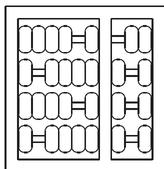

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Architecture-Centric Fault Tolerance with Exception Handling

Patrick Henrique da Silva Brito
Rogério de Lemos *Fernando Castor Filho*
Cecília Mary Fischer Rubira

Technical Report - IC-07-04 - Relatório Técnico

February - 2007 - Fevereiro

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Architecture-Centric Fault Tolerance with Exception Handling

Patrick H. da S. Brito
Institute of Computing
State University of Campinas
Campinas, SP, Brazil
pbrito@ic.unicamp.br

Rogério de Lemos
Computing Laboratory
University of Kent
Canterbury, UK
r.delemos@kent.ac.uk

Fernando J. Castor Filho
Cecília M. F. Rubira
Institute of Computing
State University of Campinas
Campinas, SP, Brazil
{fernando, cmrubira}@ic.unicamp.br

Abstract. *This technical report considers the problem of developing dependable component-based software systems through an architectural approach, which combines fault prevention, fault removal, and fault tolerance techniques. The architecture-centred solution comprises a rigorous approach, which systematises the verification and validation of fault tolerant systems. Using B-Method and CSP, we analyse the exception flow at the architectural level and verify important properties regarding the system dependability. Besides that, the it is adopted an architectural solution based on exception handling for transforming untrusted software components into idealised fault-tolerant architectural components, which can be used as building blocks for creating fault-tolerant software architectures. The feasibility of the proposed architectural solution was evaluated on a business critical case study.*

Keywords: *Dependable systems, rigorous software development, fault-tolerant software architecture, formal specifications, model checking.*

1 Introduction

In the modern world, software systems are used in a variety of applications where the price of failure is simply too high. Software systems that can cause risks to human lives or great financial losses are usually made fault-tolerant, so that they are capable of providing their intended service, despite the presence of failures. The growing pervasiveness and complexity of software systems with strict dependability requirements demands techniques and tools

that leverage the development of applications that are both fault-tolerant and easy to build and maintain.

Amongst the several existing techniques for building fault-tolerant systems, exception handling is a well-known mechanism for structuring error recovery in software systems [15]. Since exception handling is an application-specific technique, it promotes the implementation of very specialised and sophisticated error recovery measures by complementing other techniques, such as atomic transactions [25]. Furthermore, in applications where a rollback is not possible, such as those that interact with physical environments, exception handling may be the only choice available. On the other hand, it is also accepted that exception handling mechanism has its disadvantages, if we consider the fact that a large part of a system's code is devoted to error detection and handling [15, 31, 38]. One reason for this is that developers tend to focus on the normal behaviour of applications and only deal with the code responsible for error detection and handling at the implementation phase. Although extremely complex, this part of the code is usually the least understood, tested and documented [15, 31], which tend to decrease the overall dependability of software systems. A claim recently being made is that, to achieve the desired levels of dependability, mechanisms for detecting and handling errors should be systematically incorporated from early phases of software development [21, 32, 33].

In this technical report, we present an approach for structuring fault-tolerant software systems so that mechanisms for detecting and handling errors have minimal impact on the overall system complexity. The motivation for this work is twofold. First, it is widely accepted that the architecture of a software system has a strong impact on its capacity to meet its intended quality requirements [5]. And second, it is also accepted that dependability of a software system is inherently associated with its structure [30]. The contribution of this work is on the provision of structured means for handling inconsistencies among architectural elements in the context of software component failures. The approach is based on exceptional handling, and relies on the peer-to-peer architectural style for incorporating the appropriate adaptors that allow a clear separation between normal and abnormal behaviours within a component. This paper will be focusing into three major issues: the formal modelling and verification of the architectural abstraction using a combination of the B-Method and CSP, the definition of architectural properties that have to be verified when instantiating the architectural abstraction, and the formal analysis of exception propagation at the architectural level.

The rest of this document is organised as follows. The following section presents a brief background in the area of architectural fault tolerance based on exception handling. Section 3 describes the architectural solution based on the *idealised fault-tolerant architectural element* (iFTE). Section 4 presents a rigorous development approach, which was addopted by us to development software systems with strict dependability requirements. Section 5 presents a tool, constructed to identify the possible scenarios of formal models, which is useful for verification purpose of the iFTE elements. The formal verification of the iFTE and the formal analysis of architectural exceptions propagation are presented in Section 6. In Section 7 the feasibility of the overall approach was evaluated in the context of a business critical application. Finally, the last section provides some concluding remarks and future directions of research.

1.1 Related Work

Fault tolerance at the architectural level has received considerable attention lately, mostly in the context of fault handling. In particular, issues related to architectural reconfiguration that includes replacing, adding, removing architectural elements, or changing the topology of the configuration [7]. However, the same cannot be said in the context of error handling when defining novel structuring techniques for error confinement, or mechanisms for removing errors from the system state.

One of such contributions is the idealised C2 component (iC2C) [17], based on the idealised fault-tolerant component [2], has been proposed for structuring software systems compliant with the C2 architectural style [36]. The internal protocol followed by the internal elements of an iC2C enforces error confinement and makes it possible to define multiple exception handling contexts at the architectural level. Later work by Castor et al [11] defined and implemented an architectural level exception handling mechanism based on the concept of iC2C. The work presented in this paper can be seen as an extension of the iC2C for a broader class of software architectures that adhere to the peer-to-peer architectural style [13].

Recent work by Castor et al [10], in the Aereal framework, leverages existing languages and tools to support the description and analysis of exception flow in software architectures adhering to multiple architectural styles. This work is similar to ours in its goal, but does not model the interplay between an architectural element and the error recovery mechanisms that make it fault-tolerant. Hence, it ignores exceptions that are successfully handled internally to an architectural element. Another important difference is the possibility on the proposed approach to specify specific behaviour related to error and fault handling, for example the procedure of a successful masking error. These behavioural restrictions reduce the number of states generated during the architecture verification, consequently increasing the scalability of the proposed approach.

Seminal work by Issarny and Banâtre [27] describes an extension to architectural description languages that allows the specification of architectural invariants. Violations of these invariants, called configuration exceptions, trigger architectural reconfigurations. This work differs from ours because it emphasises fault handling at the architectural level. Our work, on the other hand, emphasises error recovery.

Some researchers have proposed frameworks [34, 37] that support the detection and handling of exceptions in component-based platforms, such as J2EE. These frameworks provide means to introduce in a non-invasive way error detection mechanisms, such as monitors, pre-, and post-conditions. Moreover, they define hotspots that simplify the implementation of handlers and the association of these handlers with components. These implementation infrastructures are complementary to our work.

2 Background

2.1 Fault Tolerance at the Architectural Level

The structure of a system is what enables it to generate its intended behaviour from the behaviour of its components. One of the benefits of a well-structured system is to avoid overly complex relationships between its components, which in turn should lead to a more dependable system [30]. One of the means to attain dependability is through fault tolerance, which aims to provide system services despite the presence of faults [4]. System structuring should ensure that the extra software required for tolerating faults should provide effective means for error confinement, does not add to the complexity of the system, and improves the overall system dependability [30].

During run-time, system failure is avoided via error detection and system recovery [4]. Error detection at the architectural level relies on monitoring mechanisms, or probes, for detecting erroneous states at the interfaces of architectural elements or in the interactions between these elements. On the other hand, the aim of system recovery is twofold. First, through error handling, to eliminate erroneous states from the system, and second, through fault handling, to prevent located faults from being activated again. The main activities associated with fault handling are: the identification of the type faults and their location, the isolation of faulty components to avoid faults to be reactivated, the reconfiguration of system components, and the resumption of system services.

Architectural abstractions offer a number of features that are suitable for the provision of fault tolerance [22], including error confinement, which is the ability of a system to avoid the propagation of errors. They also provide a global perspective of the system that enables high-level interpretation of system faults, thus facilitating their identification. The separation between computation and communication, which enforces modularisation and information hiding, facilitates error detection, confinement, and system recovery. The architectural configuration, being structural constraints, helps to identify anomalies in the system structure. Software architectures promote error confinement by fostering the creation of appropriate architectural structures, and providing the means for representation analysis, and error confinement mechanisms. Explicit system structuring facilitates the introduction of mechanisms such as program assertions, pre- and post-conditions, and invariants that enable the detection of potential erroneous architectural states. Architectural changes, for supporting fault handling during system recovery, can include the addition, removal, or replacement of components and connectors, modifications to the configuration or parameters of components and connectors, and alterations in the component/connector networks topology.

2.2 Exception Handling at the Architectural Level

Exception handling has shown to be an effective mechanism for incorporating fault tolerance into software systems [15]. Exception handling has been traditionally associated with late design and implementation phases of the software lifecycle, during which all the effort is made to protect the application software from faults that may be introduced during requirements analysis, design, and implementation, or faults that can occur at the support

level. The consequence of such an approach is that the appropriate context in which errors should be detected and recovered is lost. The potential correlation that might exist between the error states of the different contexts and how these should be recovered in an optimised way is also lost [21]. Hence, it is necessary for each identified phase of software development to define a class of exceptions depending on the abstraction level (or context) of the software system being modelled and analysed.

The incorporation of exception handling at the architectural level has been suggested as a valuable mechanism for error handling, if properly incorporated into the system structure [17]. It allows one to reason about the software fault tolerance properties at a higher level of abstraction by properly assigning abnormal behaviour responsibilities among the architectural components and connectors of a software system. Architectural-level and implementation-level exception handling are complementary techniques that should be employed synergistically in order to achieve best results.

2.3 Idealised Fault-Tolerant Component

The idealised fault-tolerant component is a structuring concept that makes it easy to identify *what* parts of a system have *what* responsibilities for trying to cope with *which* sorts of faults [2]. It makes it possible to structure systems in such a way that normal and abnormal behaviour can be kept separate: *normal*, correspond to those situations where service is provided as specified; and *abnormal*, correspond to those situations where errors are detected, and the component cannot provide the requested service. Abnormal responses are usually called exceptions [24].

Exceptions can be classified into two different categories: *internal*, raised by a component in order to invoke its own error recovery measures, and, if this exception is handled successfully, the component can return to provide its normal service; and *external*, signaled if a component determines that, for some reason, it cannot provide its specified service. External exceptions can be partitioned into *interface exceptions*, which are due to an invalid service request, and *failure exceptions*, which are due to a failure in the processing of a valid request. In this sense, exceptions and exception handling provide a suitable framework for structuring the fault tolerance activities incorporated in a system.

