The *Availability* property is verified by a system whose designated services remain available, despite the occurence of failures. Finally, a system verifies the *Reliability* property if it delivers the results to a service request, albeit the failures that may occur while the request is being served. Another dependability property that corresponds to a well-known fault tolerance abstraction is the *Masking* of failures, which expresses the fact that a failure event is not perceived by some part of the failed system's surrounding environment.

$$
\begin{array}{rcl}
\mathit{Dependability}(\Sigma) & \equiv & \forall \sigma \in \mathcal{R}(\mathcal{X}(\Sigma))|(\exists \sigma' \in \Sigma|[\sigma] \prec [\sigma']) \\
\mathit{Safety}(\Sigma) & \equiv & \forall \mathcal{R}(\mathcal{X})|((\sigma \in \mathcal{R}(\mathcal{X}) \wedge \mathit{faulty}_{\mathcal{X}}(\sigma)) \Rightarrow \\
 & & \exists \sigma' \in \mathcal{R}(\mathcal{X})|([\sigma] \prec [\sigma'] \wedge \neg \mathit{faulty}_{\mathcal{X}'}(\sigma'))) \\
\mathit{Availability}(\Sigma) & \equiv & \forall \mathcal{R}(\mathcal{X})|((\sigma \in \mathcal{R}(\mathcal{X}) \wedge \mathit{faulty}_{\mathcal{X}}(\sigma)) \Rightarrow \\
 & & \exists \sigma' \in \mathcal{R}(\mathcal{X})|(\sigma' \in \mathcal{R}(\mathcal{X}^C) \wedge ([\sigma] \prec [\sigma'])) \\
\mathit{Reliability}(\Sigma) & \equiv & \forall \mathcal{R}(\mathcal{X}(\Sigma))|\ \exists \Sigma'|((\Sigma' \equiv \Sigma) \wedge ((\sigma \in \mathcal{R}(\mathcal{X}(\Sigma)) \wedge \mathit{faulty}_{\mathcal{X}}(\sigma)) \Rightarrow \\
 & & ((\exists \sigma' \in \mathcal{R}(\mathcal{X}(\Sigma))|[\sigma] \prec [\sigma']) \wedge (\exists \sigma'' \in \mathcal{R}(\mathcal{X}^C(\Sigma'))|\sigma'' \subseteq \sigma')))) \\
\mathit{Masking}(\Sigma) & \equiv & ([\sigma] \wedge \mathit{faulty}_{\mathcal{X}}(\sigma)) \Rightarrow \exists \sigma' \in \mathcal{R}(\mathcal{X})|(([\sigma] \prec [\sigma']) \wedge \exists \sigma'' \in \mathcal{R}(\mathcal{X}^C)|\sigma'' \subseteq \sigma'))
\end{array}
$$

Table 3.1: Some abstract dependability properties

Table 3.1 gives the formal expressions of the dependability properties mentioned above. To render the formulas more legible, the notation $\sigma \in \Sigma$ is used to express the fact that a state σ belongs to the set of states defined by the specification Σ. For the same reason, subscripts are neglected when this does not introduce ambiguities. First, the formalization of *Dependability* expresses the fact that, independently of failure occurences, a system progresses and reaches a state described in its specifications (there is always a *"next"* state in an execution trace, which conforms to the system specification). The specification of *Safety* expresses the fact that, after a failure, the system reaches a state where the *faulty* predicate is no longer true. The specification of *Availability* states that, after a failure, the system reaches a state that belongs to the trace of the correct system execution. In a similar way, the specification of *Reliability* states that, each faulty state in the execution trace of the system is followed by a state which is a superset of some state in the trace of the correct execution of some equivalent system (*i.e.* even if failure products are not removed, the behavior expected by a correct system execution will still be observed). Finally, the specification of *Masking* captures the fact that a faulty state is followed by a state which is a superset of some state in the trace of the correct execution of the *same* system.

The reader may notice that all formulas in Table 3.1 express properties that refer to a single system, without mentioning anything about the properties of its surrounding envi-

$$
\begin{array}{rcl}
\mathit{DetectionObj}(\Sigma, \Sigma_\alpha) & \equiv & ([\sigma] \wedge \mathit{faulty}(\sigma)) \Rightarrow ((\mathit{xport}(\epsilon, f(\mathit{failure})) \in \Sigma_\alpha) \wedge \\
& & \qquad ([\sigma] \prec \mathit{xport}(\alpha, \epsilon, f(\mathit{failure}))))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{MaskingObj}(\Sigma_\alpha) & \equiv & (\alpha.\sigma \in \Sigma_\alpha \wedge [\alpha.\sigma] \wedge \mathit{faulty}(\alpha.\sigma)) \Rightarrow (\exists \Sigma_\beta | ((\Sigma_\beta \equiv \Sigma_\alpha) \wedge \\
& & m = max\{n \in \mathbb{N} | \mathcal{X}_{\rightarrow n}(\Sigma_\alpha) \equiv \mathcal{X}^C_{\rightarrow n}(\Sigma_\alpha)\} \wedge (\mathcal{X}_{\rightarrow m}(\Sigma_\alpha) \equiv \mathcal{X}_{\rightarrow m}(\Sigma_\beta)) \wedge \\
& & \qquad (\exists m' | (m < m' \wedge (\mathcal{X}_{\rightarrow m'}(\Sigma_\beta) \equiv \mathcal{X}^C_{\rightarrow m'}(\Sigma_\beta))))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Diffuse}(\Sigma_\alpha, G) & \equiv & \exists G = \{\Sigma_\alpha, \Sigma_{\alpha_1}, ..., \Sigma_{\alpha_N}\} \mid ((\Sigma_\beta \in G \wedge \mathit{xport}(\alpha, \beta, D)) \Rightarrow \\
& & \qquad \forall \Sigma_{\beta_i} \in G | \mathit{mport}(\alpha, \beta_i, D))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Active}(\Sigma_\alpha, N) & \equiv & \exists G = \{\Sigma_\alpha, \Sigma_{\alpha_1}, ..., \Sigma_{\alpha_N}\}, \exists \Sigma_\beta \mid ((\forall \Sigma_i, \Sigma_j \in G | (\Sigma_i \equiv \Sigma_j)) \wedge \\
& & ((\mathit{mport}(\alpha_i, \beta, d_i) \wedge \mathit{mport}(\alpha_j, \beta, d_j) \wedge (\Sigma_{\alpha_i} \in G) \wedge (\Sigma_{\alpha_j} \in G) \wedge \\
& & \qquad (\mathit{id}(d_i) = \mathit{id}(d_j))) \Rightarrow \exists ! \mathit{xport}(\beta, \epsilon, d) | \mathit{id}(d) = \mathit{id}(d_i)) \wedge \\
& & (((\Sigma_\alpha \in G \wedge \mathit{mport}(\epsilon, \alpha, d)) \Rightarrow \forall \Sigma_{\alpha_i} \in G | \mathit{mport}(\epsilon, \alpha_i, d)) \wedge \\
& & \qquad ((\Sigma_\alpha \in G \wedge (\mathit{mport}(\epsilon, \alpha, d_1) \prec \mathit{mport}(\epsilon, \alpha, d_2))) \Rightarrow \\
& & \qquad\quad \forall \Sigma_{\alpha_i} \in G | (\mathit{mport}(\epsilon, \alpha_i, d_1) \prec \mathit{mport}(\epsilon, \alpha_i, d_2)))))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Semi\text{-}Active}(\Sigma_\alpha, N) & \equiv & \exists G = \{\Sigma_\alpha, \Sigma_{\alpha_1}, ..., \Sigma_{\alpha_N}\} \mid ((\forall \Sigma_i, \Sigma_j \in G | (\Sigma_i \equiv \Sigma_j)) \wedge \\
& & (\exists ! \Sigma_l \in G | ((\Sigma_\alpha \in G) \wedge (\Sigma_\epsilon \notin G) \wedge \mathit{xport}(\alpha, \epsilon, d)) \Rightarrow (\Sigma_\alpha = \Sigma_l)) \wedge \\
& & (((\Sigma_\alpha \in G \wedge \mathit{mport}(\epsilon, \alpha, d)) \Rightarrow \forall \Sigma_{\alpha_i} \in G | \mathit{mport}(\epsilon, \alpha_i, d)) \wedge \\
& & \qquad ((\Sigma_\alpha \in G \wedge (\mathit{mport}(\epsilon, \alpha, d_1) \prec \mathit{mport}(\epsilon, \alpha, d_2))) \Rightarrow \\
& & \qquad\quad \forall \Sigma_{\alpha_i} \in G | (\mathit{mport}(\epsilon, \alpha_i, d_1) \prec \mathit{mport}(\epsilon, \alpha_i, d_2)))) \wedge \\
& & ((\exists \Sigma_x \in G | \mathit{mport}(\epsilon, x, f(\mathit{l'sfailure}))) \Rightarrow \\
& & \exists ! \Sigma_{l'} \in G | ((\Sigma_\alpha \in G) \wedge (\Sigma_\epsilon \notin G) \wedge \mathit{xport}(\alpha, \epsilon, d)) \Rightarrow (\Sigma_\alpha = \Sigma_{l'})))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Passive}(\Sigma_\alpha) & \equiv & \exists \Sigma_\beta, \Sigma_\gamma \mid ((\Sigma_\alpha \equiv \Sigma_\beta) \wedge \\
& & (\forall \mathcal{X}(\Sigma_\gamma) | \mathcal{X}(\Sigma_\gamma) \equiv \mathcal{X}^C(\Sigma_\gamma)) \wedge (\exists f : M(\alpha) \rightarrow Z | d = f(\alpha.\sigma)) \wedge \\
& & \qquad \mathit{mport}(\alpha, \gamma, d) \wedge ([\alpha.\sigma] \prec \mathit{xport}(\alpha, \gamma, d)) \wedge \\
& & (\mathit{mport}(\epsilon, \beta, f(\alpha' \mathit{sfailure})) \Rightarrow (\exists g : M(\alpha) \rightarrow Z | d' = g(\alpha.\sigma)) \wedge \\
& & \qquad ([\alpha.\sigma] \prec \mathit{mport}(\gamma, \beta, d')) \wedge \forall \Sigma_\epsilon | (([\alpha.\sigma] \prec \mathit{xport}(\epsilon, \alpha, D)) \Rightarrow \\
& & \qquad\quad \mathit{mport}(\epsilon, \beta, D))))
\end{array}
$$

Table 3.2: Refining the property *Reliability*

ronment or of its constituent objects. By strengthening the property *Masking*, its impact to the architectural structure of the system is revealed. One way to strengthen *Masking* is by combining the properties *DetectionObj* and *MaskingObj* given in Table 3.2. The former property expresses the fact that there exists an object in the system specification which, in the case of a failure, sends a message containing a notification about the failure event. The latter property expresses the fact that the failure of a given object can be masked by an object with equivalent specification, which has performed until the failure event an execution equivalent to the one of the failed object, and it continues its execution after the point of the first object's failure. These two properties (*DetectionObj* and *MaskingObj*) have an impact on the architectural structure of the system to which they are assigned. Both of them are defined for a given system specification Σ_α but they refer to some other system

specification Σ_β which interacts with the former. So, for a system specification Σ_α to be consistent with *DetectionObj* and *MaskingObj*, it should possess at least the I/O actions imposed by those two properties (*i.e.* the xport action referred in the case of *DetectionObj* and the set of I/O action that assure the equivalence of system execution in the case of *MaskingObj*).

Another way to stengthen *Masking* is by combining the properties *Diffuse* and *Active* given in Table 3.2. The former property expresses the fact that a message sent to a member of a multicasting group will be eventually received by all members of that group. The latter property expresses the semantics of the replication scheme that follows the *State Machine Approach* [65]. In brief, it states that all members of group G receive the same input in the same order, and there exists a component β which filters the identical outputs of group members to produce a single output exported to the environment. Similarly to the *DetectionObj* and *MaskingObj*, these two properties have an impact on the system structure too (*e.g.* the component β introduced by *Active*). Likewise, the last two properties defined in Table 3.2, *Semi-Active* and *Passive* which indicate two ways to strengthen *MaskingObj*, compel modifications of the system structure. These two properties express the well-known fault tolerance techniques of semi-active and passive replication respectively. Each of these properties introduces a number of sub-system specifications, besides those given in its arguments. Table 3.3 presents some properties that further refine the *Active, Semi-Active,* and *Passive* refinements of *Reliability*. In both Table 3.2 and Table 3.3, symbol ϵ denotes any object in the system, and Z denotes the set of messages exchanged by the objects in a given context.

$$
\begin{array}{rcl}
\mathit{Replication}(G) & \equiv & G = \{\Sigma_i\}_i \wedge \forall \Sigma_\alpha, \Sigma_\beta \in G | (\Sigma_\alpha \equiv \Sigma_\beta) \\
\\
\mathit{Filter}(G) & \equiv & G = \{\Sigma_i\}_i \wedge \exists \Sigma_\beta | ((\mathit{mport}(\alpha_i, \beta, d_i) \wedge \mathit{mport}(\alpha_j, \beta, d_j) \wedge (\Sigma_{\alpha_i} \in G) \wedge \\
& & (\Sigma_{\alpha_j} \in G) \wedge (\mathit{id}(d_i) = \mathit{id}(d_j))) \Rightarrow \exists ! \mathit{xport}(\beta, \epsilon, d) | \mathit{id}(d) = \mathit{id}(d_i)) \\
\\
\mathit{AtomicDelivery}(G) & \equiv & G = \{\Sigma_i\}_i \wedge ((\Sigma_\alpha \in G \wedge \mathit{mport}(\epsilon, \alpha, d)) \Rightarrow \forall \Sigma_{\alpha_i} \in G | \mathit{mport}(\epsilon, \alpha_i, d)) \wedge \\
& & ((\Sigma_\alpha \in G \wedge (\mathit{mport}(\epsilon, \alpha, d_1) \prec \mathit{mport}(\epsilon, \alpha, d_2))) \Rightarrow \\
& & \forall \Sigma_{\alpha_i} \in G | (\mathit{mport}(\epsilon, \alpha_i, d_1) \prec \mathit{mport}(\epsilon, \alpha_i, d_2))) \\
\\
\mathit{Leader}(G) & \equiv & G = \{\Sigma_i\}_i \wedge \exists ! \Sigma_l \in G | ((\Sigma_\alpha \in G) \wedge (\Sigma_\epsilon \notin G) \wedge \mathit{xport}(\alpha, \epsilon, d)) \Rightarrow \Sigma_\alpha = \Sigma_l \\
\\
\mathit{StableStorage}(\Sigma_\alpha, \Sigma_\gamma) & \equiv & (\forall \mathcal{X}(\Sigma_\gamma) | \mathcal{X}(\Sigma_\gamma) \equiv \mathcal{X}^C(\Sigma_\gamma)) \wedge (\exists f : M(\alpha) \rightarrow Z | d = f(\alpha.\sigma)) \wedge \\
& & \mathit{mport}(\alpha, \gamma, d) \wedge ([\alpha.\sigma] \prec \mathit{xport}(\alpha, \gamma, d)) \\
\\
\mathit{Restore}(\Sigma_\alpha, \Sigma_\beta, \Sigma_\gamma) & \equiv & (\exists f : M(\alpha) \rightarrow Z | d = f(\alpha.\sigma)) \wedge \mathit{mport}(\gamma, \beta, d) \wedge ([\alpha.\sigma] \prec \mathit{mport}(\gamma, \beta, d)) \wedge \\
& & \forall \Sigma_\epsilon | (([\alpha.\sigma] \prec \mathit{xport}(\epsilon, \alpha, D)) \Rightarrow \mathit{mport}(\epsilon, \beta, D))
\end{array}
$$

Table 3.3: Some more refinements of the property *Reliability*

3.3.2 Property Refinement

The interpretation of the english terms associated to the properties presented above, and their formal specifications reflect only the author's perspective rather than a widely accepted characterization. The specifier is free to choose the term that seems more adequate for each property and to be more or less precise on its specification, following another perspective on the refinement of a property. This is more a question of the specifier's style, and hence subjective. What is important is to transform a property like *Dependability*, which expresses a very abstract constraint on the behavior of a software system in the presence of failures into a set of properties like *Filter*, which express much more concrete constraints on the behaviors and the interconnections of the software system constituents.

Such transformations are based on successive property refinements, starting from the abstract constraint and ending up with a set of concrete properties that can be used as the software specification of the components composing the fault tolerant mechanism that should be incorporated to the initial system to satisfy its dependability requirements. In addition, it is more convenient if the intermediate refinements produce properties that represent well understood abstractions engaged in the conception and construction of fault tolerant mechanisms. Intuitively, this would help in the reuse of intermediate refinements in the analysis of more than one dependability constraint and the identification of the specifications of more than one fault tolerant mechanism. For example, the *Replication* property is used in the refinement of the properties *Active, Semi-Active* and *Passive*.

Combining the contents of Tables 3.1, 3.2, and 3.3, a number of interrelated refinement patterns are revealed. The *Dependability* property can be refined to the *Safety, Availability,* or *Reliability* property. The *Reliability* property can then be refined to the *DetectionObj* and the *Masking* properties and so forth. Table 3.4 gives a synopsis of the refinement patterns related to the dependability properties discussed so far. Symbol "$\rightarrow$" denotes the refinement of the property in the left hand side into the property on the right hand side, and symbols "|" and "+" are used in the right hand side expressions to denote respectively the disjunction and the conjunction of properties they separate.

The form in which the refinement patterns are expressed in Table 3.4 resembles to a set of grammar rules (see [42] for an introduction on grammars), each yielding a stronger expression of the property given in its left hand side. Indeed, let T be the set of all properties for which there are no refinement patterns to transform them into a stronger form (*e.g.* properties *Filter, Restore, StableStorage, etc*), and let S be the set of all properties mentioned in Table 3.4 (*e.g.* properties *Dependability, Availability, Masking, Active, etc*). Let also R be the set of rules stemming from the refinement patterns that transform properties from S into stronger properties from S, and let *Dependability* be the weakest property in S that may potentially yield any element in T. Then, the quadruple $G = \langle S, T, R, \mathit{Dependability} \rangle$ defines a grammar in the given context.