3 Architectural Fault Tolerance using Exception Handling

The architectural solution being advocated for the structuring of fault-tolerant systems is the *idealised fault-tolerant architectural element* (iFTE). The iFTE enforces the principles associated with the concept of the idealised fault-tolerant component, and includes the following mechanisms: (i) detection of errors in the architectural elements and their interactions; (ii) raising and handling of internal exceptions associated with the detected errors; (iii) handling of exceptions that were raised externally by other architectural elements; (iv) masking of internal or external exceptions by means of redundant resources; and (v) propagation of exceptions, internally or eternally raised, that cannot be masked.

Similarly to the idealised fault-tolerant component, the proposed approach advocates a complete separation between the normal and abnormal behaviour of architectural elements.

An advantage of the proposed architectural solution is that it can be instantiated into both idealised fault-tolerant architectural components and connectors, whose interactions are governed by exception communication rules:

- *idealised fault-tolerant architectural component (*iFTComponent*).* Responsible for preventing internal errors from propagating to the rest of the system through its interfaces by constraining its abnormal behaviour to a given specification;
- *idealised fault-tolerant architectural connector (*iFTConnector*).* Responsible for resolving potential mismatches that might exist among collaborating components, and preventing the propagation of errors by handling their exceptions.

The architectural solution is a specialisation of the peer-to-peer style [13]. In this architectural style any element can interact with other elements for providing services to them or requesting their services. Compared with the peer-to-peer style, the proposed architectural solution has the following three differences: (i) connectors are first class architectural elements that coordinate the interactions among components; (ii) interfaces of architectural elements have been expanded to include provided and required abnormal interfaces for enforcing the behaviour of the idealised fault-tolerant component concept [2]; and (iii) the communication between architectural elements is a request/reply interaction captured by the connectors and the links between architectural elements.

3.1 The Structure of the iFTE

The general model of an iFTE is shown in Figure 1. Its internal structure and interfaces are clearly partitioned into normal and abnormal (exceptional) behaviour. The iFTE has four types of external interfaces:

- I_ProvidedServices which is responsible for the provision of (fault-tolerant) services;
- I_SignalledExceptions which responsible for signalling either interface or failure exceptions;
- I_RequiredServices which specifies the required services; and
- I_ReceivedExceptions which specifies the external exceptions that need to be handled.

Internally, the iFTE is structured with four main architectural elements, as follows. The Normal component implements the normal behaviour of the iFTE. The Abnormal component handles the exceptions raised by the Provided or Required connectors, or the Normal component. The Provided connector detects abnormal conditions that may arise from the services provided by the Normal component, and raises exceptions when they occur. Finally, the Required connector detects abnormal conditions that may arise from the services required by the Normal component, and raises exceptions when they occur. The Normal component may raise internal exceptions.

The internal architectural elements of the iFTE interact through four internal interfaces. For being a call/return implementation of the iFTE, the propagation of exceptions follows

the requesting flow. Internal exceptions are propagated from Normal to Provided through interface I1. After that, the Provided requests a handling service of the Abnormal component through interface I3. If Abnormal is able to handle the exception, the return will follow to Provided, which returns normally through the interface I_ProvidedServices, masking the error. However, if Abnormal is not able to handle the exception, it throws an exception to Provided. If the Provided receives an abnormal response from the Abnormal component, it propagates the exception through the I_SignalledExceptions interface. If Normal requires external services, the request goes through I2, and it is re-directed by the Required connector to the I_RequiredServices interface. The service will return through the same path (normally or abnormally). When an external exception is signalled through I_ReceivedExceptions, it is intercepted by Required, and propagated to Normal through I2, and hereafter follows the same flow of an internal exception. If the Abnormal component requires internal services from Normal, the request goes through I4. If the Abnormal requires external services, the request goes through I2, as explained before.

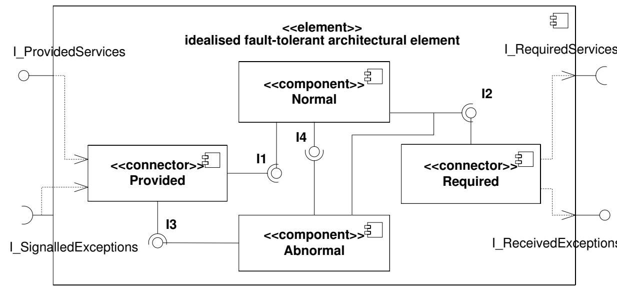

Figure 1: General model of the idealised fault-tolerant architectural element (iFTE).

From the perspective of an iFTE environment, the goal of partitioning its structure into normal and abnormal (exceptional) is to ensure that no other behaviour at the iFTE's interface is allowed except for the normal and exceptional behaviours that are specified in terms of what is provided and what is required. Also from the perspective of the environment, the iFTE is also responsible for handling exceptions that are propagated from other components or connectors with which collaborates, and propagating exceptions that it cannot handle locally.

It is worth noting that if some components and connectors are not based on the iFTE, it is necessary to provide additional mechanisms for dealing with errors related to these elements. In the presence of such non-fault-tolerant architectural elements, the separation between normal and abnormal behaviours should nevertheless be enforced at the level of the architectural configuration. This separation enhances the software understandability and maintainability, which also contributes positively to its dependability.

3.2 Exception Propagation

This section describes how exceptions are propagated among architectural elements. The presentation is based on the architectural configuration of Figure 2, which contains three

components and a connector. The connector (conn_X) manages the cooperation among the three components (comp_A, comp_B, and comp_C). The propagation of exceptions among components and connectors is very similar to that performed in the context of objects and cooperative actions (CO Actions) [18].

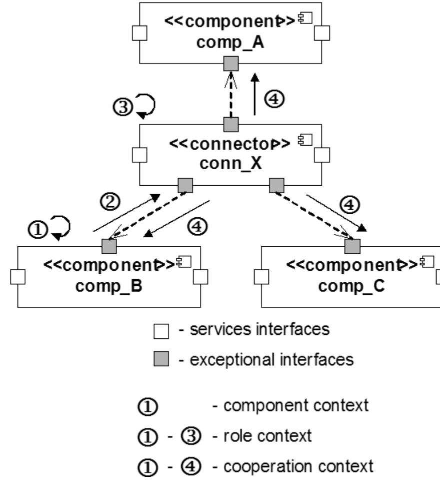

Figure 2: Propagation of exceptions.

In order to maintain the architectural integrity, internal exceptions are propagated through interface of the architectural element either as declared or undeclared architectural exceptions [10]. For the idealised fault-tolerant architectural connector, this may require to translate the type of an external exception being propagated. The objective of this translation is to deal with potential mismatches that might exist between collaborating iFTComponents. With such approach, internal exceptions that are raised inside components and connectors are encapsulated by their architectural interfaces.

3.2.1 From Components to Connectors

If an internal exception is raised by the normal part of the comp_B after a service has been requested (according to Figure 2, comp_B raises an internal exception $^{(1)}$), this exception should be handled by the component's abnormal part. For that, it should use local resources in a way that is transparent to the rest of the system. This scenario characterises the *component* context in the propagation of exceptions. If the abnormal part of a component is unable to handle the exception, the latter should be propagated to the connector, or connectors, managing the component's interactions (exception $^{(2)}$ is propagated to conn_X). This characterises the *role* context in the propagation of an exception because the connector might be able to handle the exception without involving other components.

From the perspective of a component's environment, the goal of partitioning its structure into normal and abnormal is to ensure that no other behaviour at the component's interface is allowed except for the normal and exceptional behaviours that are specified in terms of what is provided and what is required. Also from the perspective of the environment,

the component is also responsible for handling exceptions that are propagated from other components or connectors with which collaborates, and propagate exceptions that it cannot handle locally.

3.2.2 From Connectors to Other Elements

If an internal exception is raised when occur a violation on the conditions dictating the collaborative behaviour associated with a connector, it is raised an internal exception ((3)). In case an exception (either an internal exception or another received from a component) cannot be handled at the connector level, this should be propagated among the collaborating components for them to take the necessary corrective actions (exception (4) which is propagated by conn_X to comp_A, comp_B, and comp_C). This is the *cooperation context* scenario in which the connector propagates an exception to a group of collaborating components for them to handle the exception individually. Note that a collaborating component may be a connector that encapsulates another collaboration context recursively.

Although the collaborating components should not be aware of the additional behaviour introduced by a connector and its respective exceptions, the exceptions being propagated should be meaningful for the collaborating components. An example of such exception might be a failure exception notifying one of the collaborating components that the requested service cannot be provided. The same happens when the connector receives an exception from one of the collaborating components. Depending on its handlers, either the connector can treat this exception locally, or request the collaboration of other components for handling the exception. The recovery actions associated with the handling of an exception might involve replacing a collaborating component by its replica, or launching an alternative connector that might contain a different collaboration profile. The recovery activities performed by the handlers of a connector are not trivial since they might involve the restoration of state consistency across several architectural elements.

3.3 Possible Scenarios of a 'call/return' iFTE

Assuming a call/return implementation of the iFTE abstraction, presented in Section 3.1, we have enumerated all possible scenarios of its execution. These scenarios reffer to a situation where the iFTE receives requests of a client component (Client), and requests service of a server component (Server). Figure 3 presents this configuration.

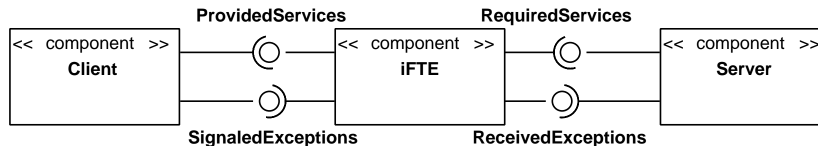

Figure 3: Configuration Adopted for these Scenarios

First, we present the scenarios in a "black-box" way, abstracting the internal structure of the iFTE (abstract scenarios). Second, we present the scenarios, considering the internal

behaviour of the iFTE (detailed scenarios). To simplify the representation of the detailed scenarios, we have classified them in three groups: (i) basic detailed scenarios, which abstract the representation of exception handling actions, showing only a single operation for this; (ii) handling detailed scenarios, which detail the exception handling scenarios and can be combined with both basic and wrapping detailed scenarios; and (iii) wrapping detailed scenarios, which represent the scenarios where the Normal component is reused and it is not able to detect exceptions from the normal returns of the Server component (the Required connector take this role).