The aforementioned grammar captures a certain knowledge on the analysis of dependability requirements placed upon a software system. Such a grammar is extensible with respect to all three sets S, T and R. The specification of a new property representing some well understood dependability abstraction, extends the symbol set S with a new element. The refinement patterns that show how to obtain this property from more abstract proper-

$$
\begin{array}{rcl}
\mathit{Dependability}(\Sigma) & \rightarrow & \mathit{Safety}(\Sigma) \mid \mathit{Availability}(\Sigma) \mid \mathit{Reliability}(\Sigma) \\
\mathit{Reliability}(\Sigma) & \rightarrow & \mathit{Masking}(\Sigma) \\
\mathit{Masking}(\Sigma) & \rightarrow & \mathit{MaskingObj}(\Sigma) + \mathit{DetectionObj}(\Sigma) \mid \mathit{Active}(\Sigma, N) + \mathit{Diffuse}(\Sigma_\epsilon, G) \\
\mathit{MaskingObj}(\Sigma_\alpha) & \rightarrow & \mathit{Semi\text{-}Active}(\Sigma_\alpha, N) \mid \mathit{Passive}(\Sigma_\alpha) \\
\mathit{Active}(\Sigma_\alpha, N) & \rightarrow & \mathit{Replication}(G) + \mathit{Filter}(G) + \mathit{AtomicDelivery}(G) \\
\mathit{Semi\text{-}Active}(\Sigma_\alpha, N) & \rightarrow & \mathit{Replication}(G) + \mathit{Leader}(G) + \mathit{AtomicDelivery}(G) \\
\mathit{Passive}(\Sigma_\alpha) & \rightarrow & \mathit{Replication}(\{\Sigma_\alpha, \Sigma_\beta\}) + \mathit{StableStorage}(\Sigma_\alpha, \Sigma_\gamma) + \mathit{Restore}(\Sigma_\alpha, \Sigma_\beta, \Sigma_\gamma)
\end{array}
$$

Table 3.4: Refinement patterns represented as grammar rules

ties expressing dependability constraints and how to strengthen this property, extend the set R with new grammar rules. Finally, if there are no rules having the new property at the left hand side (*i.e.* no rules for strengthening the new property), the set of terminals T is extended by a new element. Hence, the grammar introduced above is not restricted to the dependability properties mentioned thus far, nor to the adopted specifications. This grammar forms the basis for the systematic analysis of the dependability requirements discussed in the next chapter. More precisely, the important feature of this grammar, used for the systematic dependability analysis, is the fact that it introduces a hierarchy among the dependability properties represented by the elements of its set of symbols S.

Figure 3.2 illustrates this hierarchy in form of a graph, where each node represents a different dependability property. The graph capturing the hierarchy among the dependability properties is a directed acyclic graph (DAG), and more precisely, a *tree*. The particularity of the tree depicted in Figure 3.2, is that it distinguishes the *childs* of a node from its *successors*. A successor of a node is itself a node and represents a single dependability property. A child of a node is a root of a sub-tree rooted on that node, and captures a refinement of the property represented by the given node. In Figure 3.2, childs are depicted by the small squares that are attached underneath the nodes. Hence, a child might correspond to a single link towards one successor node (*e.g.* the child of the node representing *Reliability* in Figure 3.2), or to a set of links towards multiple successor nodes (*e.g.* the child of the node representing *Active* in Figure 3.2).

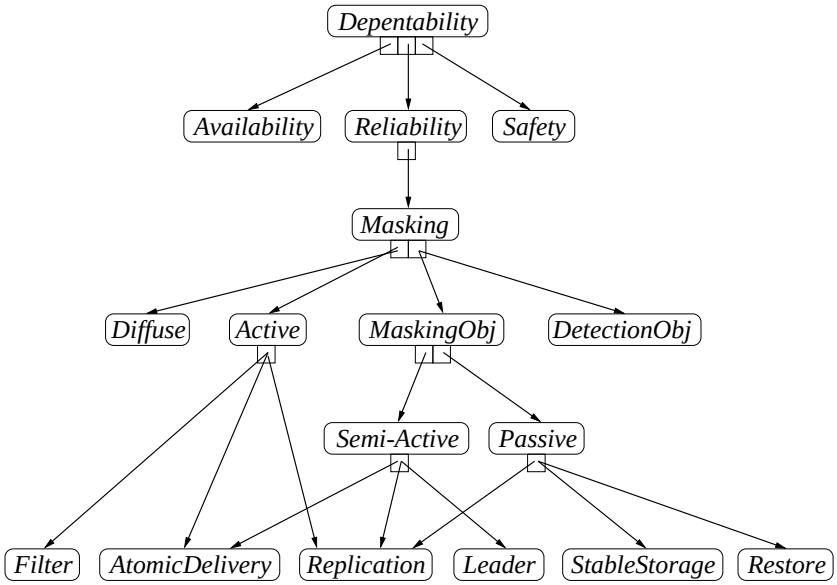


Figure 3.2: A hierarchy of dependability properties

3.4 Evaluation

The formal framework presented in this chapter, provides the basis for giving the specification of a system and describing its associated dependability properties. The refinement rule presents a means for transforming an initial system specification and its associated property into a more elaborated description of the system composition and the properties of its constituent. In this concluding section, the rationale behind the choice to define a minimal logic vehicle for the dependability analysis is revealed. In addition, the need for the presented formal framework is justified through a brief comparison with the most significant work related to formal specification of properties related to dependable systems.

3.4.1 Minimal Formal Framework

Using formal methods to specify system properties and reason about their truthfulness is not a recent research domain. A number of logic variants like predicate calculus, set theory, and temporal logic have served as the basis for developing new process algerbas (*e.g.* CSP [27] and CCS [48]), specification methods and languages (*e.g.* LOTOS [73], SDL [18], and Z [70]), and theorem provers (*e.g.* LP[2], STeP[3], and CoQ[4]), making the domain of formal methods in computing a fairly rich one (see the relevant section in the

[2]http://www.sds.lcs.mit.edu/spd/larch/
[3]http://theory.stanford.edu/people/zm/step.html
[4]http://coq.inria.fr/assis-eng.html

WWW Virtual Library[5]). Given this plurality of available tools and techniques, a question raises naturally about the necessity of yet another formal framework for specifying system properties and reasoning about their refinements.

One way to justify a new formal framework is to prove that the problems in the context where it is employed are not covered by existing methods. This is clearly not the case with the framework presented in this chapter. Lamport's TLA [40] largely covers the need for temporal ordering of system events, the SDL hierarchical system model is a proper superset of the one defined in this chapter, and existing theorem provers like STeP and LP are sufficiently powerful to provide mechanical support for reasoning on the system properties. However, a single existing method cannot completely replace the presented formal framework. TLA does not explicitly deal with I/O operations and is best suited for closed system models (*i.e.* models of systems that do not interact with their environment). In the same way, SDL does not provide for temporal ordering of system events, and theorem provers do not provide any support for describing system structure. Hence, although the proposed formal framework is not innovative in its parts, it is an assembly of parts of existing formal methods in a way that fits best the needs of dependability analysis in the Aster context.

The advantage with this assembly is that it can be replaced partially or completely by different combinations of existing formal methods, without affecting the validity of the dependability analysis presented in the following chapter. The system developer is free to choose which parts of the formal framework will be replaced and by which existing formal method. For example, TLA may replace the logic vehicle and the system model can be substituted by extended finite state machines. Another alternative is to combine SDL, Lotos, and Lotos tools to obtain a framework providing at least the same functionality as the proposed one. In addition, existing process algebras can be used to describe some parts of the specified system. For example, CSP can be used to capture the communication patterns of the system constituents and CCS can be used to describe the externally observed system behavior. Hence, the formal framework introduced in this chapter should not be seen as an attempt to define a new formal method, more adequate than existing ones in dealing with the analysis of dependability constraints. Rather, it is a composition of features of existing formal methods and gives the system developer the freedom to replace them with the formal methods that suit best the specificities of a given system and its development environment.

3.4.2 Related Work

The need to master the conceptual complexity of analyzing applications' dependability requirements, has fostered the research towards the identification of abstractions for modeling fault tolerance software properties. The work cited in this section is only a representative sample of the research results published in numerous conferences and journals. The goal is to show the evolution on the analysis and the comprehension of the fault tolerance abstractions employed in the development of dependable software, and to explain why ex-

[5] http://www.comlab.ox.ac.uk/archive/formal-methods.html

isting specification means were not adequate for the needs of frameworks focusing at the middleware customization, and Aster in particular.

Examples of work focusing on the identification of commonly used fault tolerance abstractions like Agreement, Atomic Actions, Failure Detection, Membership, Resilient Processes, Stable Storage, State Machine and others, are the work of Mishra and Schlichting [50], and Schneider [64]. In addition, the dependencies among these abstractions and the results of some of their combinations are discussed, producing *common patterns* that indicate how and for which cases these abstractions can be employed. Such approaches that are not based on formal specifications, lack the flexibility to adapt to the introduction of new abstractions or to changes in their initial assumptions caused by the evolution of the fault tolerant mechanisms and the applications' dependability requirements.

A four-layer architecture and a set of formally specified refinement patterns are presented by Di Vito and Butler [74]. The highest level offers an ultra-reliable uniprocessor model and its formal specifications serve to verify that the dependability requirements of the application conform with the uniprocessor semantics. If no conflicts are detected, the refinement patterns describe how to transform the abstract dependability constraints into a fault tolerant software system based on the active replication abstraction. This approach supports the partial analysis of the applications' dependability requirements only in the case where they conform with the State Machine semantics [65], in contrast to the presented framework, which is not restricted only to one abstraction. This is because Di Vito and Butler concentrate on the analysis of the dependability requirements for which a fixed set of refinements may apply, and not in the study of dependability constraints in general, as does the approach presented in this chapter.

A refinement technique based on relatively correct refinement steps is proposed by Cau and de Roever [13]. Despite the apparent similarity with the presented approach, there is a number of fundamental differences. This technique is based on the rather complex *dense linear time logic*, which is a combination of temporal logic, set theory, history lists and a system model based on state transitions. In addition, besides the last refinement which is correct with respect to the initial assumptions, all the other refinements are relatively correct for a restricted set of assumptions and the final correct refinement is obtained by refining the intermediate assumptions to meet the initial ones. On the other hand, the logic vehicle proposed in this chapter, is fairly simpler and the refinement process does not produce incorrect intermediate results.

The approach presented in this chapter has been inspired by the analysis of hierarchical software construction presented by Cristian [16]. In his approach, dependable software consist of layers of services, where the correct functioning of the software components implementing a given layer *depends upon* the correct functioning of the software components implementing the underpinning layer. The presented analysis of the dependability constraints describes a similar layered structure, where the specifications of software components are revealed in a descending hierarchy order, *i.e.* the specification of a software component is subsequent to the specifications of the software components that *depend upon* it. The presented approach extends Cristian's approach, by proposing a formal framework, which permits the specification of dependability properties. Consequently, it supports the verification of the correctness of the *depends upon* relations, which correspond to the hierar-

chical construction of the fault tolerant mechanism that meets application's dependability requirements.

3.4.3 Epilogue

This chapter presented the formal model for the specification of the various system requirements and its extension for the formalization of the dependability properties, which provides the means for the rigorous analysis of the system's dependability requirements. The formal model is simple and comprehensible since it is based on states and actions and an extension of the first-order logic with the precedence operator [38]. Additionaly, the formal framework for the analysis of the system dependability requirements is equally comprehensible and minimal since it is based on the definition of a single additional predicate (*i.e.* *faulty*) for expressing the occurrence of failure events. This results in a handy means for the system developer who can specify and reason about the system's dependability properties without having to employ complex system models and obscure logic vehicles.

The dependability analysis is based on successive refinements of the initial system specification and the associated dependability property, until a blueprint is formed which gives the specifications of the software components composing a fault tolerant mechanism that satisfies the applications' dependability requirements. The employment of the formal framework for the *systematic* dependability analysis is detailed in the next chapter. The compatibility of this framework with other frameworks used in Aster for the description and analysis of the system's quality properties from some other viewpoint (namely, the one concerned with security properties [10]) is a field of ongoing research in Aster (*e.g.* see [34]).

Chapter 4

Systematic Dependability Analysis

The framework presented in the previous chapter serves for the qualitative description of the system's dependability properties. The property refinement defined in this framework provides a correctness criterion for the analysis of some property expressing a dependability constraint to a description of the behaviors of the objects composing the system specification. In this chapter, the discussion is focused on the impact of the property refinement on the architectural structure of the system. A system specification is coupled with the property expressing the dependability requirements placed on it, and for each property refinement the corresponding architectural refinement on the specification is identified. Refinement patterns capture the knowledge of how to transform pairs of dependability properties and system specifications, into an elaborated description of the system architectural structure and the properties of its objects. These refinement patterns are ordered according to the hierarchical structure of the corresponding dependability properties, as discussed in the previous chapter. A classification scheme codifies this order and forms the basis for performing the analysis of system's dependability requirements, and guiding the retrieval of the appropriate fault tolerant mechanisms. The design of a CASE tool is presented, which maintains a database of such refinement patterns, and uses it to aid the dependability analysis and to guide the retrieval of fault tolerant mechanisms. The chapter concludes with an assessment on the contribution of the proposed dependability analysis, and a comparison with existing work.

4.1 Dependable Software Architecture

During the conception of a software system, its dependability requirements that constrain the system behavior in the presence of failures, provide a macroscopic view on the dependability-specific part of the system architecture. This macroscopic view describes vaguely, or not at all, the fault tolerant mechanism that meets best the given requirements. In many cases, the interpretation of a failure itself may not be well-defined yet. The source of this problem has two aspects: *(i)* the property expressing the dependability requirements describes sufficiently the desired system behavior in the presence of failures, but it does not provide any indications on the means for achieving it, and *(ii)* even if the

property describes precisely how to achieve the desired behavior, the interpretation of the property's impact on the architecture of the dependability-specific part of the system, is non trivial. The formal framework presented in the previous chapter gives the basis to the system designer for strengthening the dependability properties and rendering them as precise as needed. In addition, it delineates a relational structure among the dependability properties, which reveals the refinement paths that should be followed to obtain from an abstract dependability requirement a detailed description of how to satisfy it.

In this section, the dependability properties are coupled with their impact on the software architecture of the system to which they are associated. The pairs of dependability properties and the corresponding architectural structures are organized according to the hierarchical structure stemming form the property refinement relation. This hierarchical structure represents a twofold refinement, on system properties and system architecture. The section discusses how to store the refinement patterns expressed by the aforementioned hierarchical structure, into a database of refinement patterns. It also discusses how to maintain and use the database, in order to guide the analysis of the system's dependability requirements.

4.1.1 Specification Refinement

Specification composition can be used to deduce the specification of a system composed from a number of known objects. It can also be used as a criterion for the correct decomposition of a system specification into its constituents: the decomposition of a specification into its constituents is correct *if and only if* the initial specification is subordinate to the composition of the constituent objects. As already raised, this does not provide a method for the systematic decomposition of system specifications, but only a correctness criterion. On the other hand, the analysis of the dependability requirements on a given system aims at identifying the objects that participate to the specification composition which are dependability-specific, *i.e.* the objects having the properties that guarantee the system behavior asked by the dependability requirements. When a specification Σ provides the behavior expressed by a property P, it is said that the specification Σ *models* the property P, and is noted as $\Sigma \models P$. Hence, the dependability analysis aims at transforming a specification and its associated dependability requirement into a set of objects, whose composition produces the same behavior as the initial specification, and at the same time it models the associated property. In addition, the result of the analysis can be a set of objects whose composition produces *at least* the behavior of the initial system and models a property stronger than the one expressing the dependability requirements.