Each possible scenario is presented bellow (Sections 3.3.1 to 3.3.4), classified according to its group.

3.3.1 Abstract Scenarios

Analysind the iFTE as a "black-box", there are a total of six scenarios for the configuration presented in Figura 3:

Abstract Scenario 1. Figure 4 represents a simple and successful execution scenario, where the iFTE did not need any external service and has returned normally.

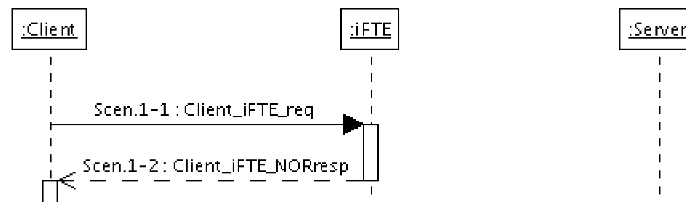

Figure 4: A simple and successful execution

Abstract Scenario 2. Figure 5 represents a simple and unsuccessful execution scenario, where the iFTE did not need any external service, but has returned abnormally.

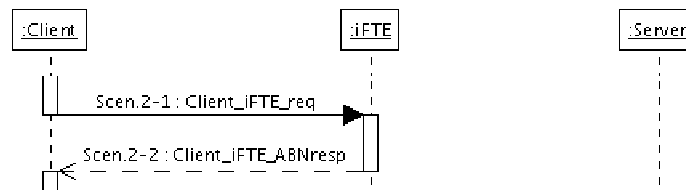

Figure 5: A simple and unsuccessful execution

Abstract Scenario 3. Figure 6 represents a successful execution scenario, where the iFTE requested some external service, received a normal response and has returned normally.

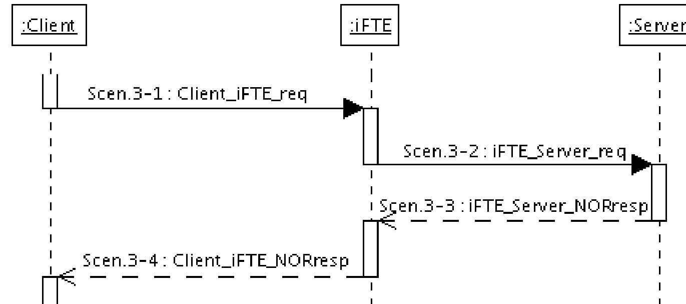

Figure 6: A successful execution without error masking

Abstract Scenario 4. Figure 7 represents an unsuccessful execution scenario, where the iFTE requested some external service, received a normal response but has returned abnormally for some internal reason.

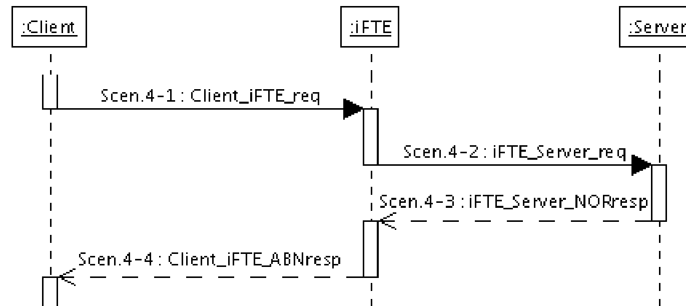

Figure 7: An unsuccessful execution because an internal error

Abstract Scenario 5. Figure 8 represents a successful execution scenario, where the iFTE requested some external service, and although it has received an abnormal response, it returned normally.

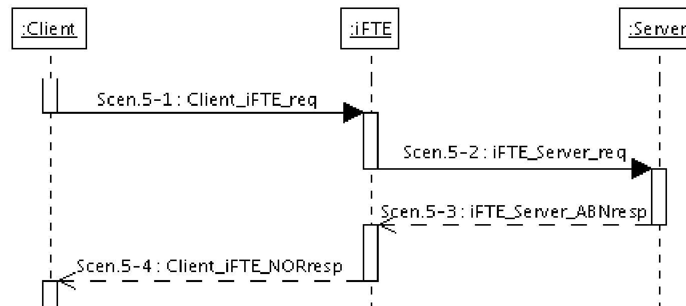

Figure 8: A successful execution with error masking

Abstract Scenario 6. Figure 9 represents an unsuccessful execution scenario, where the iFTE requested some external service, received an abnormal response and has returned abnormally because could not mask the error.

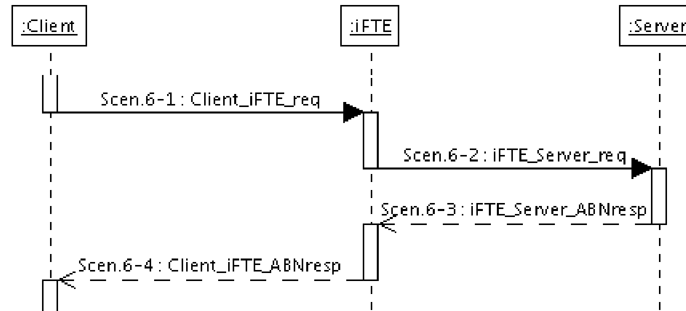

Figure 9: An unsuccessful execution because an external error

3.3.2 Basic Detailed Scenarios

Analysind the iFTE with its four internal elements, there are a total of nine basic scenarios for the configuration presented in Figura 3. This scenarios do not detail all ways an excepcion an be handled by the Abnormal component inside the iFTE. The nine basic scenarios are following presented:

Basic Detailed Scenario 1. Figure 10 represents a simple and successful execution scenario, where the iFTE did not need any external service and has returned normally. It corresponds to the abstract scenario 1.

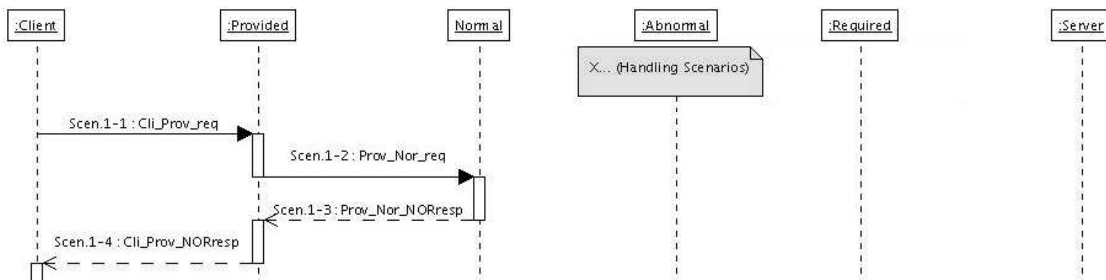

Figure 10: A simple and successful execution

Basic Detailed Scenario 2. Figure 11 represents a successful execution scenario, where the iFTE did not need any external service, recovered from an internal error, and has returned normally. It also corresponds to the abstract scenario 1.

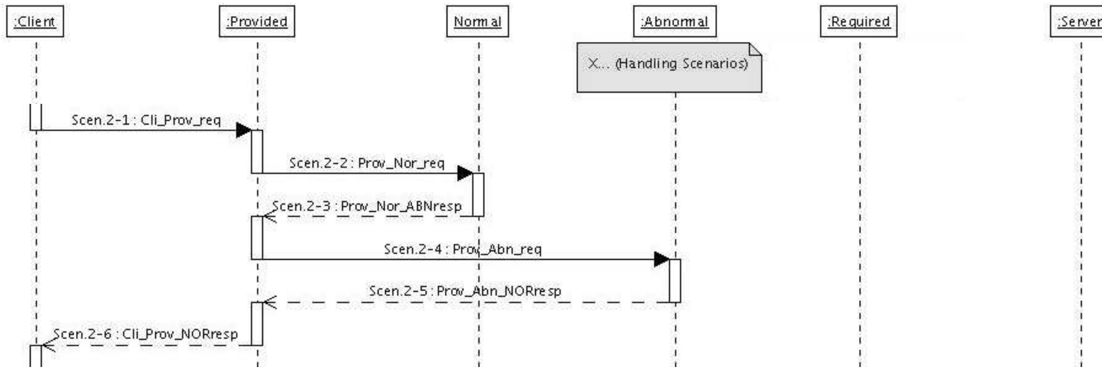

Figure 11: An simple error masking (successful) execution

Basic Detailed Scenario 3. Figure 12 represents an unsuccessful execution scenario, where the iFTE did not need any external service, raises an internal exception, and has not been capable to recover from it, returning abnormally. This scenario corresponds to the abstract scenario 2.

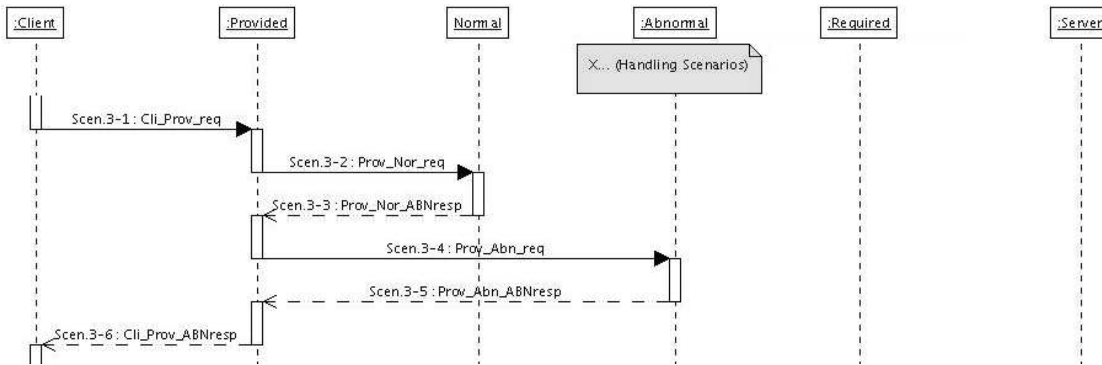

Figure 12: An simple unsuccessful execution

Basic Detailed Scenario 4. Figure 13 represents a successful execution scenario, where the iFTE needed some external service and has returned normally. It corresponds to the abstract scenario 3.

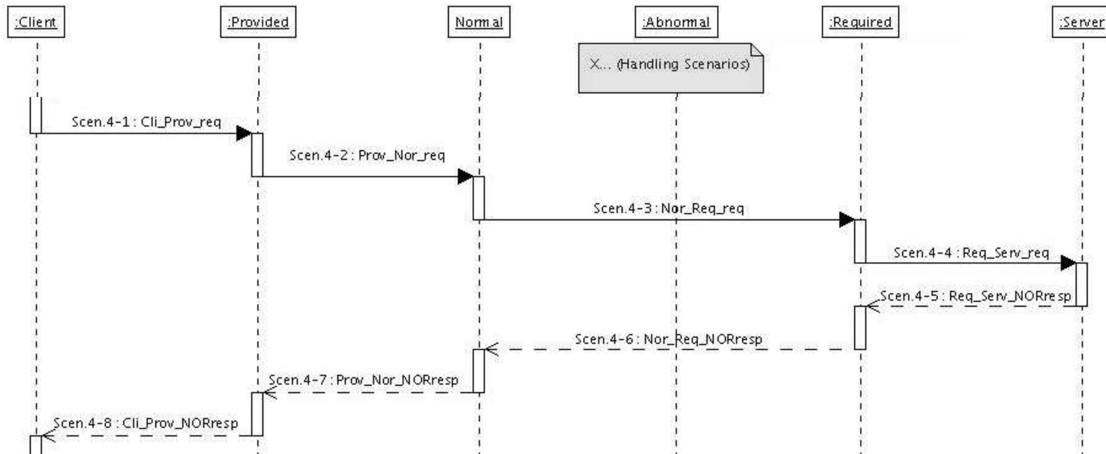

Figure 13: A simple and successful execution

Basic Detailed Scenario 5. Figure 14 represents a successful execution scenario, where the iFTE needed some external service, received a normal return from the server, but raised some exception that chould be successful handled (error masking). The iFTE has returned normally and this scenario also corresponds to the abstract scenario 3.

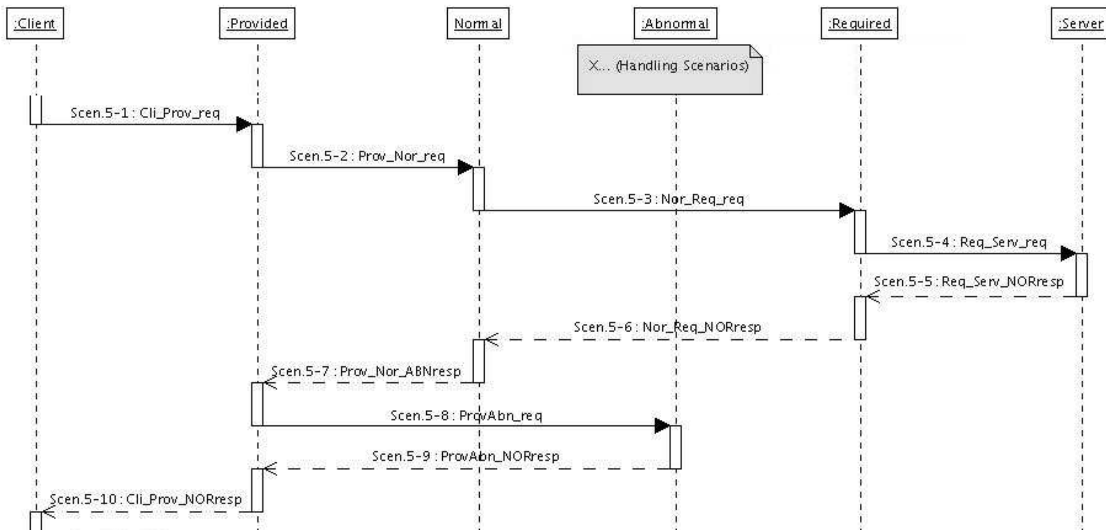

Figure 14: An error masking (successful) execution

Basic Detailed Scenario 6. Figure 15 represents an unsuccessful execution scenario, where the iFTE needed some external service, and although it has received a normal return from the server, it raised some exception that could not be masked. The iFTE has returned abnormally and this scenario corresponds to the abstract scenario 4.

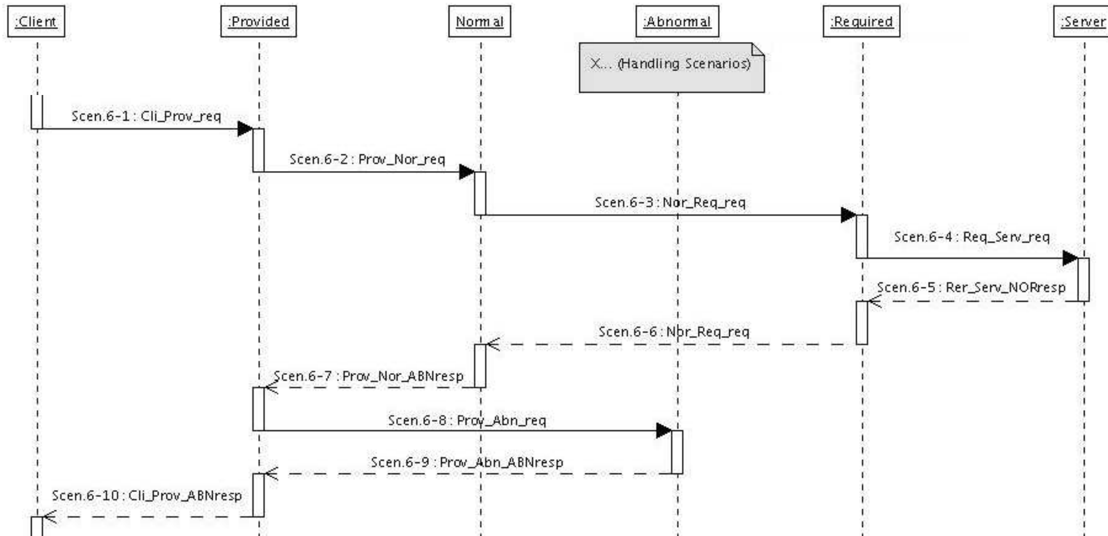

Figure 15: An internal exception after a normal return (unsuccessful)

Basic Detailed Scenario 7. Figure 16 represents a successful execution scenario, where the iFTE needed some external service, received an abnormal return from the server, but recovered from this error (error masking), and has returned normally. It corresponds to the abstract scenario 5.

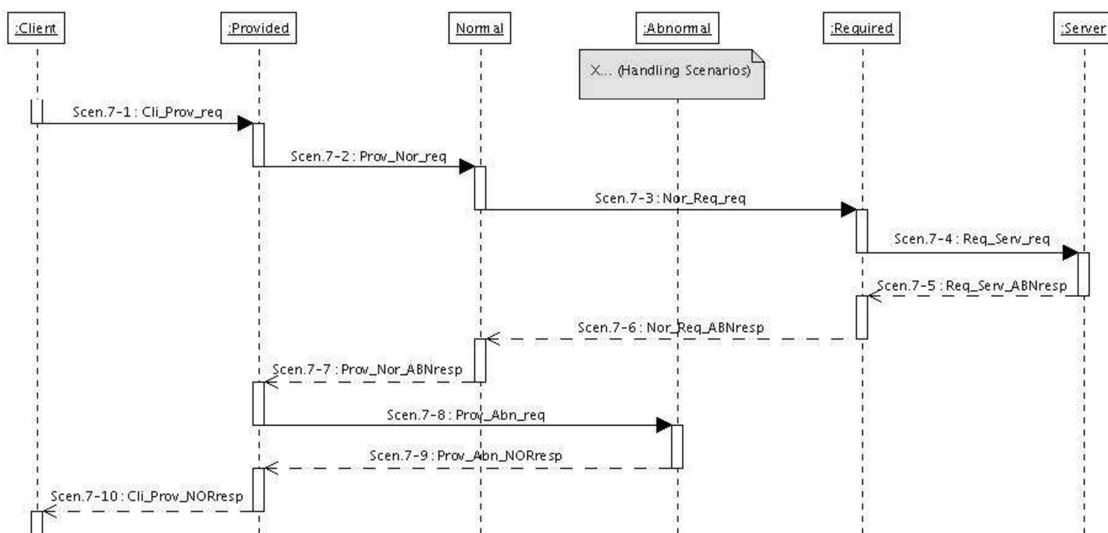

Figure 16: An external error masking (successful) execution

Basic Detailed Scenario 8. Figure 17 represents an unsuccessful execution scenario, where the iFTE needed some external service, received an abnormal return from the server, and could not recover from this error, returning abnormally. It corresponds to the abstract scenario 6.

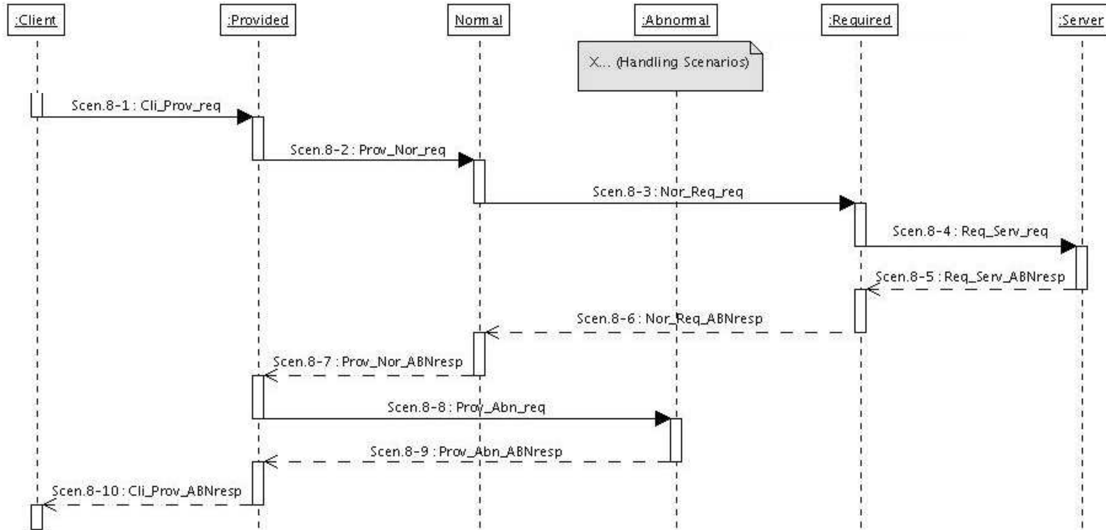

Figure 17: An external exception (unsuccessful) execution

Basic Detailed Scenario 9. Figure 18 represents a simple and unsuccessful execution scenario, where the iFTE has received an invalid service request from its client, and immediately returned an interface exception. Note that the Provided connector is responsible for detect this kind of errors and do not propagate the request to the Normal component. It also corresponds to the abstract scenario 6.