It is most unlikely to perform in one step the transition of an initial specification and its associated dependability requirements to a set of objects whose composition has the aforementioned characteristics. Depending on the abstraction level of the dependability requirements, the dependability analysis may take a number of steps for precisely determining the expected system behavior and the means to achieve it. Each step of the dependability analysis is based on the notion of *specification refinement*. Starting from a pair of a specification and an associated property, the specification refinement yields a set of pairs of objects and their associated properties. The composition of all these objects

produces at least the behavior of the initial specification, and the conjunction of their associated properties and the properties modeled by the objects, implies the initial property. More formally, using the symbol "$\rightarrow$" to denote the specification refinement process, the refinement of a specification Σ according to property P, is defined as follows:

$$\langle \Sigma, P\rangle \rightarrow \langle \bowtie_i \Sigma_i, \bigwedge_i P_i\rangle \quad \equiv \quad \Sigma \sqsubseteq (\bowtie_i \Sigma_i) \ \wedge \ \forall i | (\exists Q_i | \Sigma_i \models Q_i) \ \wedge \ (\bigwedge_i (P_i \wedge Q_i)) \Rightarrow P \quad (4.1)$$

When formula (4.1) holds, it is said that the composition of the set of objects $\{\Sigma_i\}_i$ is a refinement of the system specification Σ, and that the set of pairs $\{\langle \Sigma_i, P_i\rangle\}_i$ is a refinement of the pair $\langle \Sigma, P\rangle$. Also, according to the definition of the property specification given in 3.3.2, the conjunction of the properties $\{P_i\}_i$ and $\{Q_i\}_i$ is a refinement of the property P. Using the above notation, dependability analysis is the process which aims at weakening the properties P_i and strengthening the properties Q_i. The result of the dependability analysis is a set of objects that do not have any particular associated property, and they model a property that is part of the behavior asked by the initial dependability requirement. This is summarized in the following formal expression:

$$\langle \Sigma, P\rangle \rightarrow \ \cdots \ \rightarrow \langle \bowtie_i \Sigma_i, \mathit{TRUE}\rangle, \ \mathit{where} \ (\Sigma \sqsubseteq \bowtie_i \Sigma_i) \wedge (\bowtie_i \Sigma_i \models P) \quad (4.2)$$

Although the goal of dependability analysis is to weaken P_i and strengthen Q_i, the definition of specification refinement does not guarantee it. Actually, intermediate steps may strengthen P_i, in order to render them more precise, before determining the specifications that model them. For example, a specification and the associated property can be refined to a stronger property associated to the same specification, as shown by formula (4.3). Another special case of specification refinement, shown by formula (4.4), is the one that refines a specification into two parts, one being the dependability-specific part and the other that does not have any associated property. This case is useful in the dependability analysis, since it permits to dissociate the algorithmic functionality of the system from its dependability aspects, and to concentrate the dependability analysis only on the latter.

$$(P' \Rightarrow P) \ \Rightarrow \ (\langle \Sigma, P\rangle \rightarrow \langle \Sigma, P'\rangle) \quad (4.3)$$

$$(\Sigma \sqsubseteq \Sigma_1 \bowtie \Sigma_2) \ \Rightarrow \ (\ \langle \Sigma, P\rangle \rightarrow \langle \Sigma_1 \bowtie \Sigma_2, P \wedge \mathit{TRUE}\rangle\ ) \quad (4.4)$$

By successively employing the above two formulas on the dependability specific part of a specification, one can obtain pairs of system specifications with no algorithmic constraints (called *trivial* specifications hereafter) associated to properties expressing well-known fault tolerance techniques. Such pairs are replaced by the specifications of existing fault tolerant mechanisms associated to the property *TRUE* expressing the fact that the trivial requirement is placed on the accompanying specification. The dependability analysis stops when

the initial specification is transformed into a set of objects with no requirements placed on them. Hence, the aim of the dependability analysis can be summarized as: *(i)* dissociating the algorithmic from the dependability aspects of the specifications, *(ii)* rendering precise the dependability properties, and *(iii)* identifying properties modeled by well-known fault tolerant mechanisms.

4.1.2 Architectural Impact of Specification Refinement

Performing the dependability analysis described by formula 4.2, results in a set of objects with no requirements placed on them, whose composition models the initial dependability requirements. However, the structure of the composition does not appear directly in the result of the refinement process. Consider the example of a client-server interaction, where the client has certain dependability requirements on the server behavior. By analyzing the requirements, one may find out that they are satisfied by the composition of the server with a fault tolerant mechanism providing the *Passive* property. However, the result of this analysis does not provide any indication about how to integrate the server specification with the fault tolerant mechanism specification. One should study the implementation of the server and fault tolerant mechanism objects, to deduce the architecture of the composition. This approach of studying the object specifications to deduce the architectural structure of their composition, has a number of inconveniences:

- The interactions of dependability-specific objects, are most captured by the dependability property that they implement. For example, the *Restore* property involves three objects: the object that checkpoints its internal state, the object playing the stable storage role, and the object that restores the checkpointed state. Analyzing the *Restore* property, gives a quick and clear view on the objects interactions. On the other hand, to obtain the same information by analyzing individually the specification of the three objects involved in *Restore*, is a much more difficult task.
- An object may define an interaction with some other object in the surrounding environment but without explicitly identifying that object. During the specification decomposition that yielded the object in question, the interaction link can be easily deduced. But if this information is not stored, it can be lost in subsequent decompositions.
- In the Aster context, the specifications of fault tolerant mechanisms are replaced by their implementation. This makes almost prohibitive the *a posteriori* deduction of the architectural structure, by studying the source/executable code of the implementation.

Besides these inconveniences, studying the object implementations is a time consuming task and requires computer-aid. So, it seems more interesting to provide directly some mechanical support for restoring the architectural structure of the composition. The adapted solution consists of storing in a graph structure the architectural structure of the objects produced by the specification decomposition related to a specification refinement

(*e.g.* see [62]). The constituents of the graph structure are nodes and links. A node represents a distinct object which may have one or more associated links. Links are directed, denoting the data-flow. Links defined between any two nodes of the graph represent interactions of objects participating in the composition. Also, there might be links that have only one edge defined in the graph (*i.e.* they start or they end on some graph node). These links define interactions with the surroundig environment. Figure 4.1 depicts a graph representing the architectural structure of an object composition corresponding to the *Passive* property given in Table 3.2 in page 32, where interactions with the surrounding environment are marked with thicker links.

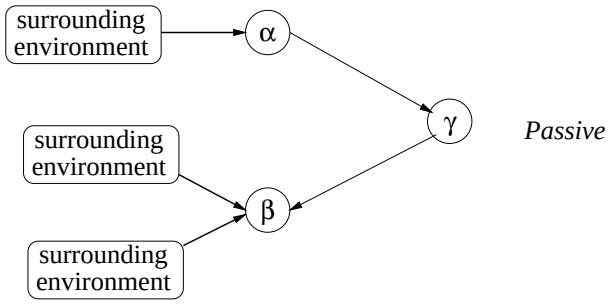


Figure 4.1: The architectural structure of an object composition corresponding to property *Passive*

To be able to produce such a graph, one needs to know the objects participating in the composition and the *potential* interaction paths among them. When this information is available, the construction of the graph is straight-forward: each object is represented by a node and a link corresponds to a potential object interaction. A difference between nodes and links is that the former signifies that an object is *necessarily* present in the composition, whereas a link between two nodes signifies that a message can be *possibly* exchanged between the corresponding objects (or, the absence of a link between two nodes signifies that there is no direct communication between the corresponding objects). This observation has an impact on the specification of the dependability properties. The formula capturing a dependability property, should not introduce ambiguities concerning the existence of an object. Such ambiguities may stem from formulas like $(\exists \Sigma_\alpha | P) \Rightarrow Q$, since the existence of Σ_α is not imperative for the formula to hold. To avoid the ambiguities, the specifications of dependability properties should conform to the following form:

$$\exists \langle objects \rangle | \langle property \rangle \tag{4.5}$$

In the above form, all objects used in the specification expressed by *property* should be declared in advance. An additional constraint in the above form is that the clause $\langle objects \rangle$ should contain exactly those objects appearing in the clause $\langle property \rangle$, and that the latter clause must not contain terms that evaluate to the identity or the impossible formulas

(*i.e.* terms that are always *TRUE* or *FALSE*). Notice that the object ϵ, which denotes an interaction with the surrounding environment, needs not be *"declared"*. Table 4.1 contains the properties given in Table 3.2 in page 29 re-written to conform with the above rule. Properties *Active* and *Semi-Active* remained unchanged, since they were already in the form compelled by the rule given in formula (4.5).

$$
\begin{array}{rcl}
\mathit{DetectionObj}(\Sigma,\Sigma_\alpha) & \equiv & \exists\Sigma,\Sigma_\alpha|(\sigma\in\Sigma\wedge[\sigma]\wedge \mathit{faulty}(\sigma))\Rightarrow((\mathit{xport}(\epsilon,f(\mathit{failure}))\in\Sigma_\alpha)\wedge \\
& & \qquad([\sigma]\prec \mathit{xport}(\alpha,\epsilon,f(\mathit{failure}))))\\[2ex]
\mathit{MaskingObj}(\Sigma_\alpha) & \equiv & \exists\Sigma_\alpha,\Sigma_\beta|((\Sigma_\beta\equiv\Sigma_\alpha)\wedge((\alpha.\sigma\in\Sigma_\alpha\wedge[\alpha.\sigma]\wedge \mathit{faulty}(\alpha.\sigma))\Rightarrow \\
& & m=max\{n\in\mathbb{N}|\mathcal{X}_{\rightarrow n}(\Sigma_\alpha)\equiv\mathcal{X}^C_{\rightarrow n}(\Sigma_\alpha)\}\wedge(\mathcal{X}_{\rightarrow m}(\Sigma_\alpha)\equiv\mathcal{X}_{\rightarrow m}(\Sigma_\beta))\wedge \\
& & \qquad(\exists m'|(m<m'\wedge(\mathcal{X}_{\rightarrow m'}(\Sigma_\beta)\equiv\mathcal{X}^C_{\rightarrow m'}(\Sigma_\beta))))))\\[2ex]
\mathit{Diffuse}(\Sigma_\alpha,G) & \equiv & \exists G=\{\Sigma_\alpha,\Sigma_{\alpha_1},...,\Sigma_{\alpha_N}\},\Sigma_\alpha\mid((\Sigma_\beta\in G\wedge \mathit{xport}(\alpha,\beta,D))\Rightarrow \\
& & \qquad\forall\Sigma_{\beta_i}\in G|\mathit{mport}(\alpha,\beta_i,D))\\[2ex]
\mathit{Active}(\Sigma_\alpha,N) & \equiv & \exists G=\{\Sigma_\alpha,\Sigma_{\alpha_1},...,\Sigma_{\alpha_N}\},\exists\Sigma_\beta\mid((\forall\Sigma_i,\Sigma_j\in G|(\Sigma_i\equiv\Sigma_j))\wedge \\
& & ((\mathit{mport}(\alpha_i,\beta,d_i)\wedge\mathit{mport}(\alpha_j,\beta,d_j)\wedge(\Sigma_{\alpha_i}\in G)\wedge(\Sigma_{\alpha_j}\in G)\wedge \\
& & \qquad(id(d_i)=id(d_j)))\Rightarrow\exists!\mathit{xport}(\beta,\epsilon,d)|id(d)=id(d_i))\wedge \\
& & (((\Sigma_\alpha\in G\wedge\mathit{mport}(\epsilon,\alpha,d))\Rightarrow\forall\Sigma_{\alpha_i}\in G|\mathit{mport}(\epsilon,\alpha_i,d))\wedge \\
& & \qquad((\Sigma_\alpha\in G\wedge(\mathit{mport}(\epsilon,\alpha,d_1)\prec\mathit{mport}(\epsilon,\alpha,d_2)))\Rightarrow \\
& & \qquad\forall\Sigma_{\alpha_i}\in G|(\mathit{mport}(\epsilon,\alpha_i,d_1)\prec\mathit{mport}(\epsilon,\alpha_i,d_2)))))\\[2ex]
\mathit{Semi\text{-}Active}(\Sigma_\alpha,N) & \equiv & \exists G=\{\Sigma_\alpha,\Sigma_{\alpha_1},...,\Sigma_{\alpha_N}\}\mid((\forall\Sigma_i,\Sigma_j\in G|(\Sigma_i\equiv\Sigma_j))\wedge \\
& & (\exists!\Sigma_l\in G|((\Sigma_\alpha\in G)\wedge(\Sigma_\epsilon\notin G)\wedge\mathit{xport}(\alpha,\epsilon,d))\Rightarrow(\Sigma_\alpha=\Sigma_l))\wedge \\
& & (((\Sigma_\alpha\in G\wedge\mathit{mport}(\epsilon,\alpha,d))\Rightarrow\forall\Sigma_{\alpha_i}\in G|\mathit{mport}(\epsilon,\alpha_i,d))\wedge \\
& & \qquad((\Sigma_\alpha\in G\wedge(\mathit{mport}(\epsilon,\alpha,d_1)\prec\mathit{mport}(\epsilon,\alpha,d_2)))\Rightarrow \\
& & \qquad\forall\Sigma_{\alpha_i}\in G|(\mathit{mport}(\epsilon,\alpha_i,d_1)\prec\mathit{mport}(\epsilon,\alpha_i,d_2))))\wedge \\
& & ((\exists\Sigma_x\in G|\mathit{mport}(\epsilon,x,f(l'sfailure)))\Rightarrow \\
& & \exists!\Sigma_{l'}\in G|((\Sigma_\alpha\in G)\wedge(\Sigma_\epsilon\notin G)\wedge\mathit{xport}(\alpha,\epsilon,d))\Rightarrow(\Sigma_\alpha=\Sigma_{l'})))\\[2ex]
\mathit{Passive}(\Sigma_\alpha) & \equiv & \exists\Sigma_\alpha,\Sigma_\beta,\Sigma_\gamma\mid((\Sigma_\alpha\equiv\Sigma_\beta)\wedge \\
& & (\forall\mathcal{X}(\Sigma_\gamma)|\mathcal{X}(\Sigma_\gamma)\equiv\mathcal{X}^C(\Sigma_\gamma))\wedge(\exists f:M(\alpha)\rightarrow Z|d=f(\alpha.\sigma))\wedge \\
& & \qquad\mathit{mport}(\alpha,\gamma,d)\wedge([\alpha.\sigma]\prec\mathit{xport}(\alpha,\gamma,d))\wedge \\
& & (\mathit{mport}(\epsilon,\beta,f(\alpha'sfailure))\Rightarrow(\exists g:M(\alpha)\rightarrow Z|d'=g(\alpha.\sigma))\wedge \\
& & \qquad([\alpha.\sigma]\prec\mathit{mport}(\gamma,\beta,d'))\wedge\forall\Sigma_\epsilon|(([\alpha.\sigma]\prec\mathit{xport}(\epsilon,\alpha,D))\Rightarrow \\
& & \qquad\mathit{mport}(\epsilon,\beta,D))))
\end{array}
$$

Table 4.1: Re-writing the refinements of the property *Reliability*

The rule given in formula (4.5) is a simple case of the familiar constraint of declarative programming languages, where variables must be declared before used. On one hand, this form restricts the expression of the dependability properties specifications. On the other hand, the specifications become more precise. Indeed, the dependability properties specifications aim at elaborating on what are the objects that should be employed to satisfy given dependability constraints. Hence, by asking the specifier to *"declare"* the

objects participating in a property specification at the beginning of it, results in enforcing a more clear view of the object composition related to the property. In addition, this form of dependability property specification has two interesting properties, captured in the following theorems.

Theorem 1 *For a property P whose specifications conform to the rule given by formula 4.5, the derived graph is unique.*

Proof. In the proof of this theorem, the matrix-from of the graph is used. This form consists of a matrix whose rows and columns are indexed with the labels of the graph nodes. A cell (`a,b`) is assigned the value `1` *if and only if* there is a link from node `a` to node `b`, otherwise it is assigned the value `0`. Supposing that one could derive two different graphs from the property P, then there should exist two different matrixes `A` and `B`. Both matrixes have the same number of rows and columns, which equals the number of the objects involved in the property P. Hence, their difference must be in the values assigned to their cells. Let's assume that cell (`a,b`) is a point of difference, say `A(a,b)=0` and `B(a,b)=1`. Then, according to matrix `A`, property P does not contain either the predicate xport(`a, b,` D) or the predicate mport(`a, b,` D). But, according to matrix `B`, P contains at least one of those two predicates. This is impossible and hence the assumption that the two matrixes differ in the value assigned to some cell, is void. So, matrixes `A` and `B` are identical since they have the same number of rows and columns and the same value assigned to each cell. □

Theorem 2 *Two equivalent properties conforming with the rule given in formula (4.5), have the same number of "declared" objects.*

Proof. Let's assume two properties P and Q, conforming with the aforementioned rule, and that P has at least one more declared object than Q. Supposing that the additional object is A, according to the above rule, there must be at least on term $\mathcal{T}(A)$ involving that object, which appears in the $\langle$*property*$\rangle$ clause of P but not of Q. Moreover, $\mathcal{T}(A)$ is not an identity (*i.e.* always *TRUE*) or impossible (*i.e.* always *FALSE*), nor does it participate to some identity or impossible sub-formula of P. Hence, changing the truthness of $\mathcal{T}(A)$ without altering any other term in P, changes the truthness of P. On the other hand, since $\mathcal{T}(A)$ does not participate in Q, the former's truthness does not influence the latter's truthness. So, the truthness table of P and Q differ at least at one point involving the $\mathcal{T}(A)$ term, and therefore the two properties cannot be equivalent. This proves that a different number of *"declared"* objects makes impossible the equivalence of two properties. Hence, two equivalent properties have the same number of *"declared"* objects. □

Although the graphical representation of the architectural structure provides an intuitive understanding of the object interactions in their composition, it has two basic drawbacks: *(i)* it cannot be easily parameterized (*e.g.* to express the interactions of a group of N objects), and *(ii)* it cannot be easily processed by a computer. To cope with these two drawbacks, a textual description of the graph is necessary. Besides the declaration of the graph nodes and links, the description proposed by the author also includes the

```
Architecture  =  Name ParamList { Description } |
                 Name { Description };
Name          =  ID;
ParamList     =  ParamList , Param |
                 Param;
Param         =  ID;
Description   =  NodeDcl GenDcl LinkDcl PropDcl |
                 NodeDcl LinkDcl PropDcl;
NodeDcl       =  Nodes NodeList;
NodeList      =  NodeList , Node |
                 Node;
Node          =  ID[ID] |
                 ID;
GenDcl        =  Generic NodeList;
LinkDcl       =  Links LinkList;
LinkList      =  LinkList , Link |
                 Link;
Link          =  MultiLink |
                 (Node, Node);
MultiLink     =  for ID = ID to ID Link;
PropDcl       =  Property Formula;
Formula       =  string;
```

(a)

```
Filter_arch(N) {
   Nodes b
   Generic a[N]
   Links for i=1 to N (a[i], b),
         (b, ϵ)
   Property Filter(G)
}

Restore_arch {
   Nodes c
   Generic a, b
   Links (c, b), (ϵ, a), (ϵ, b)
   Property Restore(Σa, Σb, Σc)
}

Active_arch(N) {
   Nodes b
   Generic a, a[N]
   Links for i=1 to N (a[i], b),
         for i=1 to N (ϵ, a[i]),
         (a, b), (b, ϵ), (ϵ, a)
   Property Active(Σa, N)
}
```

(b)

Table 4.2: The architectural description of the specification refinement impact

dependability property which led to the specification decomposition represented by the graph. The rationale behind this choice is that a specification decomposition is guided by the property expressing the dependability requirement placed upon the initial specification. Hence, given a property P, the graph expressing its architectural impact consists of a number of nodes equal to the number of objects involved in the property formula, and a number of links equal to the number of *import* and *export* predicates. Notice that matching pairs of *export/import* predicates are represented by a single link, and multiple *import* or *export* engaging the same objects but defined for different data are also represented by a single link.

Table 4.2 (a) gives the general form of the architectural description related to a property, and Table 4.2 (b) illustrates its employment using as examples the *Filter* and *Restore* properties from Table 4.1. This form of architectural description is a self-explained artifact concerning the structure of an object composition imposed by a dependability property. Remember that object ϵ is used to indicate any object of the surrounding environment. Thus, links involving the ϵ object express interactions with the surrounding environment.

A noteworthy point is the optional field **Generic**. This field is used for declaring the architectural components that are mapped to the initial specification, *i.e.* the objects that do not provide a dependability-specific behavior. For example, in the declaration of the *Filter_ arch* in Table 4.2 (b), the behavior compelled by the *Filter* property is provided by the object *b*, whereas the set of objects *a[N]* are mapped to the initial specification whose output is being *"filtered"*. Notice that the declaration of the generic objects related to a given dependability property, is not automatically inferred from the property. Rather, the specifier of the dependability property undertakes the responsibility of declaring which objects engaged in the property, are considered generic.

4.1.3 Integrating Property and Specification Refinement with their Architectural Impact.

Back in Section 3.3.2, the refinement of a property has been defined to give a stronger property, and a DAG structure has been proposed to store the patterns that capture the knowledge of how to refine dependability properties. According to formula (4.1), stronger properties, and hence property refinement, are closely related to specification refinement. Hence, the aforementioned DAG structure can be used to guide the specification refinement process. The specification refinement is also closely related to the decomposition of specifications. However, the hierarchy of dependability properties does not provide any information on how to decompose a system specification by using a refinement of its associated dependability property. Consequently, the DAG structure has to be enriched with complementary information, in order to become an efficient guide for the refinement of system specifications according to their associated dependability properties.

The complementary information that need to be stored in the DAG structure, includes: *(i)* a codification of the property impact on the decomposition of a system specification, and *(ii)* the instructions for the interconnection of the resulting objects, in order to produce a composition equivalent to the initial specification. The description of the property impact on a system architecture introduced in Table 4.2, also describes the interconnection of the objects resulting from the specification decomposition. Hence, by adding this description to the information stored in the DAG nodes, the system designer disposes the means to guide the refinement of a system specification according to some associated dependability requirement. In addition, a set of instructions are available to the designer, for interconnecting the specification decomposition results, in order to obtain a dependable specification equivalent to the initial one.

The enhanced DAG structure can be used to systematically perform the dependability analysis. Given as input the specification of a software system and a dependability requirement placed on it, the developer employs the information found in the DAG structure to identify the architectural impact of the dependability requirement on the system specification. Then, based on this information, the developer decomposes the specification into two classes of objects: those providing a behavior equivalent to the one of the initial specification, and those providing the behavior compelled by the dependability requirement. More concretely, the guided dependability analysis performs a specification refinement process, which separates the initial specification Σ and its associated property P, into the same

specification Σ associated to the trivial property *TRUE*, and a trivial specification Σ_ϕ associated to the property P. The developer's goal is to reveal, after a number of specification refinement steps, the specification Σ_{FT} of the fault tolerant middleware modeling the property P (*i.e.* $\Sigma_{FT} \models P$). The following formal expression summarizes the dependability analysis process.

$$\langle \Sigma, P \rangle \rightarrow \langle \Sigma \bowtie \Sigma_\phi, TRUE \wedge P \rangle \rightarrow \ \cdot \ \cdot \ \cdot \ \rightarrow \langle \Sigma \bowtie \Sigma_{FT}, TRUE \wedge TRUE \rangle$$

4.2 A Database of Refinement Patterns

It has been already mentioned that the dependability analysis is based on the successive refinement of a specification and an associated property expressing the dependability requirements placed on the specification. A record containing information similar to those given in Table 4.2 codifies the knowledge of how to perform distinct refinement steps. In other words, given a pair of a specification and its associated dependability requirement, the record corresponding to the property can be employed to guide the specification decomposition into a set of constituent objects and to provide their interconnections. To use the information captured in these refinement patterns for constructing CASE support for the dependability analysis, the author proposes to store them in a database. The organization, maintenance and utilization of this database, is the subject of study in this section.

4.2.1 Classification of Refinement Patterns

The property refinement given in Table 3.4 in page 32, guides the specification decomposition, which, after a number of refinement steps, results in a set of objects associated to some dependability property. Some of these objects map to the initial specification and are associated to the trivial property *TRUE*, while the rest capture trivial specification and are associated to properties for which there is no stored knowledge of how to further decompose them (*i.e.* the set T of terminals in the grammar $G = \langle S, T, R, Dependability \rangle$ defined in page 32). The data structure storing the knowledge provided by the grammar rules that capture property refinements, consists of the following fields:

- The *property* field contains the english term associated to the property (*e.g. Active, Passive, Filter, etc*).
- The *record* field contains the architectural description corresponding to the property in question (*e.g.* for *Active* the contents of the *record* field are given in Table 4.2).
- The *rules* field contains the grammar rules that capture the known refinements of the property in question. In other words, the contents of this field reveal the conjunctions of known properties that imply the property stored in the current record.
- The *ancestors* field contains pointers to data structures encompassing properties for which the property in question is a refinement (*e.g.* for *Replication* this field points to the records of *Active, Semi-active,* and *Passive*).

- The *successors* field points to the refinements of the current property and their constituents (*e.g.* for *Active* this field points to *Replication, Filter,* and *AtomicDelivery*).
- The *components* field has pointers in the Aster software repository, which indicate the software components that implement the property in question.

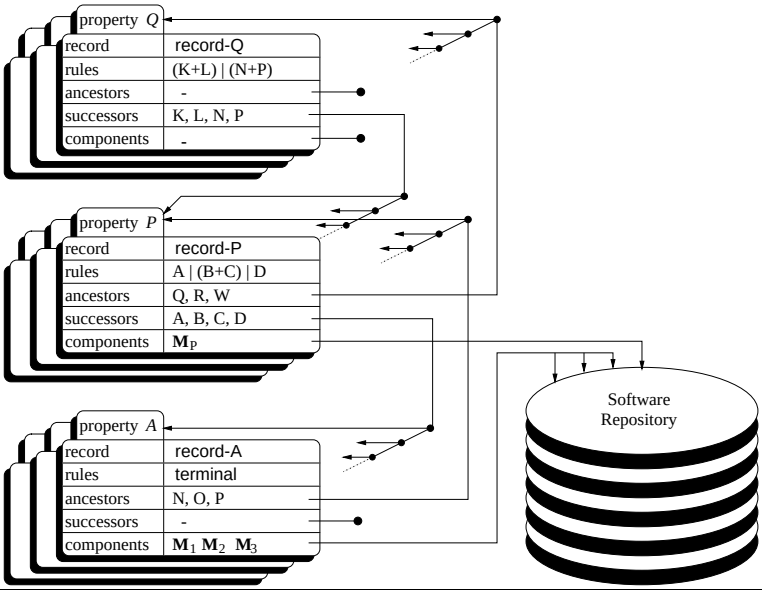


Figure 4.2: Classification of refinement patterns

Figure 4.2 illustrates graphically the organization of the classification scheme of the refinement patterns. Following the notation used in the grammar introduced in the previous chapter, symbol "|" is used to denote different refinement choices as indicated by alternative grammar rules, and symbol "+" is used to denote the indispensable constituents of a refinement choice as indicated by symbols yielded by a grammar rule. Notice that the proposed classification is complementary to the Aster property database, and does not impose any modification, extension or other type of adjustment to it. This was a principal objective during the design of the support for the systematic dependability analysis, which should not alter the standard Aster environment. The rationale behind this objective, is to enable the utilization of the proposed approach in conjunction with similar approaches to the requirement analysis related to other software aspects (*e.g.* related to security, timeliness, *etc*).

4.2.2 Updating the Database

The insertion of a new record in the refinement database is somewhat more tricky than the insertion of a new property in the grammar of dependability properties. This is due

to the fact that, in addition to the relation of the new property with the properties in the database, the relation of its associated architecture with the architecture associated to the existing properties has to be verified. The description of the architectural impact of a property can be generated automatically, and human intervention is necessary only to determine which of the produced objects map to the algorithmic aspects of the systems (*i.e.* the field **Generic** in the architectural description). For P being the new property to be inserted, and Q the property corresponding to a DAG node S, the insertion process is based on two conditions: *(i)* $P \Rightarrow Q$ and *(ii)* $Q \Rightarrow P$.

```
% P, Q: Properties

Place(P, Q)
egin
  Lp := Links (P)
  Lq := Links(Q)
  f ((P⇒Q) nd not (Q⇒P)) hen
    eturn (STRONGER)
  lse if ((Q⇒P) nd not (P⇒Q)) hen
    eturn (WEAKER)
  lse if ((P⇒Q) nd (Q⇒P)) hen
    eturn (EQUIVALENT)
  lse if (ot (P⇒Q) nd not (Q⇒P)) hen
    eturn (INDEPENDENT)
nd
```

```
Insert(P, Q)
egin
  N = Node(Q)
  W = Place(P, Q)
  f (W = STRONGER) hen
  egin
    Anc := N
    oreach Ni n N.successors o
      Insert(P, Ni.property)
  nd
  lse if (W = WEAKER) hen
    - Make P successor of Anc -
    - Fix P's successors -
  lse if (W = EQUIVALENT) hen
    - P already inserted -
  lse if (W = INDEPENDENT) hen
    f (Anc = il) hen
      oreach Ni n N.successors o
        Insert(P, Ni.property)
    lse
      - Put P between Anc and N -
      - Fix Anc's and P's successors -
nd
```

Table 4.3: The outline of the insertion algorithm for the refinement pattern database

Starting from the root of the DAG, which stores the record corresponding to the *Dependability* property for the grammar defined in Table 3.4 in page 32, the above two conditions are checked. The insertion algorithm is captured by functions *Place (P, S)* and *Insert(P, S)* given in Table 4.3. The insertion process starts by assigning to variable *Anc* the value *nil* and then calling the *Insert()* function for S being the node containing the *Dependability* property (*i.e.* the root of the DAG). *Anc* is a global variable, used to keep track of the last node identified as a potential ancestor of the property to be inserted (*i.e.* P). A noteworthy point is that the insertion algorithm is defined for a property P and not for a DAG node storing the information related to P. This makes possible to

insert a conjunction of new properties, which corresponds to a set of more than one DAG nodes. The actions of the insertion algorithm are explained in detail below:

- If *only* the first condition holds, then property P is a refinement of property Q. Node S takes the place of a candidate direct ancestor of the new node, noted as *Anc*. The process is repeated for each of S's successors.
- If *only* the second condition holds, then node S contains a refinement of property P. Property P is inserted as a direct successor of the node *Anc*, and the DAG links among P, *Anc*, and S are fixed. If property P is not given as a conjunction of other properties (*e.g. DetectionObj*), then the node corresponding to P is inserted between *Anc* and S, *i.e.* S is replaced by P in *Anc successors* field, and it is added in P *successors* field, and the *ancestors* fields are fixed accordingly. If property P is a conjunction of other properties (*e.g. Active*), then it corresponds to more than one nodes. All these nodes become successors of *Anc*, but the insertion does not alter the DAG link between *Anc* and S. This latter case is graphically depicted in Figure 4.3. In addition, when P is a conjunction of other properties, for each of the conjunction terms the *Place (P, S)* function is called to determine whether the corresponding property has already been inserted. In that case, only the DAG links with the existing node are established. For example, if the *Passive* property is inserted after the *Active*, then no DAG node should be added for the *Replication* sub-property, which is already present in the DAG since the insertion of *Active*.

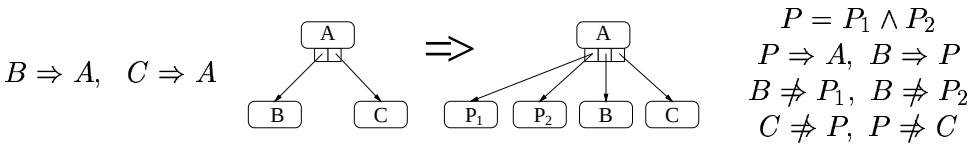


Figure 4.3: Property P inserted as a successor of A

- If *both* conditions are satisfied, the new property is found equivalent with some existing one. Consequently, the number of objects involved in the formulas of the two properties is equal, as it has been already proved. In this case, the architectural structure corresponding to the new property is identical to the one of the equivalent property already in the database. Hence, no further insertion actions are necessary, since an identical node exists already in the classification scheme[1].
- If *none* of the two conditions is satisfied, then there are two sub-cases:

[1] Actually, there might be a difference in the **Generic** fields of the records corresponding to the two nodes. In this case, the CASE tool discussed later in this chapter, alerts the database administrator for a possible conceptual conflict.

1. There is no DAG node marked as candidate ancestor of the new node (*i.e.* $Anc = nil$). This means that from the beginning of the insertion process, the new property has not been found to refine any of the properties corresponding to the nodes visited. In this case, the rest of the DAG nodes are visited in a depth-first traversal order. The rationale behind this is that when the new property is the refinement of some constituent property (*e.g.* consider the insertion of the *DetectionObj* property), it may traverse some part of the DAG without satisfying any of the two conditions above, before reaching its insertion point. If the whole DAG is traversed without ever satisfying either of the two properties, then the new property is not inserted at all. For example, this case may raise if the *Filter* property is attempted to be inserted independently of the *Replication* and *AtomicDelivery* properties.
2. In previous steps of the insertion process, the new property has been found to satisfy the first condition (*i.e.* $Anc \neq nil$). Hence P has been recognized as the refinement of the ancestor of S, but it is not related in any manner with S. Then, the new node is inserted as a successor of Anc. In addition, Anc is interrogated to find out whether Q must be combined with the property of some other successor of Anc to form a refinement of the property contained by Anc. Each such combination found, is checked for implying the property P. The combinations found to imply the property P become successors of node P. Figure 4.4 illustrates graphically that case.

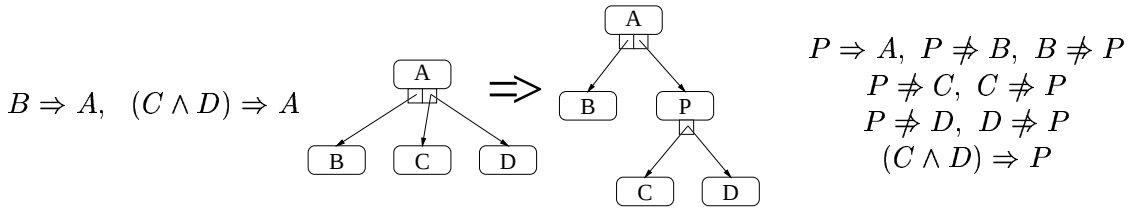


Figure 4.4: Property P inserted between A and the combination of C and D

Notice that it is possible that the insertion of a new property may cause a number of rearrangements in the DAG structure. This is the case, when the conjunction of a newly inserted property with other existing properties forms a refinement of some property stored in the DAG. This is called a *"hidden"* refinement relation. The insertion algorithm does not cope with such rearrangements. The author has not a response for accomplishing these rearrangements easily. The solution currently adopted, is the exhaustive verification of all the combinations of the properties stored in the DAG, to discover *"hidden"* refinement relations.

4.2.3 Guided Software Retrieval

The practical benefit of the dependability analysis is that it can be used to guide the retrieval of the software components that, when composed, provide the implementation of a fault tolerant mechanism which satisfies the initial requirement. To achieve this, when a refinement step yields a pair of a trivial specification associated to a property that is known to be modeled by some existing software component, the latter can be selected. The selected software component can be integrated with other software components selected in a similar manner, to compose the fault tolerant mechanism that meet the requirements on which the dependability analysis was applied.

As mentioned in the Chapter 2, software retrieval in the Aster context is based on the properties implemented by the components stored in the middleware repository. The Aster logical tool attempts an exhaustive database search, until it finds a software component (or a combination of software components) implementing a property that implies logically the search-key. Then, the corresponding components from the software repository are retrieved. The logical tool consists of two main parts. The first part does the interfacing with the property database and performs the exhaustive database search. The second part serves as a light theorem prover, and decides whether the property selected from the database implies logically the search-key.

The systematic dependability analysis can replace the first part of the logical tool, in the retrieval of the fault tolerant mechanism that fits the application's dependability requirements. The theorem prover is still necessary for the implementation of the *Place()* algorithm, but the exhaustive search in the property database can be replaced by the guided search of the DAG structure. A CASE tool based on the DAG structure *"sieves"* the population of the property database, and considers only the records of the properties that participate to some refinement of the dependability requirement of the system. Each refinement of the dependability requirement, according to the DAG structure, defines a cluster of property records. The records of one such cluster are used to guide the software retrieval, only if there exists at least one implementation for every record in the given cluster. Table 4.4 gives a more precise description of the *"sieve"* algorithm.