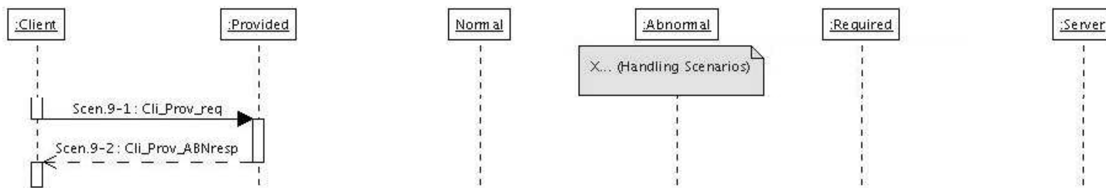

Figure 18: An external error masking (successful) execution

3.3.3 Handling Detailed Scenarios

The Abnormal component may handle exceptions in a variety of ways. It is possible, for example, to use external services (through the Required connector), or some internal service (of the Normal component). There are a total of nine scenarios, which are following presented. As can be seen in the following figures (through a UML comment), we have simplified the requestation of internal services by a method call from the Abnormal component to the Normal one. In fact, we would have to represent all combination of service execution existing in the basic detailed scenarios.

Handling Detailed Scenario 1. Figure 19 represents a simple and successful handling scenario, where the iFTE did not need any external nor internal service and has masked the error.

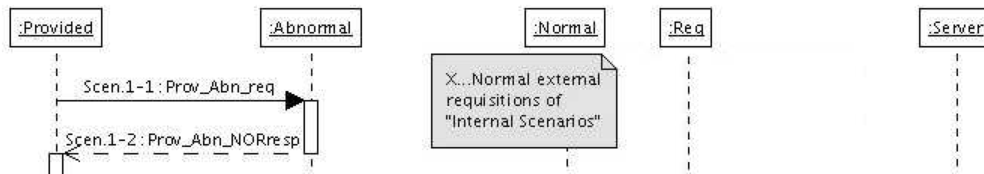

Figure 19: A simple and successful error masking

Handling Detailed Scenario 2. Figure 20 represents a simple and unsuccessful handling scenario, where the iFTE could not mask the exception and has not executed any external nor internal service.

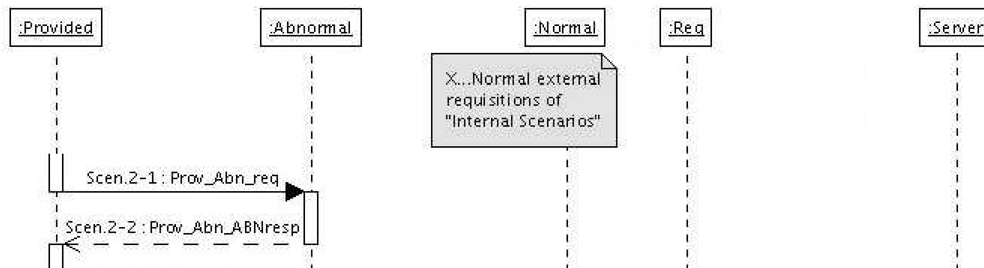

Figure 20: A simple and unsuccessful error masking

Handling Detailed Scenario 3. Figure 21 represents a successful handling scenario, where the iFTE uses an internal service (from the Normal component) for masking the error.

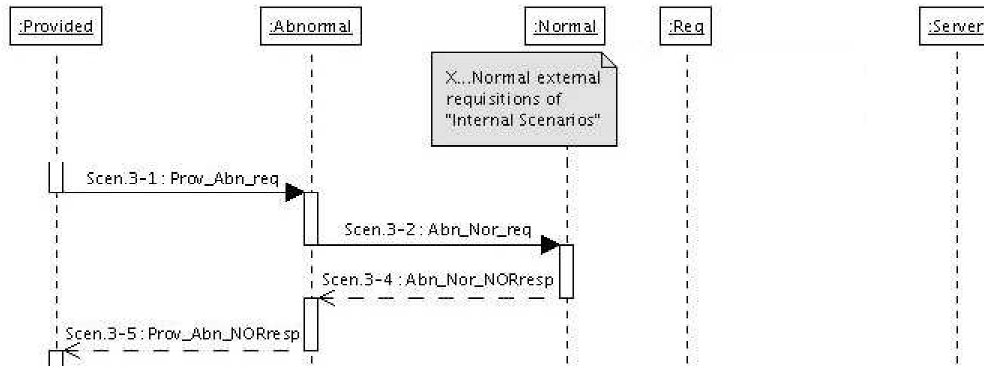

Figure 21: A successful error masking executingan internal service

Handling Detailed Scenario 4. Figure 22 represents an unsuccessful handling scenario, where the iFTE could not mask the exception, but it has executed some palliative internal service (from the Normal component); for example, a local logging service.

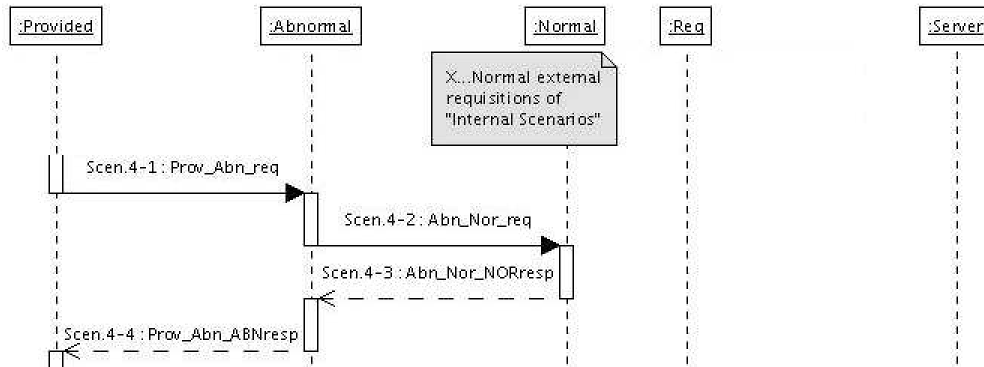

Figure 22: Successful error handling, but unsuccessful masking

Handling Detailed Scenario 5. Figure 23 represents an unsuccessful handling scenario, where the iFTE could not mask the exception. The Abnormal component also tried to execute an internal service of the Normal component (possibly a retry), but did not obtained a normal return.

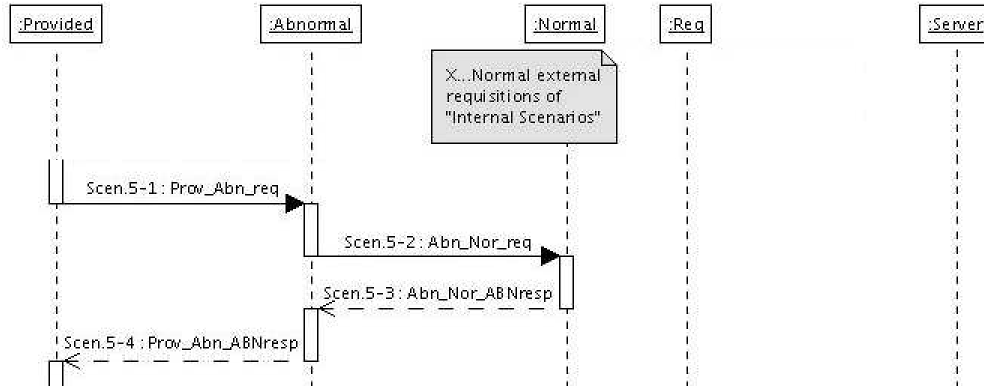

Figure 23: Unsuccessful error handling and masking

Handling Detailed Scenario 6. Figure 24 represents a successful handling scenario, where the iFTE uses an external service (through the Required connector) for masking the error.

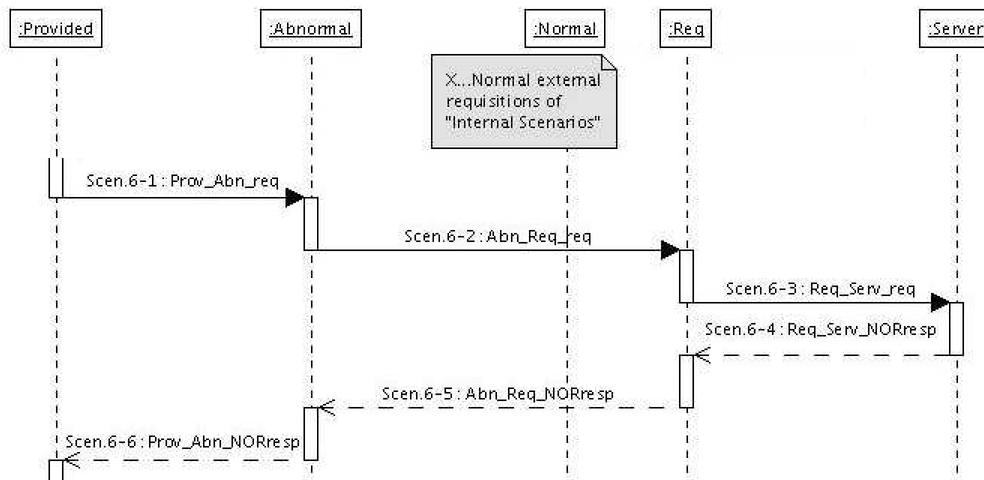

Figure 24: A successful error masking executingan external service

Handling Detailed Scenario 7. Figure 25 represents an unsuccessful handling scenario, where the iFTE could not mask the exception, but it has executed some palliative external service (through the Required connector); for example, an external logging service.