The *"sieve"* algorithm shown in Table 4.4 consists of three functions. Function *Retrieve(P)* is used to initialize the sieving process, by calling the *Sieve(P, S)* function for S being the DAG node corresponding to the *Dependability* property, and to use the returned set of P's implementations for guiding the software retrieval of fault tolerant mechanisms. The *Sieve(P, S)* function takes as arguments a property P and a DAG node S. Using S as a starting point in the classification scheme and P as a search-key, *Sieve(P, S)* employs the function *Place(P, S)* (see Table 4.3) to locate the first successor of S that contains a property which forms a plug-in match to P (*i.e.* a property which is at least as strong as P). For each such successors found, the *GetImpl(N)* function is employed to retrieve the DAG nodes involved in its implementations. The return value of *Sieve(P, S)* is a set sets of DAG nodes, each corresponding to an implementation of P. Finally, the *GetImpl(N)* function returns a set of DAG nodes corresponding to possible implementations of the property stored in node N. Node N itself can be included in the returned set, if it indicates some components in the software repository. In any case, each rule R of the *N.rules* field is

```
GetImpl(N)
egin
  G := ∅
  f not (N.components = il) hen
    AddImpl(G, N)
  oreach R n N.rules o
  egin
    OK := RUE
    X := ∅
    oreach S n R o
    egin
      Z := GetImpl(S)
      f (Z = ∅) hen
        OK := ALSE
      AddImpl(X, Z)
    nd
    f (OK = RUE) hen
      AddImpl(G, X)
  nd
  eturn(G)
nd
```

```
Sieve(P, S)
egin
  G := ∅
  N := Node(S)
  W = Place(P, S)
  f ( (W = EQUIVALENT) r
      (W = WEAKER) ) hen
    G := GetImpl(N)
  lse
    oreach Ni n N.successors o
      G := G ∪ Sieve(P, Ni.property)
  eturn(G)
nd


Retrieve(P)
egin
  G := Sieve(P, Dependability)
  - Populate temporary database with G -
  - Employ the Aster logical tool -
nd
```

Table 4.4: The *"sieve"* algorithm

followed and for each constituent S in rule R, *GetImpl(N)* is recursively called to return the set of DAG nodes indicating its possible implementations. If there does not exist an implementation for a symbol of a given rule, then the given rule does not contribute at all to the returned set (since the implementation corresponding to this rule is not complete).

In cases where the retrieved implementations of P are more than one, the developer's intervention is required, to decide which of the suggested solutions to use. This is acceptable because the human interaction serves to clarify the utility of dependability constraints and to make explicit design choices on the fault tolerant mechanism that should be employed to satisfy dependability constraints. In addition, the intellectual effort for this intervention is minimal for an experienced designer, since alternative solutions are available in hand and the choice does not require any computations. The cost of the human intervention is out-balanced by the practical benefits that are demonstrated in the next chapter, where dependability requirements are translated into the retrieval of fault tolerant mechanisms from the Aster software repository.

The configuration and interactions of the tools that participated in the systematic dependability analysis which guides the software retrieval of fault tolerant mechanisms in Aster, are detailed in Figure 4.5. The property database contents are filtered by the *"sieve"* tool, based on the set of refinement patterns, stored in the DAG structure. The

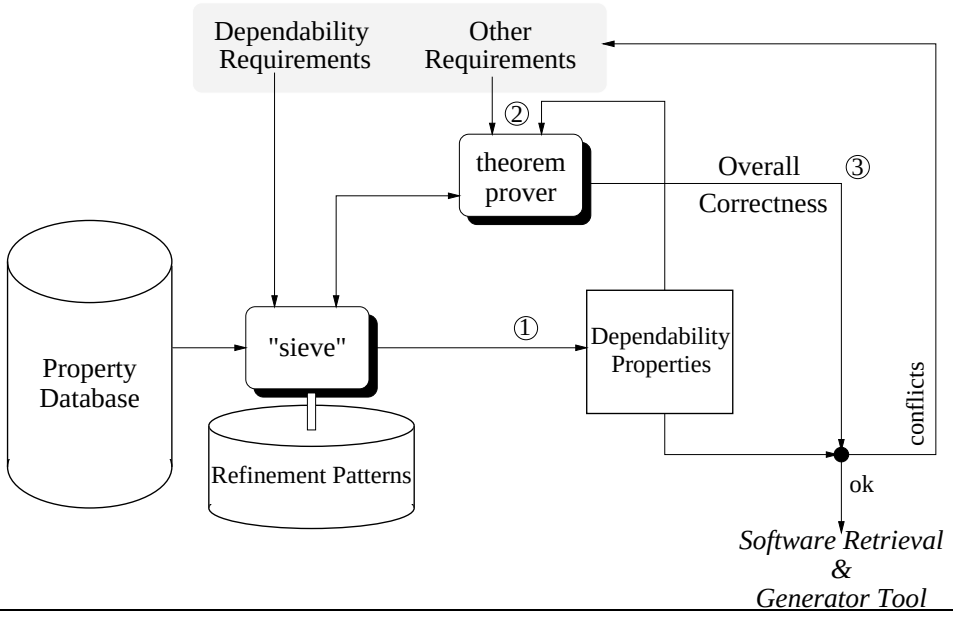


Figure 4.5: Integration of our CASE tool within the Aster software retrieval process.

theorem prover part of the Aster logical tool, is used by the *"sieve"* tool, to implement *Place()*, which is called in the *Sieve()* algorithm. The output of this first step includes the records of the properties that form a refinement of the dependability requirement of the system, and that also indicate a complete implementation of the fault tolerant mechanism that satisfies it. In the next step (step 2 in Figure 4.5), the theorem prover is used again, to verify the correctness of the dependability analysis results, with respect to the requirements concerning other software aspects (*e.g.* constraints related to the security quality). If no conflicts are detected (step 3 in Figure 4.5), then the selected components are passed to the Aster tools that perform the software retrieval and the incorporation of the retrieved components in the structure of the initial system.

4.2.4 Deriving the Dependable System Architecture

Besides guiding the software retrieval, another practical benefit of the presented dependability analysis, is that it also provides the guidelines for integrating the retrieved fault tolerant mechanism, with the system architecture. Using the hierarchy of the dependability properties captured by the classification scheme, the *Sieve()* algorithm finds a property X, which satisfies the requirements passed as parameter to the algorithm (argument P). Then, it employs the *GetImpl()* algorithm to retrieve all the DAG nodes that participate to some refinement of X, and that are also implemented by some software components in the Aster repository. The guidelines for incorporating the retrieved components in the

system architecture, are given by the DAG path that leads from the property X to the nodes storing the properties implemented by the components.

Let's take the example of a dependability requirement Y, whose analysis starts with a call to *Sieve(Y, Dependability)*. Let's also assume that the dependability analysis is based on the grammar given in Table 3.4 in page 32, and that the property found to satisfy the requirement is *Masking*. In this case, a call to *GetImpl(Node(Masking))* is made inside the *Sieve()* algorithm, to retrieve the DAG nodes that indicate which software components participate to the composition of the fault tolerant mechanism that satisfies the dependability requirement. Then, a number of recursive calls to *GetImpl()* aim at visiting all the refinement of the *Masking* property, and selecting the DAG nodes that indicate an implementation of such a refinement. As a result of these recursive calls, *GetImpl()* visits the subtree rooted at node *Masking* in Figure 3.2 in page 33. Let's suppose that the *DetectionObj* and *Active* nodes do not indicate some implementation, but the rest of the aforementioned subtree nodes do. In this case, *GetImpl()* cannot retrieve a complete implementation of the *Masking* refinement expressed as the conjunction of *MaskingObj* and *DetectionObj*. On the contrary, a complete implementation for the refinement expressed as the conjunction of *Active* and *Diffuse* can be retrieved, which concerns the fault tolerant mechanism composed by *Diffuse, Filter, AtomicDelivery,* and *Replication*.

To reveal the guidelines for incorporating the fault tolerant mechanism in the system architecture, the DAG paths leading from *Masking* to each of the properties for which an implementation has been selected, are used. Each DAG node contains a codification of the architectural impact caused on the system specification by the dependability property, represented by the node. In the above example, developer obtains three levels of blueprints describing the architectural impact of the mechanism satisfying the dependability requirement Y. The first level, describes how the *Masking* property influences the system architecture. The second level, describes how each of the *Diffuse* and *Active* properties influence the system architecture. Finally, the last level of blueprints gives the description of the architectural impact caused by each of the properties *Filter, AtomicDelivery,* and *Replication*. Given these blueprints, the developer has in hand the following instructions for interconnecting the software components, and obtaining the overall dependable architecture:

- How to interconnect the software components implementing *Filter, AtomicDelivery,* and *Replication*, to obtain a composition that provides *Active*.
- How to interconnect the above composition with the software component providing *Diffuse*, to obtain a composition that provides *Masking*.
- How to interconnect the above composition providing *Masking* with the system components, which are required to provide a property that is satisfied by *Masking* (property Y in the above example).

4.3 Evaluation

The grammar rules expressing property refinements and the derived classification scheme, gave the basis for rendering the dependability analysis *systematic*. By re-writing in a specific form the formulas of the dependability properties, it became possible to infer the architectural description corresponding to a property specification. This yielded a set of patterns that capture the property refinements and their associated architectural impact, which are organized according to the classification scheme expressing the refinements of dependability properties. The practical benefits of this classification scheme in the dependability analysis, are shown by the design of two CASE tools: *(i)* one that inserts automatically a new property and its architectural impact in the classification scheme, and *(ii)* one that uses the classification scheme to perform the dependability analysis, whose results guide the retrieval of the software components composing the fault tolerant mechanism, that satisfies a given dependability requirement.

4.3.1 Assessments

The classification scheme for the database of refinement patterns has a number of interesting characteristics. First, it allows the automatic insertion of a new property in the DAG structure expressing the classification scheme. The algorithm described by functions *Place (P, S)* and *Insert(P, S)* in Table 4.3, shows how to insert a node corresponding to a new property P, and how to establish the links expressing the refinement relations of it with the existing node in the DAG structure. The insertion process is completely automated. Even for fixing the links of the new node with the software repository (*i.e.* the *components* field), human interaction is not required. A light theorem prover is used to identify the dependability properties that form an *exact match* (see [75] and the discussion on the Aster logical tool in Chapter 2) to property P. These pointers are added to the *components* field of the new node.

It should be mentioned here that the light theorem prover, used for the insertion of new dependability properties into the classification scheme and for the dependability analysis process, is not the one of the logical tool in the Aster prototype. The theorem prover of the Aster prototype works for predicate logic, but the system model and the accompanying framework for the formalization of the dependability properties are both based on temporal logic. To test the proposed approach for the systematic dependability analysis, the author worked on the evolution of the standard logical tool to support temporal logic formulas. This involvement with a version of the logical tool for temporal logic led to an important observation: a fully fledged theorem prover is mandatory for coping with a wide scope of dependability properties. Simple pattern-matching technology and elementary re-writing techniques do not suffice when dealing with formulas as complex as those given in Table 4.1 in page 42. Although the validity of the proposed approach to dependability analysis is not influenced by the efficiency of the theorem prover used, its practical success is heavily depended on it. This statement is not true only for the analysis of dependability properties but for all quality system properties. As a result, the Aster team has started working on

the integration of the STeP[2] theorem prover with the Aster development environment, in order to solidify the practical contribution of Aster in the analysis of system quality requirements.

The reader should notice that the utilization of the classification scheme may produce more than one results (*i.e.* sets of properties whose conjunctions satisfy the initial dependability requirements). In such cases, the developer is asked to choose which one should be applied to the system. The selection among different alternatives can be restrained by requirements concerning other qualities of the system (*i.e.* efficiency, security, timeliness, *etc*). However, there are no guarantees that the combination of constraints concerning different system qualities will result in a *single* choice for the developer. The experience of the Aster group on the combinations of the requirements concerning security and dependability, confirms this assertion.

Another characteristic of the classification scheme, is that it guarantees the *soundness* of the dependability analysis results, *i.e.* the fact that every aspect of the initial dependability requirement is correctly covered. Due to DAG construction, the conjunction of the properties resulting from the dependability analysis is guaranteed to imply the property expressing the dependability requirement in its totality. On the contrary, the classification scheme cannot guarantee the *completeness* of the results, *i.e.* the fact that the dependability results provide every possible means for satisfying the initial requirement. It can only provide all registered means for satisfying the initial requirement. The completeness of the results is heavily based on the completeness of the refinement patterns database, which in turn is based on the knowledge on dealing with the type of failures expressed in the initial requirement. Hence, to assure the completeness of the dependability analysis results, one must first prove that a given set of fault tolerance techniques contains every possible means to confront a given type of failures. Then, it suffices to insert the corresponding properties in the classification scheme, in order to claim the completeness of dependability analysis results. As long as such a proof does not exist, on may keep enriching the refinement pattern database with new dependability properties, and obtaining this way more alternative choices (although not a complete set of them) for satisfying the dependability requirements of software systems.

4.3.2 Related Work

The systematic dependability analysis is shown to play two roles in the development of software systems. First, it methodically transforms the dependability requirements into a set of software components composing the fault tolerant mechanism that satisfies them, and reveals the interconnections that permit the correct composition of the identified components and their incorporation in the initial system. Second, it guides the retrieval of the identified components from a software repository. With respect to the first role, the systematic dependability analysis shares a number of common points with the work done in the area of the software architecture. The second role is more relevant to the work done in the domain of software classification and retrieval.

[2] http://theory.stanford.edu/people/zm/step.html

Software Architecture

The field of software architecture aims at describing the structure and the properties of the software components composing a system. The structure is expressed at the design level in terms of *components* and *connectors*, which are dissociated from their implementations. However, the perspective revealed by software architecture does not capture all the details that would permit the straight-forward instantiation of a given architecture in a given execution environment. Hence, when it comes to materialize an architectural description, a number of conflicts may raise, as reported by the AESOP experience [21]. This suggests that guidance should be provided, to facilitate the transition from an abstract architecture to its executable instantiation.

Despite the generality of the theoretical objective, existing approaches to software architecture focus mainly on the structural and interconnection aspects of a software system, ignoring most of the time a number of aspects like dependability, efficiency, security, timeliness, *etc.* Indeed, the software properties that have attracted designers' attention are those related to architectural styles [68], *i.e.* those related to the instantiation patterns and interaction models that describe the deployment and the coordination of the components constituting the software system. Among the prevalent work in the area, Darwin [44] focuses on properties related to reconfigurable system structures and provides means to describe the dynamic instantiation of architectural elements according to execution conditions. With similar objectives, Le Metayer [46] has modeled architectures as graphs and architectural styles as grammars on these graphs, in order to express the evolution of the states of the entities participating in the architectural description of a software system.

Of equal importance, *architecture description languages* (ADLs) like AESOP [20] and Wright [2] support the explicit encoding of a wide range of interaction models. These descriptions can be used to feed systems like Unicon [67] in order to produce implementations of the input architectural description. Complementary to the above work are efforts like SAAM [35], which aim at studying the functionality allocation to architectural elements. Analyzing the functionality allocation and the architectural styles used in a software system gives an insight on *what* the system does and *how* it does it. In the context of event-based architectures, Rapide [43] combines these capabilities by providing a suite of tools for the specification of: *(i)* the architectural components, *(ii)* the rules for connecting them, and *(iii)* their algebraic and interconnection constraints.

The contribution of the presented dependability analysis is orthogonal and, at the same time, complementary to the above approaches. It allows the designer to associate dependability attributes to existing architectural elements, without stating explicitly the mechanism implementing them, and to refine these attributes in later design stages in order to reveal the corresponding mechanisms. This preserves the rationale of design choices and the clarity of the architectural description, without forcing choices concerning the implementation of the quality system properties to be taken at early design stages. On the other hand, it does not provide means to describe the interaction patterns and their associated constraints, which is currently the focus of interest in the software architecture domain. For this task, ADLs like Wright or AESOP and their associated environments, can be employed.

Refining abstract attributes into concrete descriptions of an assembly of software components, is not a new idea in the domain of software architecture. Moriconi *et al.* [51] have proposed a formal framework that permits the correctness verification of refinements that produce *instances* from *reference* architectures. Also, Rapide [43] supports the correctness verification for refinements of event-patterns. However, the *reference* and the *instance* architectures have to be known in advance to verify the refinement correctness, which implies an extended study and profound cognition of the refined domain, as stressed by the experience report from the ADAGE project [6].

Fortunately, fault tolerance has been studied for years and the domain has reached a maturity level, which permits the deduction of refinement patterns for a set of well-understood dependability abstractions. As a demonstrative example, the RCP architecture [74] provides four refinement levels for transforming the *uniprocessor* model of an application into a *distributed asynchronous* model, which is the fault tolerant version of the application based on active replication. The systematic dependability analysis is based on the formal specification of the known fault tolerance refinement patterns. Potentially, it can transform any non-dependable software system into its fault tolerant version that conforms with a set of dependability constraints. Hence, it is more general than RCP, since it does not produce fault tolerant system versions based on the active replication solely.

Finally, it should be mentioned that the use of refinement patterns has been proposed by Moriconi *et al.* [51] for the entire set of constraints related to the diverse interwoven software aspects. But, a refinement pattern for the entire software architecture is difficult to reuse since a small variation in the initial requirements may compel a different refinement to result into the specifications of a concrete software system. Such general purpose refinement patterns are inflexible, fail to keep pace with the evolutions in the requirement analysis of each separate software aspects, and compromise the usability and the application domain of the CASE tools built on them, as argued by Henninger [24].

Software Retrieval

Formal specifications have been used for indexing purposes in software repositories, mainly due to the verifiable correctness of the retrieval results with respect to the search key. A representative example is the work of Mili *et al.* [47], who propose the organization of a software repository in a lattice structure that preserves the partial order of component properties specified by the *more defined* relation. The basic difference with the presented organization of the refinement pattern database, is that the latter contains also information about the interconnections of the selected components among them and with the system with which they should be integrated.

Other approaches that are not based on formal specifications, result in a repository structure easier to construct. However, they leave open the correctness issues. For example, the use of a vector of values that specify different aspects (*facets*) of the component that needs to be retrieved [56], does not require an excessive effort for building the corresponding repository structure. On the other hand, it does not provide any correctness guarantees for the integration of the retrieved component in a given software context. Sim-

ilar remarks apply for free-text indexing, although the latter can be proven to better adapt in evolutionary changes of the knowledge on efficient software retrieval (*e.g.* see [24]).

A software retrieval method that shows a remarkable similarity with the approach presented in this chapter, is the one called NORA/HAMMR [66]. This method employs a pipeline of rejection filters, which perform signature matching and model checking. The output of these filter is used by a theorem prover, which is the heart of the software retrieval. The NORA/HAMMR objectives are very similar to the retrieval of fault tolerant mechanisms in the Aster context, but the use of filters is completely different in the two cases. In the NORA/HAMMR case, filters are based on data type and model compatibility. In the Aster case, the CASE tool that performs the dependability analysis is based on specifications compatibility. This makes the two approaches complementary rather than competitive.

4.3.3 Epilogue

The composition of the constituents of a fault tolerant mechanism and their incorporation in a given system, were discussed at the architectural level. A number of important issues remain open, concerning the integration of the application software components with those implementing the fault tolerant mechanism. These issues are complementary to the systematic dependability analysis and the guided retrieval of a fault tolerant mechanism. They are closely related to software composability and interoperability, which are research axes in the area of software reuse [37]. Although the proposed software retrieval guided by the dependability analysis is independent of these aspects, it is required that the software repository is populated with reusable components (*e.g.* see [19, 22]) in order for the software retrieval to be practically beneficial. This issue is extensively studied in the domain of software reuse, and there are results proving that it is possible to build libraries of reusable components that can be automatically composed [7].

Among the proposals concerning the construction of software repositories containing reusable constituents of fault tolerant mechanism, is the *micro-protocol* suite [26, 25]. This proposal suggests populating a software repository with components implementing functionalities corresponding to basic abstractions from the field of fault tolerance (*e.g.* atomic execution, reliable communication, membership management, *etc*). In the terminology used in this document, the basic fault tolerance abstractions are the dependability properties found closer to the leaf-level of the DAG capturing the refinement relation of the dependability properties. Such repositories can be used in conjunction to the presented systematic dependability analysis, to provide the system developer with the constituents of the desired fault tolerant mechanism. Then, the selected fault tolerant protocols can be configured and instantiated employing means that are proposed for the incorporation of fault tolerant mechanism in software systems at the implementation level (*e.g.* see [72]).

Chapter 5

A Case Study

The focus of this chapter, is the utilization of the refinement database and the accompanying CASE tool for systematic dependability analysis. The utilization is exemplified through the analysis of the dependability requirements on a simulation of a distributed file system (DFS) implemented over a CORBA platform. Based on the refinement database containing the properties discussed in previous chapters, the dependability requirements are transformed into a set of software components composing a fault tolerant mechanism that satisfies the requirements. In addition, a set of instructions is retrieved, describing how to structure the selected software components to produce the mechanism and how to integrate them with the file system. The obtained dependable software architecture is used by the Aster tools to integrate the selected software components with the components of the file system, in order to render the latter dependable.

5.1 A Distributed File System

The DFS studied in this chapter, consists of four types of components: files, servers, clients, and the *locator*. Each server object offers file-access services for the files belonging in its domain, and the domains of any two servers do not intersect. A client accesses files in a Unix-like fashion, *i.e.* it uses the file name to create a session identified by a file descriptor, and it performs read and write operations over such a session. Hence, operations on files are divided into two categories, those performed on a file name (*e.g.* create(), open(), delete(), *etc*) and those performed on a file descriptor (*e.g.* read(), write(), close(), *etc*). The locator object is responsible to prepare sessions, since the file name alone does not contain any information about the file server possessing the file. The locator interrogates file servers to find out the one providing access to a given file, but it does not create a session with it. The identifier of the right file server is returned to the client, which creates the session. Figure 5.1 gives the graphical description of the DFS architecture. A detailed description of the DFS can be found in [30].

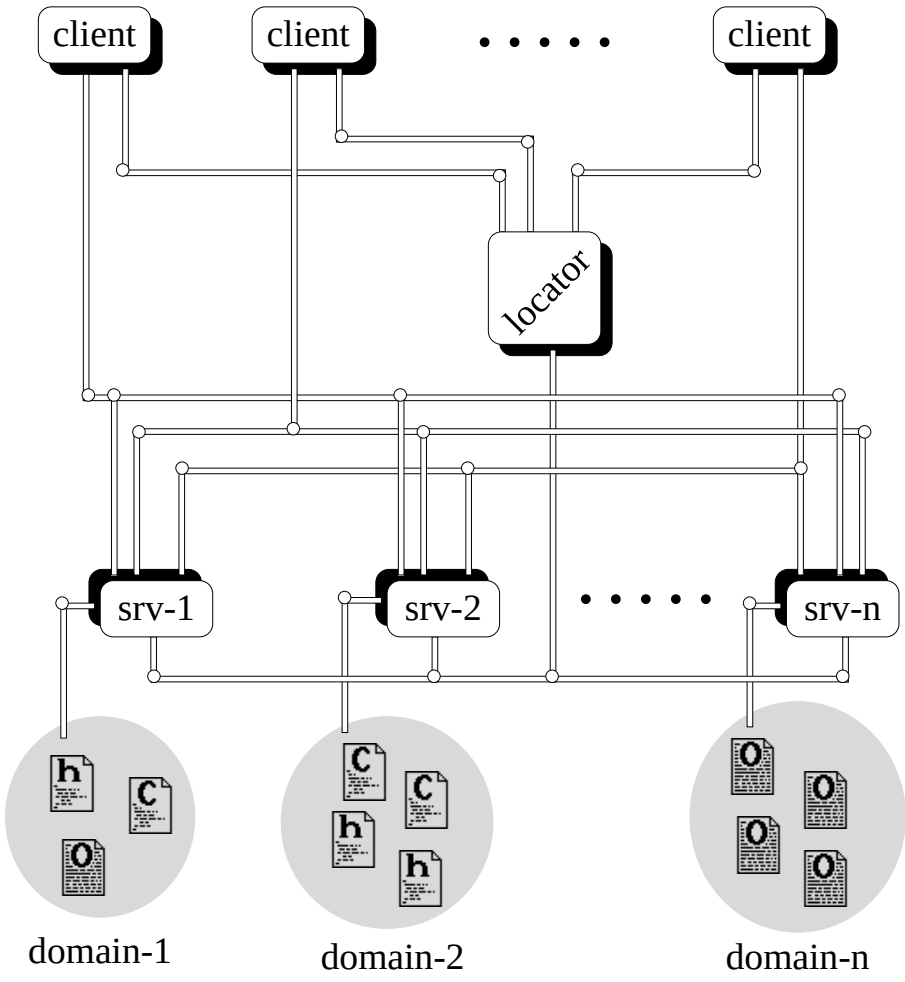


Figure 5.1: The DFS architecture

5.1.1 The CORBA Implementation

A simulation of the DFS described above is implemented as a CORBA application. The execution platform is the version 2.3 of Orbix[1], IONA's implementation of the the CORBA standard, executing over Sun's SOLARIS[2], SunOS version 5.5.1. The server, client and the locator entities are implemented as CORBA objects. On the other hand, files are passive objects implemented as native Unix files. Each server accesses files in a private directory, where other servers do not have access. The locator is a persistent CORBA object exporting an interface known to server and client objects. Upon instantiation, a server registers with the locator by sending a string containing its CORBA object reference. To perform a

[1] http://www.iona.com/products/orbix/index.html
[2] http://www.sun.com/solaris/

number of operations on a file, a client sends to the locator the interface containing these operations, and a string containing the property *exists(fname)*. The locator returns the list of registered servers that export a *compatible* interface (*i.e.* an interface being a superset of the interface required by the client), and that satisfy the aforementioned property too. Conceptually, the reader may think of the locator as an implementation of the CORBA *Trading* Common Object Service (see chapter 16 in [17]).

Based on the interactions of the file system entities, the following interaction patterns can be observed:

- Server - locator: when a file server registers with the locator. The interaction is initiated by the server.
- Client - locator - server: when a client searches the right server to contact for obtaining access for a specific set of operations on a specific file. The client initiates an interaction with the locator. The latter initiates one interaction per registered server with interface compatible to the one requested by the client, in order to find out whether the server matches the client's requirements.
- Client - server: when a client establishes a session with a server (*e.g.* when a client performs an *open()* operation). The interaction is initiated by the client.
- Client - server - file: when a client performs an operation on a file over an open session (*e.g.* a *write()* operation). The client initiates an interaction with the server, which in turn invokes the requested operation on the file corresponding to the current session.

The first interaction pattern occurs once per registered server, while the second and the third occur once per session. The last pattern is the one occurring most often (*e.g.* frequently, a file is opened to perform more than one *read()* or *write()* operations on it). The focus of this chapter is concentrated on rendering fault tolerant the file system interactions following this last pattern. Notice that interaction patterns are independent from the methods exported by the file server. Hence, the DFS can be populated with various types of file servers, and still follow the above interaction patterns. Table 5.1 gives the CORBA-IDL form of some of the file server interfaces that can be found in the DFS.

5.1.2 The Dependability Requirements

Given the fact that a number of failure events may disturb the correct DFS functioning, the developer may wish to shield the system against some of them (*e.g.* those occurring most often, or the most severe ones). The dependability requirements on the DFS contain the assumptions about the failure types that may occur in the system, and the constraints that the system should meet despite the occurrence of failures mentioned in the assumptions. In a distributed system, failure events may concern the participating entities (*e.g.* clients, servers, files and the locator in the DFS case) or the communication among them (*e.g.* method invocation and response in the DFS case). Among the well-known failure types for processes are the *"fail-stop processors"*, the *"crash processors"*, and

```
nterface fsOC
  ong open(n string fname, n long flag,
           n long mode);
  ong close(n long fildes);
;

nterface fsAPD : fsOC
  ong creat(n string fname, n long mode);
  ong append(n long fildes, n string buf,
             ong in size);
;

nterface fsACC
  ong chmod(n string fname, n long mode);
  ong chown(n string fname, n long owner,
            n long group);
;

nterface fsPR
  tring lpq(n string pname, n long flag);
  ong lpr(n string fname, n string pname,
          n long flag);
  ong lprm(n string pname, n long flag,
           n long job);
;
```

```
nterface fsRD : fsOC
  ong read(n long fildes, ut string buf,
           n long size);
  ong lseek(n long fildes, n long offset,
            n long mode);
;

nterface fsWR : fsAPD
  ong unlink(n string fname);
  ong write(n long fildes, n string buf,
            n long size);
;

nterface fsRDWR : fsRD, fsWR

nterface fsFULL : fsRDWR, fsACC
```

```
nterface fsTEX
  ong tex(n string fname);
  ong latex(n string fname);
  ong latex2e(n string fname);
  ong dvips(n string fname, n string opt);
;
```

Table 5.1: File server interfaces in the DFS

the *"byzantine processors"*. The semantics of these types of failures can be easily extended to cover the failure events that concern the files in the DFS. On the other hand, some well-known communication failures are the *"omission failure"*, the *"performance failure"*, and the *"byzantine failure"*.

A failure event may have a different impact on the correct DFS functioning, depending on the entity where it occurred. For example, assuming crash processors and omission failures, a failed client causes the temporary inaccessibility of the files it has opened, while a failed server causes the permanent inaccessibility of the files lying in its domain. Similarly, a lost message sent from a client to the locator prevents the given client from accessing a single file, while the failure of the locator prevents every client from accessing files other than those it has already opened[3]. The developer has to capture in the dependability requirements, which failure events should be confronted and what behavior the system should experience in each case. The dependability requirements are specified for some

[3]This happens due to the fact that clients should obligatory pass from the locator to find the server that provides access to a given file.

interaction pattern (*e.g.* the server - locator interaction), and express the system behavior that the entity initiating an interaction expects from that interaction (*e.g.* independently from locator failure or message loss, a server registration should be atomic, *i.e.* either the server should be successfully registered or no trace of the server should be found in the locator database). Given the fact that the client - server - file interaction is the most often used in the DFS, the rest of the chapter focuses on the specification of the dependability requirements for this pattern.

5.1.3 Fault Tolerant File Access

A client can establish a session with a server to access a file, perform a number of operations to that file, and close the session after completing the operations on the file. Failures occurring on the communication, the server, or the file during an open session, may destroy the session, cause the client to *"hang"*, and possibly render file contents inconsistent. To avoid such situations, the developer can specify a dependability requirement for the interaction pattern corresponding to open sessions, such that failure events are appropriately confronted. For example, the developer can require sessions to be atomic (*i.e.* every operation between establishing and closing a session should succeed, or the system should remove all partial products of the session), or failures occurring on an open session to be completely masked (*i.e.* despite failures, the client should be able to complete its operations over an open session and close the session without noticing possible failure events). Table 5.2 gives the formal expression of the dependability requirements concerning the atomicity and the failure masking properties.

In Table 5.2, the failure assumptions are captured by the conditions A.1 and A.2. The former condition expresses the fact that after a failure event, the entity where the failure occurred ceases all activities (*i.e.* the entity follows the *crash-processor* failure semantics). The latter condition expresses the fact that a message sent once cannot be delivered more than one time (*i.e.* the communication follows the *at most once* semantics, which include the case of message omission failures). Condition C.1 gives the dependability requirements concerning the system behavior in the presence of failure. More precisely, C.1 expresses the fact that a failure occurring in some object of the system should not prevent the system from reaching a state that can be found in the trace of a correct system execution (*i.e.* the system is capable to mask the occurrence of failures on its constituents). Table 5.2 also defines condition C.2, which expresses the fact that after a failure the system returns to its initial state (*i.e.* the system execution follow the atomic or *all-or-nothing* semantics).

In Aster, the developer associates the dependability requirements with some entities of the software system by incorporating them in the interface of the entity demanding them. For example, if a client that access files for reading, wishes to mask failures occurring during the *read()* and *lseek()* operations, it may declare in Aster the object *MyClientObj* given in Table 5.3. The Aster declarations follow the syntax of TINA-ODL [36]. The nterface construct groups methods and attributes like the nterface CORBA construct, and the bject construct declares executable entities that export a number of interfaces (declared in the upports clause), and import a number of interfaces (declared by the equires clause outside the *behavior* text). In addition, these entities may have some associated behavior which is

Label	**Condition**	**Comments**
A.1	$(b_f \in \mathcal{R}(\mathcal{X}) \wedge \sigma \in b_f \wedge faulty_{\mathcal{X}}(\sigma)) \Leftrightarrow$ $\forall i > f \mid ((b_i \in \mathcal{R}(\mathcal{X}) \Rightarrow b_i = \emptyset) \wedge (a_{i-1} \in \mathcal{X} \Rightarrow a_{i-1} = \emptyset))$	crash-processor
A.2	$\exists a_i \in \mathcal{X}(\Sigma_\alpha \bowtie \Sigma_\beta) \wedge xport(\alpha, \beta, D) \in a_i \Rightarrow$ $(\nexists a_j, a_k \in \mathcal{X}(\Sigma_\alpha \bowtie \Sigma_\beta) \mid (mport(\alpha, \beta, D) \in a_j) \wedge$ $(mport(\alpha, \beta, D) \in a_k) \wedge (j \neq k))$	omission-failure
C.1	$\forall \mathcal{R}(\mathcal{X}(\Sigma)) \mid (\exists \Sigma' \mid ((\mathcal{R}(\mathcal{X}^C(\Sigma')) \subseteq \mathcal{R}(\mathcal{X}(\Sigma))) \wedge (\Sigma' \equiv \Sigma)))$	failure-mask
C.2	$(\Sigma = (M, M_0, T)) \wedge (\mathcal{R}(\mathcal{X}(\Sigma)) = \langle b_\nu \rangle) \wedge$ $(\ \forall b_i \in \mathcal{R}(\mathcal{X}(\Sigma)) \mid (\ (\exists \sigma \in b_i \mid faulty_{\mathcal{X}}(\sigma)) \Rightarrow b_{i+1} = M_0\ )\ )$	atomic-execution

Table 5.2: The dependability requirements for failure masking

declared in the ehavior clause. The two object declarations on the left hand side of Table 5.3 refer to two DFS server entities, *MyServerObj-1* exporting the fsRDWR interface and *MyServerObj-2* exporting the fsPR and fsTEX interfaces, which do not import any interface and do not have any particular behavior requirements. *MyServerObj-1* also declares to provide the A.1 property from Table 5.2, for all the methods it exports. On the other hand, *MyClientObj* object does not export any operations since the interface it supports is empty, and reacts as a client to objects exporting the fsRD interface, or a compatible one (*e.g.* an interface that inherits from fsRD). In addition it declares certain behavior requirements in the ehavior clause. More precisely, it requires failure masking for both the *read()* and *lseek()* operations, under the assumptions given in Table 5.2.

The basic difference of the bject construct between TINA-ODL and Aster, is the fact that the former treats the string of the ehavior clause as comments, while Aster uses it to declare the various behavior requirements of the object in question. The keyword ster in the behavior string is used to mark the begin and the end of requirement declarations to be treated by Aster. The equires keyword in the behavior string is used to initialize the requirement declaration concerning a specific method in the interfaces previously declared. Finally, the ep keyword is used to notify Aster that the property that follows, concerns a dependability requirement. This guides Aster to use the CASE tool for the systematic dependability analysis presented in the previous chapter. If the ep keyword does not precede a property in the requirements declaration, then the standard Aster method for selecting the software components that satisfy a given requirement, is employed (see Chapter 2).

```
bject MyServerObj-1
  upports fsRDWR ;
  nitial fsRDWR ;
  equires ;   – nothing –
  ehavior
    "ster
      rovides fsRDWR::* : A.1 ;
    ster"
;


bject MyServerObj-2
  upports fsPR, fsTEX ;
  nitial fsPR ;
  equires ;   – nothing –
  ehavior " – none – "
;
```

```
nterface emptyIRF
  – Declares nothing –
;

bject MyClientObj
  upports emptyIRF ;
  nitial emptyIRF ;
  equires fsRD ;
  ehavior
    "ster
      equires fsRD::read :
        ep A.1 ∧ A.2 ∧ C.1 ;
      equires fsRD::lseek :
        ep A.1 ∧ A.2 ∧ C.1 ;
    ster"
;
```

Table 5.3: The Aster declarations of two server and one client objects

The Aster declarations of Table 5.3 prescribe an interaction between the objects *MyClientObj* and *MyServerObj-1*, under the condition that the dependability requirements of the first object are satisfied. The following section describes the Aster process for analyzing these requirements and selecting the appropriate software components that compose the fault tolerant mechanism that satisfies them.