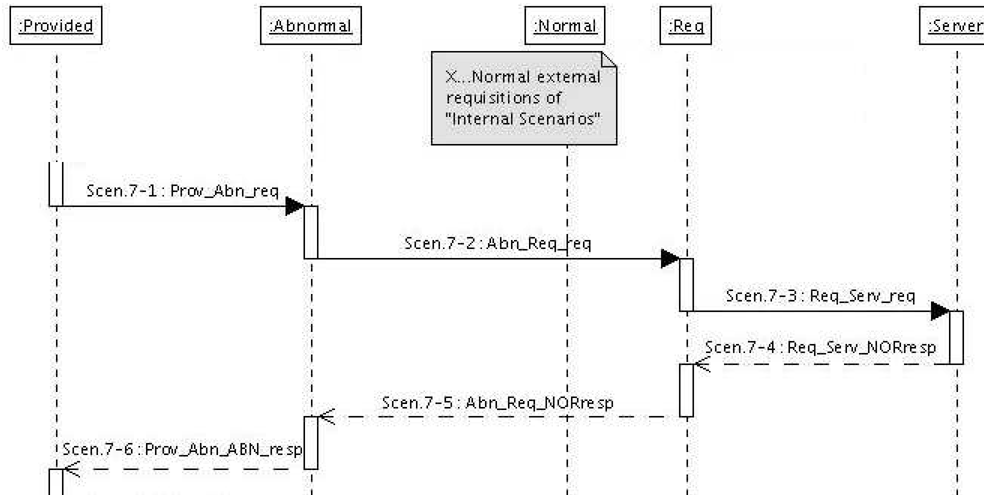

Figure 25: Successful error handling, but unsuccessful masking

Handling Detailed Scenario 8. Figure 26 represents an unsuccessful handling scenario, where the iFTE could not mask the exception. The Abnormal component also tried to execute an external service through the Required connector (possibly a workaround), but did not obtained a normal return.

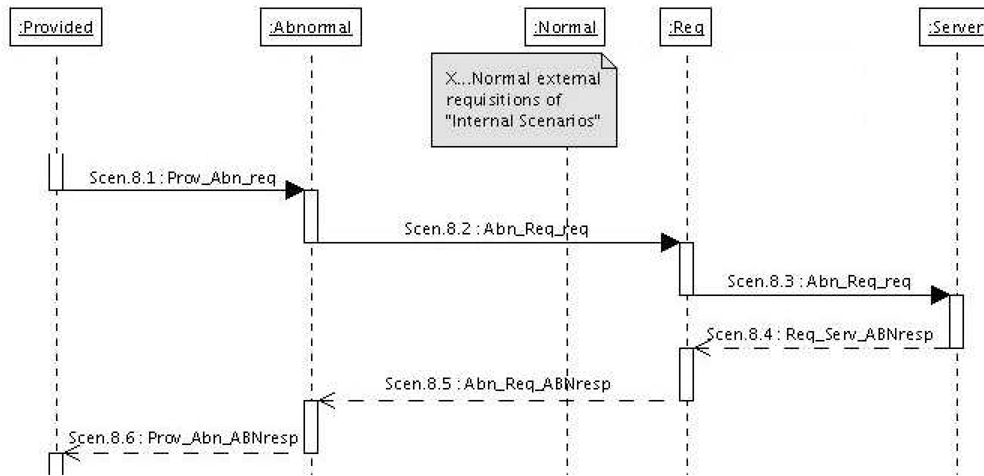

Figure 26: Unsuccessful error handling and masking

Handling Detailed Scenario 9. Figure 27 represents a generic scenario, where the Abnormal component executes both internal and external services. This scenario is generic meaning that it complies with all the eight following possibilities: (i) exception masking with both internal and external services returning normally; (ii) exception masking with a normal return for the internal service and an abnormal one for the external service; (iii) exception masking with an abnormal return for the internal service and a normal one for the external service; (iv) exception masking with both internal and external services returning abnormally; and the same four possibilities where the iFTE could not mask the error (v, vi, vii, and viii).

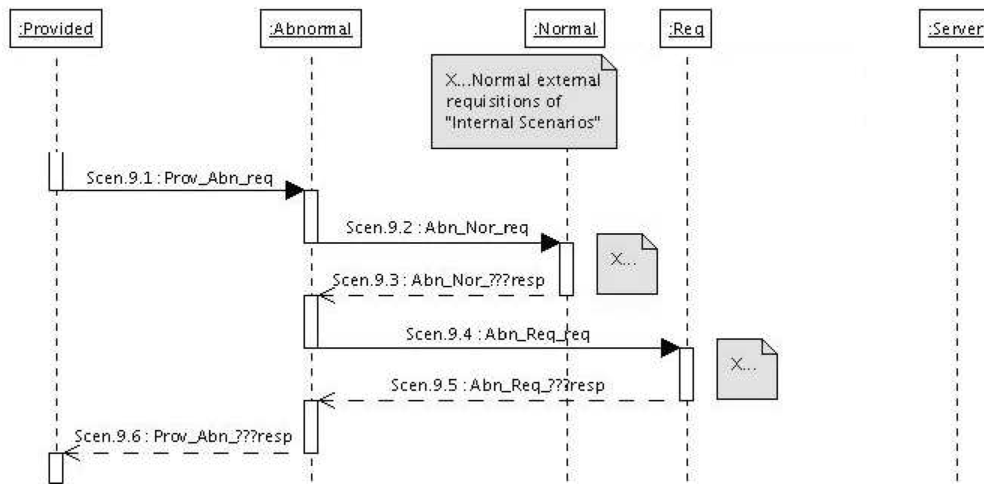

Figure 27: Generic handling scenario using both internal and external services

3.3.4 Wrapping Detailed Scenarios

Beyond the basic scenarios presented in Section 3.3.2, the iFTE can also be used as a wrapper for reused components. In this context, the reused component assumes the role of the Normal component, the Abnormal component has to be implemented because the exception handling is application specific, and the Provided and Required connector are responsible for adjusting the Normal component to be compatible with the rest of the system. Because a reused component could not identify all post-conditions expected for the external services, in these cases, the Required connector has to verify all return values from the Server component and judge if it is an accebtable value (normal return), or a violation of the post-condition (an exception). In the first case, the internal connector has to convert the result to a comprehensible data structure of the reused component, and in the second case it is necessary to Raise an exception that can be propagated until the Provided connector, which require a handling service for the Abnormal component (see the handling scenarios in Section 3.3.3). There are only two scenarios of exception detection for the Required connector; these scenarios are following presented:

Wrapping Detailed Scenario 1. Figure 28 represents a simple and successful detection scenario, where the Required connector detects an exception of a normal return, raises an exception, but the iFTE has masked the error.

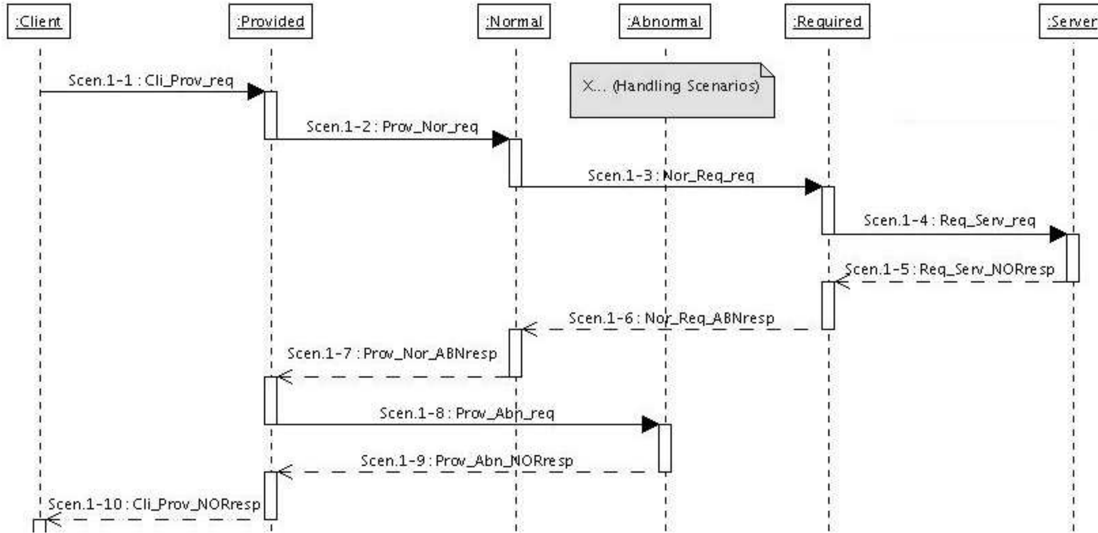

Figure 28: A simple and successful error masking

Wrapping Detailed Scenario 2. Figure 29 represents a scenario similar with the first one. The only difference is because in this case, the exception cluld not be masked successfuly.

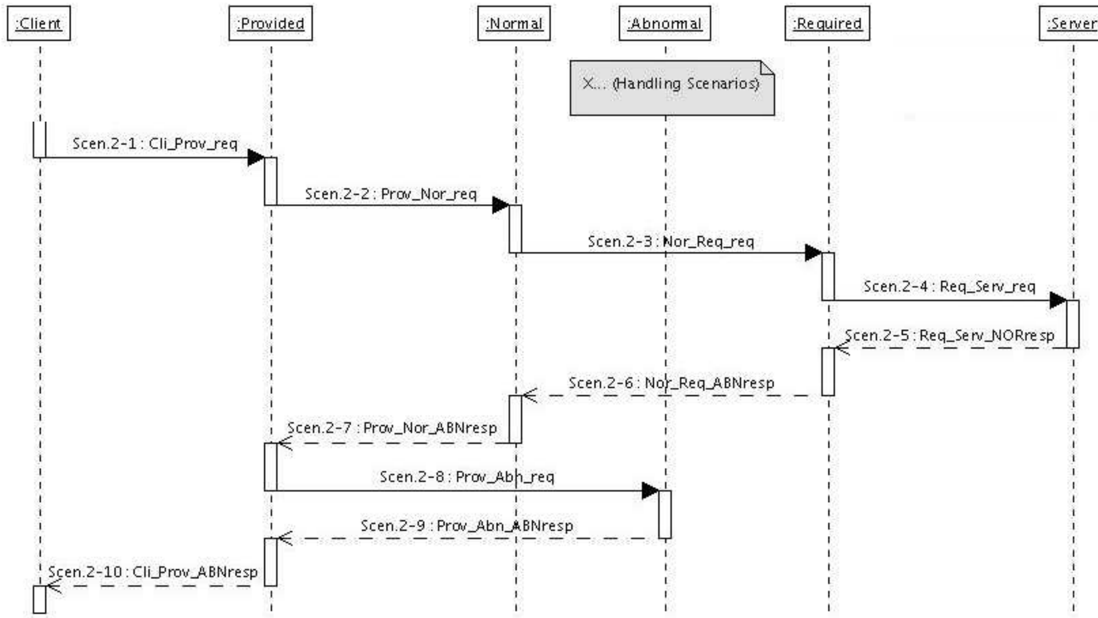

Figure 29: A simple and successful error masking

4 A Rigorous Development Approach

System structuring should ensure that the extra software required for tolerating faults should provide effective means for error confinement, does not add to the complexity of the system, and improves the overall system dependability [30]. During run-time, system failure can be avoided via error detection and system recovery [4]. The aim of system recovery is twofold. First, through error handling, to eliminate erroneous states from the system, and second, through fault handling, to prevent located faults from being activated again.

Architectural abstractions offer a number of features that are suitable for the provision of fault tolerance, including error confinement, which is the ability of a system to avoid the propagation of errors. Besides that, the separation between computation and communication, which enforces modularisation and information hiding, facilitates error detection, confinement, and system recovery. The architectural configuration, being structural constraints, helps to identify anomalies in the system structure. Explicit system structuring facilitates the introduction of mechanisms such as program assertions, pre- and post-conditions, and invariants that enable the detection of potential erroneous states at the architectural level.

Besides these guidelines, a claim recently being made is that, to control the complexity of modern software systems, the software development has to focus on the software architecture, which has to guide all development process of complex systems. Besides that, to achieve the desired levels of dependability, mechanisms for detecting and handling errors should be systematically incorporated from early phases of software development, since they directly influence the architectural design phase [21, 32, 16, 33].

Figure 30 presents a simplified view of a rigorous architecture-centric development process for constructing dependable systems. Based on this process, we are proposing a complete infrastructure for constructing software systems with strict dependability requirements.

First, during the requirements engineering and analysis, the requirements are documented through high-level scenarios. After the requirements engineering it is specified the software architecture of the system in a high level of abstraction. After this, the formal models are verified and are generated all possible scenarios through the formal models. These scenarios are used to validate the correctness of the model, according to the requirements that where specified. After verify and validate this structural view of the system, it is possible to analyse dependencies amongst architectural elements, what is important to support the development and dictates the best order for integrating them. After that the architect may detail the specification of the software architecture or directly generate the UML diagrams from the formal specification that has been done before. Following, the UML models can be used as input to an automatic process for generating test cases and part of the source code. Finally, the software is manually refined and passes by a refinement cycle, when the correctness of the generated source code is validated through the execution of the test cases.

At the actual state of the work, we have only automated the dashed activities of the diagram (Figure 30). The other possible automation activities are part of some ongoing and future work, as presented in the conclusions (Section 8).

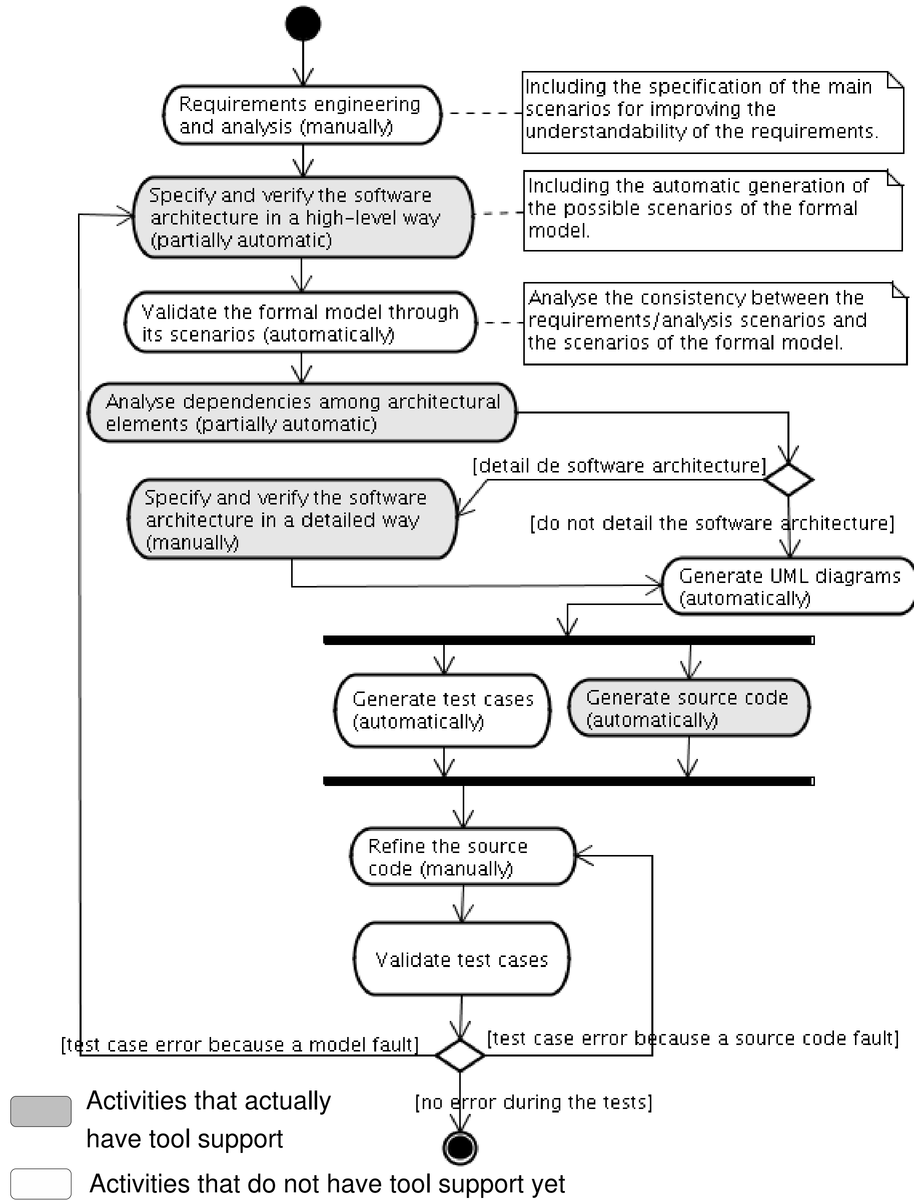

Figure 30: A Rigorous Architecture-Centric Development Process

5 The Scenario Identifier Tool

This section presents a tool for identifying scenarios from B-Method and CSP Formal Models. As presented in the rigorous development process of Section 4, these scenarios are useful both for verifying some properties of the formal models (Section 6), and for validating the formal model according to the scenarios specified during requirements engineering and analysis phase.

Following, Section 5.1 presents a short description of the tool with a screenshot of its user interface. After that, Section 5.2 presents the architecture and the modelling of the tool.

5.1 Tool Description

From two specification files (a CSP process and a B-Method machine), the scenario identifier tool automatically identifies all possible scenarios that can be executed and reports this information to a further analysis. It is important to say that the content of the input files follow the same notation of the ProB model checker tool [29]. It was an important requirement, thus we have intended to use both tools in a complementary way.

Although it is possible to analyse the output scenarios in a semi-automatic way (using a text editor, for example), the ideal solution is to provide a complete automatic analysis, including a way for formally specifying properties related with scenarios. The construction of this complementary module of the tool is one of our ongoing work, as can be seen in the conclusions of this report (Section 8).

To better ilustrate the tool, Figure 31 presents a screenshot of its user interface. As can be seen on its top, the user has to choose the two specification files (CSP and B-Method), and click on the `Run` button. The result is showed in a scrolled text output, and can be saved for a further analysis.

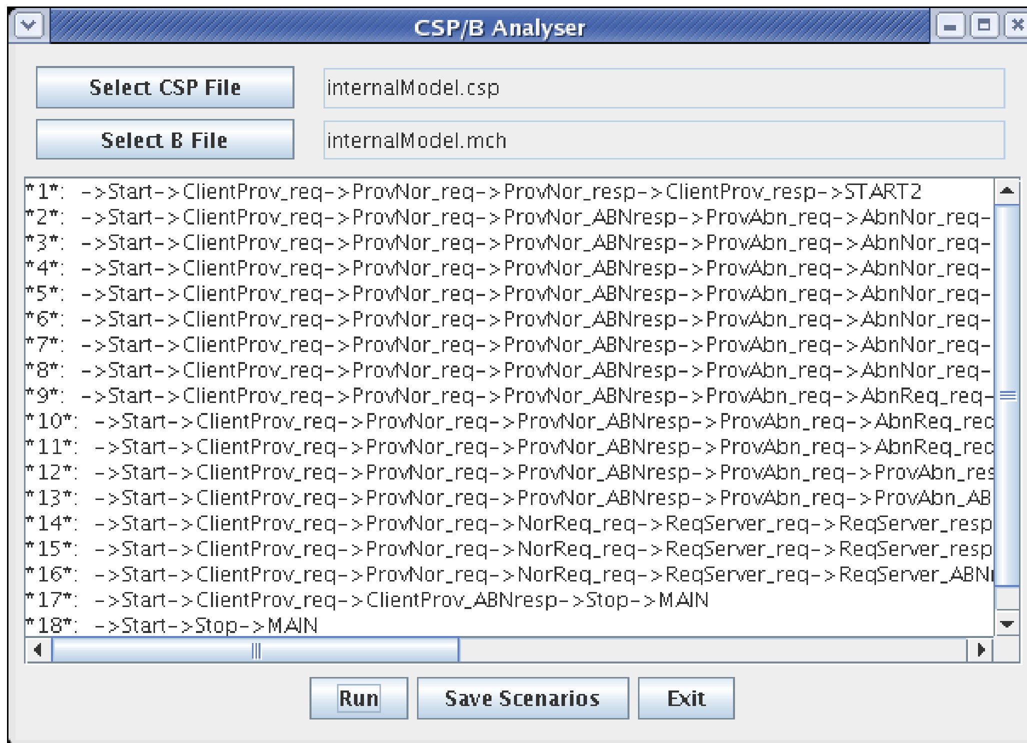

Figure 31: User interface of the scenario tool.