5.2 Dependability Analysis

When the declaration of the object *MyClientObj* is fed in the Aster compiler, a number of parsing, verification, and analysis processes are started. The clause equires in the third line of *MyClientObj* declaration, expresses the fact that the object requires contacting an object offering the fsRD interface or compatible. This causes Aster to search for objects that export the requested interface. The only candidate object in Table 5.3 is *MyServerObj-1*, which exports the fsRDWR interface. fsRDWR is compatible to the requested one, since it inherits from fsRD. Then, given the pair of the client and server objects, the Aster environment proceeds to the requirement analysis of the designated bindings. The requirement analysis is based on the contents of declaration block delimited by the ster keyword inside the ehavior clause. These contents include the properties required and provided by the objects. In the remainder of this section, the systematic analysis of the

file system dependability requirements is followed step by step. In addition, the utilization of the analysis results for the retrieval of the software components that compose the desired fault tolerant mechanism is demonstrated.

5.2.1 Searching the Property Database

Given the declarations of *MyClientObj* and *MyServerObj-1* in Table 5.3, the goal of the requirement analysis performed by Aster, is to select a base *bus*, *i.e.* communication platform to permit the interaction between the two objects, and the set of software components that customize the execution platform to meet the objects requirements. In the current Aster prototype, the only available communication platform is CORBA. Hence, the role of the requirement analysis is restrained to identifying the software components that customize CORBA to meet the application requirements. To do this, Aster aggregates all properties *required* by the application objects and the base bus, and tries to satisfy them with the properties *provided* by the application objects and the base bus. In a second requirement analysis phase, for the required properties that cannot be satisfied by the provided ones, Aster searches software components in its repository that provide properties matching the unsatisfied requirements.

In the Aster environment, base buses are registered as special objects, which do not import or export any interface, but they may provide and require some properties. The CORBA bus does not have any requirements, but it provides *at-most-once* communication semantics (condition A.2 in Table 5.2). On the other hand, from the application objects, *MyServerObj-1* provides the property captured by condition A.1 in Table 5.2, and *MyClientObj* requires the property captured by condition C.1 in the same table. To summarize, in the potential interaction of *MyClientObj* with *MyServerObj-1* over the CORBA bus, the first object requires the property A.1 $\wedge$ A.2 $\wedge$ C.1 and the second and third objects provided the aggregated property A.1 $\wedge$ A.2. During the first analysis phase, the requirements considered as failure assumptions (*e.g.* A.1 and A.2) are found to be satisfied by the provided properties. Hence, Aster performs the second analysis phase to analyze only the property C.1 and to use the analysis results to retrieve the software components that provide it.

The formulas expressing the requirements associated to the *read()* and *lseek()* operations of the *MyClientObj* object are preceded by the keyword ep. This indicates that the requirements are related to the dependability aspects of the system. In this case, instead of employing the standard logical tool, the CASE tool that performs the systematic dependability analysis is engaged. This tool employs the *Retrieve()* algorithm of Table 4.4 at page 52, with argument the property C.1 capturing the dependability requirement that needs to be analyzed. Then, C.1 is passed as argument to the *Sieve()* algorithm given in the same table. The *Sieve()* algorithm performs two filtering processes. The first is based on the *Place()* algorithm given in Table 4.3 at page 48, and aims at eliminating the dependability properties that do not imply the dependability requirement. The second is based on the *GetImpl()* algorithm given in Table 4.4, and it identifies from the selected properties, those that are implemented by some software component in the Aster repository. The results of the overall *"sieving"* process give the dependability property records that will

guide the software retrieval of the components composing the fault tolerant mechanism, which meets the initial dependability requirements.

Place(C.1, Dependability)	=	STRONGER
Place(C.1, Availability)	=	INDEPENDENT
Place(C.1, Safety)	=	INDEPENDENT
Place(C.1, Reliability)	=	STRONGER
Place(C.1, Masking)	=	STRONGER
Place(C.1, Active $\wedge$ Diffuse)	=	WEAKER
Place(C.1, MaskingObj $\wedge$ Detection)	=	WEAKER

Table 5.4: Results of the calls to the *Place()* algorithm during the analysis of C.1

Table 5.4 gives a summary of the calls to the *Place()* algorithm and the returned values. Only the dependability properties for which *Place()* returned EQUIVALENT or WEAKER, are considered in the next phase of the *Sieve()* algorithm; these are the properties that imply logically the dependability requirement. In the second phase, *GetImpl()* filters from the selected properties, those that are not being implemented by some software component in the Aster repository. The results of the second phase of the *Sieve()* algorithm reveal the software components that should be retrieved.

Figure 5.2 gives the graphical representation of the dependability properties hierarchy, where shadowed nodes host properties implemented by some software component in the Aster repository. Correlating the refinement relations revealed by this figure with the results given in Table 5.4, the following conclusion is reached: the properties in the classification scheme that satisfy the dependability requirement expressed by C.1, are the *Active* $\wedge$ *Diffuse* and the *MaskingObj* $\wedge$ *Detection* properties, and all their refinements. However, besides *Diffuse* which occupies a shadowed node, none of the *Active, MaskingObj*, and *Detection* is selected by *GetImpl()*. Instead, their refinements are examined for identifying implemented properties that satisfy the dependability requirement.

Active has a single refinement, which is expressed by the conjunction of the properties *Filter, AtomicDelivery,* and *Replication*. All three of these properties are implemented, hence the single refinement of *Active* along with the *Diffuse* record are selected by *GetImpl()*. On the other hand, the second property that satisfies C.1 is a conjunction of two properties occupying DAG nodes: the *MaskingObj* and *Detection* properties. In such cases, *GetImpl()* is called to find at least one implementation for each of the conjunction terms or their refinements, in order to select the initial property expressed as a conjunction. Concretely, *GetImpl()* is first called for the refinement of *MaskingObj*, expressed by property *Semi-Active*. The latter property does not occupy a shadowed node, hence it is not implemented by some software component in the Aster repository. So, *GetImpl()* is recursively called for each of its refinements. *Semi-Active* has a single refinement expressed by the conjunction of the properties *AtomicDelivery, Replication,* and *Leader*. All of these properties occupy shadowed nodes, and hence have an implementation in the

Aster repository. Hence, these three properties are provisionally marked as selected. In a similar manner, the properties *Replication, StableStorage* and *Restore* are provisionally selected, since their conjunction forms a refinement of *Passive*, which is itself a refinement of *MaskingObj*.

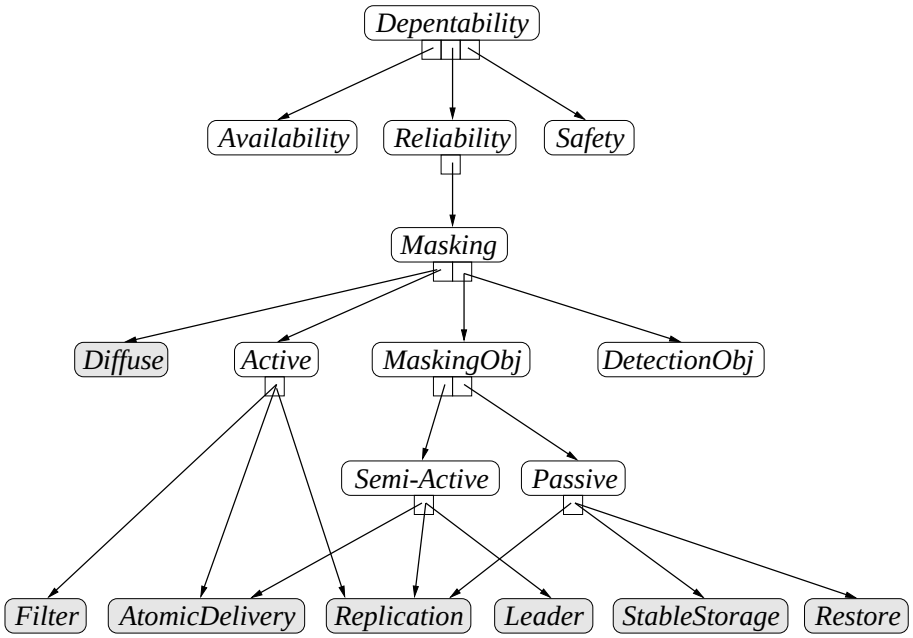


Figure 5.2: The classification scheme, where shadowed nodes host implemented properties.

The final selection of the provisionally selected properties is settled once at least one implementation for the other conjunction term (*i.e. Detection*) is found. According to the contents of Figure 5.2, there is no implementation available in Aster for the *Detection* property. *GetImpl()* is called for the single refinement of *Detection*, the *DetectionObj* property. The latter does not occupy a shadowed node, and hence it has no implementation in the Aster repository. So, *GetImpl()* is recursively called for its refinement. However, there does not exist a known refinement of *DetectionObj* in the classification scheme depicted in Figure 5.2. Hence, there is no implementation available for the second conjunction term of $MaskingObj \wedge Detection$. As a result, the implementations of the first conjunction term that were provisionally selected, cannot be used to satisfy the dependability requirement C.1. Consequently, the corresponding records are not selected to guide the software retrieval.

5.2.2 Software Retrieval

At this point, the dependability analysis is finished. Given the results of *Place()* and *GetImpl()* mentioned above, the *Retrieve()* algorithm selects only the records correspond-

ing to properties *Diffuse, Filter, AtomicDelivery,* and *Replication.* The omponents field of these records can be used to find and retrieve the corresponding software components from the Aster repository. The retrieved components are used for the customization of the middleware, according to requirements of the object *MyClientObj* given in Table 5.3. The Aster generator takes the configuration instructions derived from the architectural descriptions accompanying the aforementioned records, and uses them to incorporate the selected software components in software system. The following section focuses on the composition of the architectural descriptions associated to the selected properties, in order to obtain the overall architecture of the dependable system, which serves as configuration instructions for the Aster generator.

5.3 Deriving the Overall Dependable Architecture

The previous section stressed one of the practical benefit of this thesis, which is the minimization of the effort needed by the Aster environment to select the software components composing a fault tolerant mechanism that meets the dependability requirements of a software system. This section focuses on the benefits from the software architecture standpoint. More precisely, this section discusses how a dependability requirement is transformed into an architectural description of the system which incorporates the fault tolerant mechanism that meets the dependability requirement.

As it has been already mentioned, the records of the dependability properties stored in the classification scheme proposed by this thesis, contain also some information about the architectural impact caused by the property in question. Table 5.5 gives a number of architectural descriptions. These correspond to the properties in the DAG path that was traversed until finding the properties that satisfy C.1 and have an implementation in the Aster repository too. Using these descriptions, the software architect can derive a number of successive architectural refinements of the DFS structure. The last of those refinements provides the structure of the DFS that directly maps software components to architectural entities. Figure 5.3 illustrates graphically this derivation.

The architectural descriptions captured by the Dependability_arch, Reliability_arch, and Masking_arch declarations in Table 5.5, do not cause any structural impact on the system; they only associate a property to it. Hence, the corresponding software architecture is the one depicted by drawing `(i)` in Figure 5.3. Drawings `(ii)` and `(iii)` give respectively the architectural impact described by Active_arch and Diffuse_arch, where shadowed boxes denote generic architectural components. In the same way one obtains the drawings `(iv)`, `(v)` and `(vi)` corresponding to the architectural descriptions Replication_arch, Filter_arch, and AtomicDelivery_arch respectively.

Sketching the first six drawing of Figure 5.3 from the corresponding architectural description given in Table 5.5, is a straight-forward task that can be automatically performed by appropriate CASE tools. However, aggregating these drawings to obtain the overall software architecture of the software system plus the fault tolerant mechanism, is more complicated. Human intervention is necessary to assemble the individual drawings into the structural description of the dependable software system. In the DFS example studied

Dependability_arch **{**
Nodes a
Generic – *none* –
Links – *none* –
Property `Dependability(Σ)`
}

Masking_arch **{**
Nodes a
Generic – *none* –
Links – *none* –
Property `Masking(Σ)`
}

Diffuse_arch(N) **{**
Nodes d
Generic a[N]
Links for i=1 **to** N (d, a[i]),
(ϵ, d)
Property `Diffuse(`Σ_a`, N)`
}

AtomicDelivery_arch(N) **{**
Nodes b[N]
Generic a[N]
Links for i=1 **to** N (ϵ, b[i]),
for i=1 **to** N (b[i], a[i])
Property `AtomicDelivery(G)`
}

Reliability_arch **{**
Nodes a
Generic – *none* –
Links – *none* –
Property `Reliability(Σ)`
}

Active_arch(N) **{**
Nodes a[N]
Generic – *none* –
Links for i=1 **to** N (a[i], ϵ),
for i=1 **to** N (ϵ, a[i])
Property `Active(`Σ_a`, N)`
}

Replication_arch(N) **{**
Nodes – *none* –
Generic a[N]
Links – *none* –
Property `Replication(G)`
}

Filter_arch(N) **{**
Nodes f
Generic a[N]
Links for i=1 **to** N (a[i], f),
(f, ϵ)
Property `Filter(G)`
}

Table 5.5: The architectural descriptions related to the customization of the DFS

in this chapter, the dependability requirement C.1 is satisfied by the conjunction of the properties *Diffuse, Filter, AtomicDelivery,* and *Replication.* Hence, the software architecture of the dependable DFS arises from the aggregation of the corresponding architectural descriptions.

Drawing `(vii)` in Figure 5.3 gives the result of assembling the architectural description corresponding to *Filter, AtomicDelivery,* and *Replication.* This drawing is derived by simply over-posing the drawings `(iv)`, `(v)`, and `(vi)`. On the other hand, to derive drawing `(viii)` from the assembly of drawings `(iii)` and `(vii)` is somewhat more difficult. In this case, each a[i] component of drawing `(iii)` should be mapped to the composite

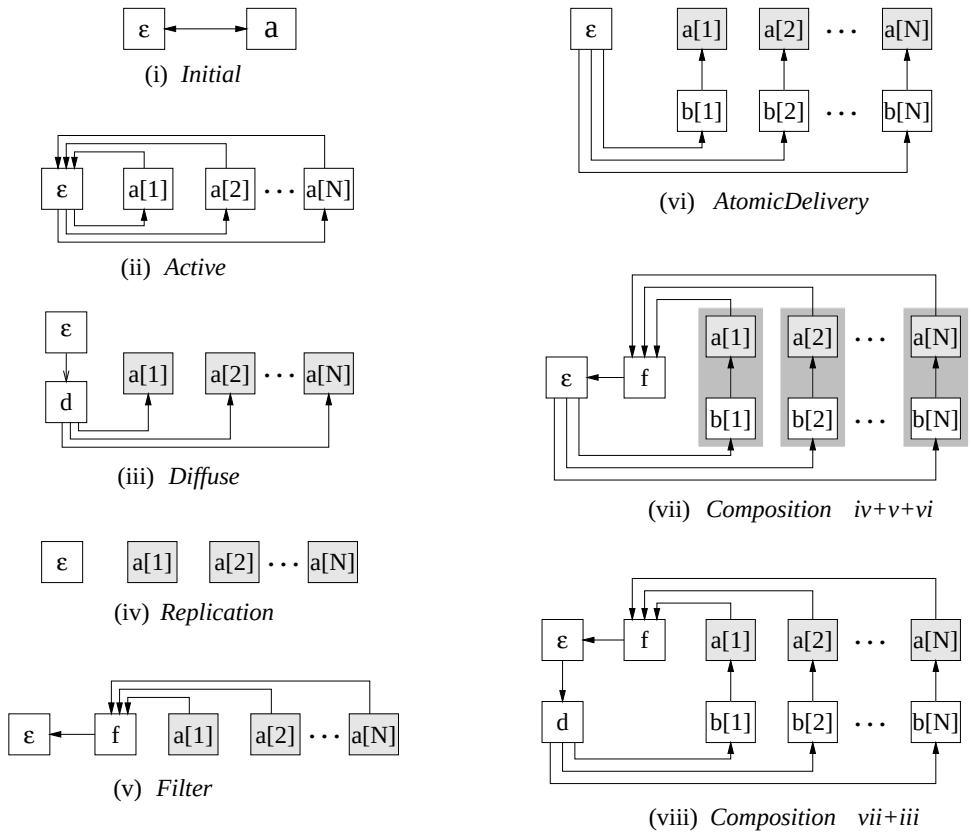


Figure 5.3: Deriving the software architecture of DFS that meets C.1

component ⟨a[i], b[i]⟩, depicted by the shadowed regions in drawing `(vii)`. So, the last drawing of Figure 5.3 gives the software architecture of the software system customized to provide fault tolerant access to DFS server components (named a[i] in drawing `(viii)` of Figure 5.3).

Notice that the selection of the component implementing the *Diffuse* property would not be necessary if the execution platform was supporting some multicasting means. For example, the IONA product that supports multicasting is OrbixTalk. If this product was available to the Aster environment as a base bus, then the software architecture of the dependable DFS (excluding the base bus) would be given by drawing `(vii)` from Figure 5.3, and not by drawing `(viii)`.

5.4 Summary

The purpose of this chapter was to demonstrate how the results of this thesis facilitate the development of software systems customized to meet some dependability requirements. The benefits can be summarized in two categories:

- The transformation of the formal specification of the dependability requirements into: *(i)* a key for the software retrieval of the appropriate fault tolerant mechanism, and *(ii)* a description of the software architecture that describes the composition of the initial software system with the selected fault tolerant mechanism.
- The acceleration of the software retrieval performed by Aster, when it is aimed at finding the fault tolerant mechanism that satisfies the application dependability requirements.

In the example studied in this chapter, the dependability requirement expressed by property C.1 of Table 5.2, was transformed into a dependable DFS customized to provide fault tolerant server access, whose architecture structure is depicted in drawing `(viii)` of Figure 5.3. In addition, the dependability analysis that preceded the software retrieval, decreased the cost of the otherwise exhaustive search to a minimum, by employing the Aster retrieval process on the set containing exactly the properties satisfying the dependability requirements and also being implemented by some software component in the Aster repository.

Chapter 6

Conclusions

The concluding chapter of this thesis starts with a brief summary of the report contents, and a reminder of the principal contributions of the thesis. These prepare the ground for the presentation of some important issues in the development of dependable software systems, which have not been addressed in the context of this thesis. The presentation includes a discussion on the importance of the open issues, some approaches that may lead to robust solutions of the raised problems, and the outline of the future work.

6.1 Summary

The dependability requirements of a system express the expected behavior in the presence of failures. By analyzing them, one may obtain a description of the fault tolerant mechanism that should be employed to provide the requested behavior. However, guarantees need to be provided concerning the robustness of the dependability analysis and the correctness of the indicated fault tolerant mechanism with respect to the initial dependability constraints. An appealing solution is to employ the formal specifications of the dependability properties, which permits their rigorous analysis. The formalization of dependability properties and an accompanying refinement framework, provide the necessary support for transforming a system specification and a dependability requirement associated to it, into the blueprint of the fault tolerant mechanism that meets a given dependability requirement.

The refinement of dependability properties is coupled with their architectural impact on the system structure. The aggregate knowledge of how to refine a dependability property, and what effect this refinement has on a system specification, is captured in a classification scheme. This scheme forms the basis for the systematic analysis of the dependability requirements of a software system. The outcome of the systematic dependability analysis, is a set of specifications that compose a fault tolerant mechanism, which meets the initial dependability requirement. Moreover, the dependability analysis outcome includes the instruction for interconnecting the aforementioned specifications among them and incorporating them in the system architecture. In addition, the results of the dependability analysis are combined with the contents of the software repository of middleware components provided by the Aster development environment [32]. This provides support for

guided software retrieval of fault tolerant mechanisms, in the Aster context. The practicality of the presented approach is demonstrated by an example, which treats the development of a dependable version of a distributed file system [30].

6.2 Contributions

The principal contributions of this thesis, are the framework for the formalization and the refinement of dependability properties, and the association of a dependability property with its impact on the system architecture. A classification scheme stores the refinement patterns, which capture the knowledge of analyzing dependability requirements, and of deducing the associated architectural impact. Based on the classification scheme, a CASE tool has been proposed, to perform the analysis the system's dependability requirements, and to use the analysis results for guiding the retrieval of the appropriate fault tolerant mechanism. The CASE tool is coupled with tools from the Aster prototype, used for the software retrieval, and for the assembly of the retrieved components with the components of the initial system. The proposed approach for the systematic dependability analysis, is shown to be an appealing suggestion for the robust development of dependable systems. In particular, it is shown to facilitate the customization of fault tolerant middleware, to meet the system's dependability requirements.

The originality of the contributions of this thesis, lies in the integration of the requirement analysis concerning the dependability properties of a system, with the architectural analysis of the system from the dependability viewpoint. The proposed dependability analysis does not offer a technological break-through to the development of dependable systems. Rather, it is built on the results of existing technologies from the fields of fault tolerance, formal specifications, requirement analysis, software architecture, software reuse, and configuration-based distributed systems. The innovation in this thesis, is the combination of these existing technologies, in such a way so as to provide a useful support for the robust development of dependable systems.

6.3 Open Issues

One important issue that was not addressed in this report, is the quantitative description of the fault tolerant mechanisms selected to satisfy system's dependability requirements. The presented formalization of dependability properties, describes in a qualitative manner the behavior of a system in the presence of failures. Otherwise stated, it describes the type of failures that should be tolerated and the dependability-specific components used to meet the system's dependability requirements. However, there does not exist a mechanism capable of providing its designated dependability guarantees in absolutely any failure scenario. For example, when N member failures occur in a group of N replicas, the group of replicas can no longer provide any service. To describe the capability of a component to perform its designated service a stochastic value is assigned to it, which is interpreted as the component's dependability degree, *i.e.* the probability that the component is operational. In the same manner, a stochastic value can be used to describe the degree of the required

dependability. Gray and Siewiorek [23] give a summary of the various characterizations associated to dependable systems according to their dependability degree.

The need for a quantitative description of the fault tolerant mechanisms, implies that the dependability analysis proposed in this report, should be followed by a stochastic analysis, which will provide the configuration parameters for the selected fault tolerant mechanism. Such configuration parameters include the number of replicas that should be instantiated, the frequency of the checkpoints, the size of the error correction codes, *etc.* Obviously, the results of this stochastic analysis are essential for the correct customization of the dependable middleware. However, the stochastic analysis does not influence the requirement and architectural analysis. The systematic dependability analysis presented in this report, can be used to derive the dependable software architecture of a system from its dependability requirements. Then, the stochastic analysis can be applied on the dependable architecture, to infer its quantitative characteristics that guarantee the requested dependability degree. Issues related to the stochastic analysis and the combination of its results with the results of the dependability analysis, belong to the future work in the Aster context.

Besides the stochastic analysis, a number of other consideration should be verified before being able to claim that the right fault tolerant mechanism has been selected and configured for a given software system. Examples of such considerations are:

- A complexity analysis, which can provide a theoretical measure for the performance of the mechanism.
- An analysis of the implementation characteristics, that will reveal the most adequate implementation of the software components to be used (*e.g.* for timeouts that require great precision, a timer based on alarm events scheduled by the operating system is expected to be much more robust and efficient than a timer which verifies the elapsed time and executes as a user-level thread).
- A composability and an executability analysis, which verifies whether the composition of the selected middleware components can be incorporated in the software system, and executed over some specific execution platform. This problem was not raised in the case study of Chapter 5, because the dependability-specific middleware components were implemented for being reusable in the Orbix execution environment used by the Aster prototype. However, to support the utilization of the various components that populate the Aster middleware repository, the composability and executability analyses are necessary.

Similarly to the stochastic analysis, such issues are independent from the proposed dependability analysis that leads to the selection of some specific fault tolerant mechanism. Hence, they have not been addressed in this report.

Finally, another issue that plays an important role in the development of software systems, is the interference of analysis results concerning different quality aspects of the system. For example, in step 3 of Figure 4.5 in page 53, the outcome of the dependability analysis is feed to the software retrieval and assembly Aster tools, only if its combination with the constraints regarding other system aspects (*e.g.* security, efficiency, timeliness, *etc*)

does not produce any conflicts. Aster has been working towards a multi-view description of the system architecture, where each different system aspect contributes to the overall software architecture. However, a systematic way to combine the architectural impacts of the different views, has not been studied yet.

6.4 Epilogue

The thesis presented in this report, proposed a method for the systematic analysis of the system's dependability requirements. The result of the dependability analysis is the retrieval of the software components composing a fault tolerant mechanism that satisfies the dependability requirements, and a set of instructions for incorporating the selected mechanism in the system architecture. The verifiable correctness of the requirement analysis and the architectural integration of the fault tolerant mechanism and the software system, contribute to the robustness of the development process. Although the proposed approach has been tightly coupled with the dependability system aspect, it worths to be mentioned that the only link of the dependability aspects with the requirement and the architectural analysis integration, is the existence of a refinement relation among dependability properties. If a similar refinement relation could be identified for the properties related to some other system aspect (*e.g.* for transactional system properties [76]), then the same approach for the requirement and architectural analysis can be applied. This will not provide a solution to the combination of the results concerning the requirement and architectural analysis of various system aspects. However, it would provide the same conceptual (*i.e.* requirement analysis combined with architectural analysis to guide the retrieval of middleware components) and material (*i.e.* the tools for the requirement analysis) framework for the analyses related to viewpoints on different quality aspects of software systems.

Acknowledgements

This report is a resumé of the author's PhD dissertation entitled "Développement Robuste de Systèmes Sûrs de Fonctionnement", Université de Rennes I, March 1999.

The author is deeply indebted to the members of his thesis committee: Prof. Jean-Pierre Banâtre (Université de Rennes 1, France), Prof. Charles Consel (Université de Rennes 1, Rennes, France), Dr. Paul Le Guernic (INRIA, Rennes, France), Dr. Valérie Issarny (INRIA, Rennes, France), Prof. Jeff Magee (Imperial College, London, UK), and Prof. Santosh Srivastava (University of Newcastle upon Tyne, Newcastle, UK).

Appendix A

Glossary

NOTICE: This glossary is provided as an aid for reading this document, and should not be considered independently of it.

Atomicity ..... (of action) Either the action is successfully completed or not performed at all, but in any case no partial action results appear.

Availability ..... The property of a piece of software to retain accessible its functionalities described in its specifications, despite the occurence of failures. Otherwise stated, *"a measure of the delivery of correct service with respect to the alternation of correct and incorrect service"* [41].

CASE ..... Initials standing for the phrase *Computer Aided Software Engineering*, which is employed to designate a research area working on techniques and tools for automating the software development process.

Concistency ..... (of action) The action represents a correct state transition which preserves the state invariants.

Dependability ..... A quality that refers to the capacity of a piece of software to correctly deliver the functionalities described in its specifications. Otherwise stated, *"the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers"* [41].

Failure ..... An observed deviation of the behavior of a system from its specification. Otherwise stated, *"transition from correct service delivery to incorrect service delivery"* [41].

Fault Tolerance ..... A set of means (concepts, abstractions, algorithms, techniques, and methods) for coping with the occurence of failures. Otherwise stated, *"methods and techniques aimed at providing a service complying with the specification in spite of faults"* [41].

Isolation ..... (of action) The execution and the results of the action are not influenced by other concurrently executing action; it rather appears that actions executed sequentially according to some global clock.

Pattern ..... The codification of a section of an artifact or a process, with repetitive nature. Otherwise stated, *"the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts"* [59].

Protocol ..... The specifications of some means that allow a certain kind of interaction among a set of pieces of software. Otherwise stated, *"the set of rules that software components on different machines use to realize a given abstraction"* [49].

Refinement ..... A process that renders more concrete an artifact. It is employed in the phrases *"property refinement", "specification refinement",* and *"architecture refinement"*. The first refinement applies on a property and produces a stronger one. The second applies on a pair of a specification and an associated property and produces a set of specification - property pairs such that the initial specification is subordinate to the composition of the resulting ones and the conjunction of the resulting properties with the properties modeled by the resulting specifications imply the initial property. The third refers to the architectural impact on the structure of an object composition, caused by a specification refinement.

Reliability ..... The property of a piece of software to correctly deliver its functionalities as described in its specifications, despite the occurence of failures. Otherwise stated, *"a measure of the continuous delivery of correct service, or equivalently, of the time to failure"* [41].

Safety ..... The property of a piece of software to preserve a correct state with respect to its specifications, despite the occurence of failures. Otherwise stated, *"a measure of continuous safeness, or equivalently, of the time to a catastrophic failure"* [41].

Specification ..... A precise description of the environmental conditions under which a piece of software is developed to function correctly, and of the functionalities that it should deliver under these conditions. Otherwise stated, *"a contractual constraint on the behavior of a module"* [39].

Bibliography

[1] *Proceedings of the 12th International Symposium on Fault Tolerant Computing*, 1982.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] H.-R. Aschmann, N. Giger, E. Hoepli, P. Janak, and H. Kirrmann. Alphorn: A Remote Procedure Call Environment for Fault-Tolerant, Heterogeneous, Distributed Systems. *IEEE Micro*, 11(5):16–19 & 60–66, October 1991.

[4] S. Baker. *CORBA Distributed Objects Using Orbix.* ACM Press & Addison-Wesley, 1997.

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

[6] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the Symposium on Software Reusability*, pages 27–37, April 1995.

[7] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[8] R. Ben-Natan. *CORBA: A Guide to Common Object Request Broker Architecture.* Computing Series. McGraw-Hill, 1995.

[9] C. Bidan. *Sécurité des Systèmes Distribuées: Apport des Architectures Logicielles.* Thèse de doctoral, Université de Rennes `I`, May 1998.

[10] C. Bidan and V. Issarny. Security Benefits from Software Architecture. In *Proceedings of the 2nd International Conference COORDINATION'97*, pages 64–80, September 1997.

[11] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 35–42, May 1998. Available at http://www.irisa.fr/solidor/work/aster.html.

[12] K. P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 79–86, 1985.

[13] A. Cau and W.-P. de Roever. Using Relative Refinement for Fault Tolerance. In *Proceedings of the 1st International Symposium of Formal Methods Europe, FME'93*, number 670 in Lecture Notes in Computer Science, pages 19–41. Springer-Verlag, 1993.

[14] L. Chen and A. Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th International Symposium on Fault Tolerant Computing*, pages 3–9, 1978.

[15] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transaction on Database Systems*, 19(3):450–491, September 1994.

[16] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[17] OMG Document. CORBAservices: Common Object Services Specification. Technical report, Object Management Group, December 1998. http://www.omg.org/corba/-csindx.htm.

[18] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[19] W. B. Frakes and S. Isoda. Success Factors of Systematic Reuse. *IEEE Software*, 11(5):15–19, September 1994.

[20] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185, December 1994.

[21] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6):17–26, June 1995.

[22] J. A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2):16–28, February 1986.

[23] J. Gray and D. P. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[24] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, April 1997.

[25] M. A. Hiltunen. Configuration Management for Highly-Customizable Services. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 197–205, May 1998.

[26] M. A. Hiltunen and R. D. Schlichting. An Approach to Constructing Modular Fault Tolerant Protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, 1993.

[27] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall International, 1985.

[28] V. Issarny and C. Bidan. Aster: a CORBA-based Interconnection System Supporting Distributed System Customization. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 194–201, May 1996.

[29] V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 586–593, May 1996.

[30] V. Issarny, C. Bidan, and T. Saridakis. Designing an Open-ended Distributed File System in Aster. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, pages 163–168, September 1996.

[31] V. Issarny, C. Bidan, and T. Saridakis. Aster: Un système de configuration distribuée au-dessus de CORBA. *L'Objet*, 3(2), June 1997.

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

[33] V. Issarny, C. Bidan, and T. Saridakis. Characterizing Coordination Architectures According to Their Non-Functional Execution Properties. In *Proceedings of the 31st Hawaii International Conference on System Science*, pages 275–283, January 1998.

[34] V. Issarny, T. Saridakis, and A. Zarras. Multi-View Description of Software Architectures. In *Proceedings of the 3rd ACM SIGSOFT International Software Architecture Workshop*, pages 81–84, November 1998.

[35] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, May 1994.

[36] B. Kitson, P. Leydekkers, N. Mercouroff, and F. Ruano. TINA Object Definition Language. Technical Report TR_NM.002_1.3_95, TINA-C Document, 1995.

[37] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[39] L. Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

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

[41] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[42] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Software Series. Prentice-Hall, 1981.

[43] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transaction on Software Engineering*, 21(4):336–355, April 1995.

[44] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14, October 1996.

[45] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[46] D. Le Métayer. Software Architecture Styles as Graph Grammars. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23, October 1996.

[47] R. Mili, R. Mittermeir, and A. Mili. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1997.

[48] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[49] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol Based on Partial Order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, pages 309–331. Springer-Verlag Wien New York, 1992.

[50] S. Mishra and R. D. Schlichting. Abstractions for Constructing Dependable Distributed Systems. Technical Report TR 92-19, The University of Arizona, August 1992.

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

[52] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3), 1995.

[53] D. E. Perry. Version Control in the Inscape Environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, March 1987.

[54] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[55] M. L. Powell and D. L. Presotto. PUBLISHING: A Reliable Broadcast Communication Mechanism. In *Proceedings of the 9th International Symposium on Operating System Principles*, pages 100–109, 1983.

[56] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[57] J. M. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[58] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2), 1975.

[59] D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems (John Wiley)*, 2(1):3–13, April 1996.

[60] T. Saridakis, C. Bidan, and V. Issarny. A Programming System for the Development of TINA Services. In *Proceedings of the International Conference on Open Distributed Processing*, pages 3–14, May 1997.

[61] T. Saridakis and V. Issarny. Fault Tolerant Software Architectures. Research Report No.3350, INRIA, January 1998.

[62] T. Saridakis and V. Issarny. Developing Dependable Systems Using Software Architecture. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83–104, February 1999.

[63] T. Saridakis, V. Issarny, and C. Bidan. Customized Remote Execution of Web Agents. In *Proceedings of the 31st Hawaii International Conference on System Science*, pages 614–620, January 1998.

[64] F. B. Schneider. Abstractions for Fault Tolerance in Distributed Systems. Technical Report TR 86-745, Department of Computer Science – Cornell University, Ithaca, New York 14853, April 1986.

[65] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Computer Surveys*, 22(4):299–319, December 1990.

[66] J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, pages 246–254, November 1997.

[67] M. Shaw, R. DeLine, D. Kelin, T. Ross, D. Young, and G. Zelesnik. Abstraction for Software Architectures and Tools to Support Them. *IEEE Transaction on Software Engineering*, 21(4):314–335, April 1995.

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

[69] A. Spector and D. Gifford. A Computer Science Perspective of Bridge Design. *Communications of the ACM*, 29(4):267–283, March 1986.

[70] M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall, 1992.

[71] S. D. Stoller and F. B. Schneider. Automated Analysis of Fault-Tolerance in Distributed Systems. Technical Report TR 96-1614, Department of Computer Science – Cornell University, Ithaca, New York 14853, November 1996.

[72] D. C. Sturman and G. A. Agha. A Protocol Description Language for Customizing Failure Semantics. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 148–157, October 1994.

[73] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Publishers, 1989.

[74] B. L. Di Vito and R. W. Butler. Provable Transient Recovery for Frame-Based, Fault-Tolerant Computing Systems. In *Proceedings of the 13th IEEE Symposium on Real Time Systems*, pages 275–278, October 1992.

[75] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

[76] A. Zarras and V. Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of MIDDLEWARE'98*, pages 257–272, September 1998.



Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399