Besides identify the scenarios, as can be seen ahead in Section 5.2, the software architecture designed for the system reduces the effort needed for adding new feature modules.

5.2 Tool Modelling

Figure 32 presents a logical view [5, 28] of the tool software architecture, using a UML component diagram. The three layers architectural style [5] has been adopted. Beyond the three layers of the architecture, this figure also shows the main modules that compose the scenario identifier tool, as well as the relationship amongst them.

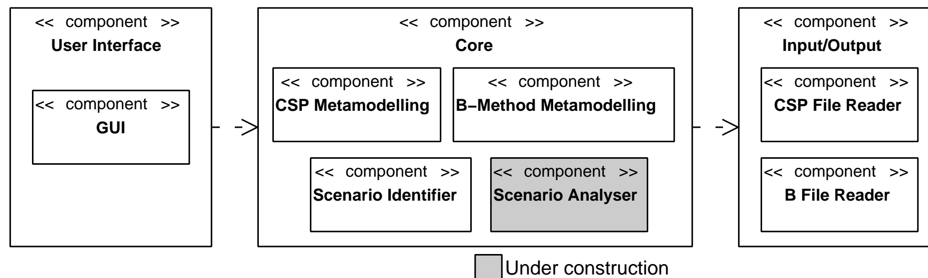

Figure 32: Logical view of the software architecture.

This system was developed using the object-oriented abstractions. For exemplifying the detailed design of the modules, Figures 33 and 34 present, respectively, the metamodel of the CSP specification and the metamodel of the B-Method machine, which are used by the other "core" modules. These metamodels are represented using UML class diagrams.

A `CSP Specification` (see Figure 33) is composed of `Process`es, which is a kind of `ComposedEvent` that uses the composite design pattern [23] to have recursive composition of `Event`s. An `Event` is the basic element of a scenario, and represents a service call or return. A `ComposedEvent` is a set of events with a kind of execution (sequential, internal choice, or external choice) [8].

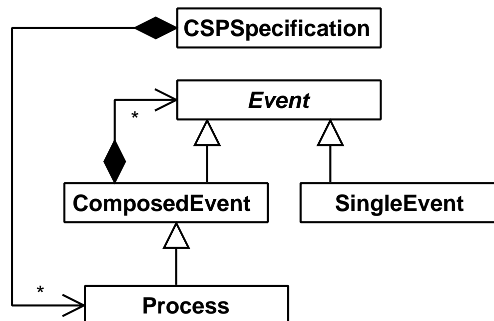

Figure 33: Metamodel of the CSP specifications.

A `BMachine` (see Figure 34) is composed of `Sets`, `Variables`, and `Operations`. A `Set` contains some elements, which identify its `Type`. A `Variable` is a structure that stores one element of a specific `Type`, because this, it is common to say that a `Variable` has a `Type`. Beyond a single set, the B-Method notation permits the representation of other `Types` of `Variables`: (i) `SingleVariable`, which stores a single element of a `Set`; (ii) `SequenceVariable`, which stores many elements of a set, ordering them chronologically; (iii) `RelationVariable`, which represents a relation among two `Types`, realising a "mapping" among elements of them; (iv) `PowerSetVariable`, which stores a subset of a `Set`; and (v) `PairVariable`, whose elements composed by other types (a mathematical ordered pair). Except by the first one (`SingleVariable`), the other variable types might be used as a `Type`. Finally, an `Operation` has `Arguments`, might return a value of an specific `Type`, might have `Preconditions`, and has a `Body` with source-code that can change the values of the machine variables.

Figure 34: Metamodel of the B-Method machines.

6 Formal Verification and Exception Propagation

In a previous work [19], the correctness of the internal operation of the iFTE has been partially verified using the UPPAAL model checker [6], which is based on extended timed automata. Although UPPAAL has a graphical notation, it has not the sufficient support for instantiating the iFTE abstraction for different components. For this, it is necessary to represent different types of exceptions to a further useful analysis of the exceptions propagation. In this technical report, beyond verify the internal flow of the iFTE, we

have also analysed the propagation of exceptions in a fault-tolerant software architecture. In order to model and verify the fault-tolerant properties of a software architecture, we have used ProB model checker [29], which supports the use of the B-Method [1], based on set theory, and CSP [8], which is a process algebra. The combined use of these methods provides an adequate notation for creating stateful models (from B-Method) with complex behavioural restrictions (from CSP). The joining point in ProB between the B-Method and CSP are the B-Method operations. In ProB, B-Method machine operation corresponds to an event in CSP [9].

To verify the dependability properties of the iFTE, presented in Section 3, we have done four complementary models: (i) iFTE abstraction model, which represents the observable behaviour of the iFTE abstraction; (ii) iFTE internal structure model, which represents how the iFTE works internally; (iii) high-level architecture model, which represents the information flow of the software architecture, abstracting the internal structure of the iFTE elements; and (iv) architecture detailed model, which represents the information flow of the software architecture, including the internal structure of each iFTE architectural element.

The separation among these four models of representation is to promote a better separation of concern among the interests of different stakeholders in different development phases. For example, the abstract representation of the iFTE could be useful for understanding the iFTE itself, while the architect could use this abstraction to represent the software architecture in a high-level way (its logical view). But the same architect could also need to detail all the iFTE architectural elements internally (the detailed view) when thinking about reuse of the Normal components of each architectural element. Finally, the internal structure of the iFTE element is useful in a software factory perspective, when the developers know the components which have been implemented, but are unaware of their contextualisation in the software architecture. These models were specified using the ProB, combining the use of B-Method machines with CSP process specifications.

An advantage of using ProB is that sets can be used for representing hierarchies of exceptions' types, which are important from the point of view of exception propagation. In UPPAAL this is not supported without increasing unnecessarily the complexity of the model. For expressing process coordination, UPPAAL's automata is more appropriate than the B-Method, however in ProB the integration of CSP into the B-Method has dealt with this limitation.

In the following four sections, we first describe the specification and verification of the iFTE abstraction, then we discuss the verification of its internal structure and the verification of software architectures based on iFTE (both high-level and detailed models). All ProB models described in this work are available in full in Appendices A, B, C, and D. Finally, Section 6.5 shows how these models are instantiated to a specific application. This instantiation consists on adjust the iFTE interfaces according to specific application components.

6.1 Formal Verification of the iFTE Abstraction

This section shows how the behaviour of the iFTE abstraction (Section 3) was formally represented and verified. Firs we consider the iFTE in a black box way, analysing its

externally observable behaviour without concern its internal structure: Provided, Normal, Abnormal, and Required. That is, this section presents the iFTE through all its possible scenarios in a software architecture. After that, are presented some properties which verifies violation of the predictable behaviour of an iFTE.

6.1.1 Formal Representation of the iFTE Abstraction

An iFTE was represented in B-Method as a machine, whose operations represents all possible events which can happen between it and the client, or it and the server. Client in this context is the role of every component which requests service to the iFTE. Analogously, server is the role of every component which the iFTE depends on. These events which are represented in the B-Machine basically represents service calls and the respective returns (normal and abnormal). The B-Method machine does not define any restrictions about the valid sequence of events (B-Method operations). The valid communication scenarios among the iFTE, the client, and the server are defined through a complementary CSP specification.

The internal components of the iFTE (Provided, Required, Normal, and Abnormal) are not represented in this model. An detailed view of the iFTE's internal functioning is presented in Section 6.2.

Besides operations represent requests and returns of services, their name also identify the source and the destination of the requesting. For example, `ClientIFTE_req` operation represents a service request that a client component do for the iFTE. The normal and abnormal returns are represented respectively by `ClientIFTE_resp` and `ClientIFTE_ABNresp` operations. Reasoning in this way, an association between the iFTE and one client or one server component is determined through the operations which are allowed or forbidden. For example, the iFTE can just receive requests from its clients, and can just request service to its servers. This restrictions are defined in CSP.

Besides representing all possible events of an iFTE interaction, the B-Method machine defines some auxiliary variables for storing information that are useful during the verification process. For example, to provide traceability, we initialy specified a B-Method variable (`sequenceHistory`) to store the sequence of operations (requests and returns) that have been executed in the B-Method machine. Although in the point of view of the correctness it was a good solution, the explosion in the number of states drastically reduced the scalability of this approach. As we shall explain in Section 6.1.2, we have developed a complementary tool to analyse the execution scenarios. The description and specification of this tool was presented in Section 5.

After modelling all external events of an iFTE, the next step is to represent the interaction rules among the iFTE, its clients, and its servers. These behavioural restrictions were defined in CSP. Basically, the CSP specification must explicitly represent the valid interaction scenarios among the components which participate in the interaction. These scenarios represent the rules of service calls and how exceptions are raised or propagated by the iFTE.

More details about the representation in B-Method and CSP are presented in Section 6.1.3.

6.1.2 Verification of Properties of Interest

The behavioural consistency of the iFTE has to be verified in three complementary ways: (i) verify the existence of a valid subset of its predicted scenarios (see Section 3.3); (ii) verify the absence of impredicted scenarios (see Section 3.3); and (iii) verify the consistency of the excepcion handling policy. Due to their strict relation with the component behaviour, most off verification uses the scenario identifier tool, which has been presented in Section 5. In order to verify the exception handling consistency of an iFTE, we have specified six properties, which are presented in Table 1.

In a general way, properties of Table 1 compare the reaction of an iFTE with its expected behaviour after receiving a response (normal or abnormal) from its server components. The second column of this table shows the objective of verification, while its third and fourth columns show how the property was specified to achieve the objective and what tool had been used to verify them. So as can be seen, these properties intend to verify basic aspects of the iFTE, for example the absence of deadlock (Property 1), and the consistency about its behaviour in the presence of exceptions (Properties 2 to 5). Properties 2 to 4 state that when the iFTE receives an abnormal return from a server component, it has to consider the exception type, in order to know if it can be masked and propagated. In this way, Property 5 prevents an exception is "ignored" in the software architecture, what certainly reduces the system reliability. Property 6 has a complementary role, preventing that the iFTE propagates an exception which was not defined as external for that element. The events presented in Table 1 are expressed in a "CSP-like" notation, where `‘‘?’’` separates the name of the event (before it) and the value returned by it (when there is one).

Table 1: Handling Properties of an iFTE.

#	Objective of verification	Specification	Used Tool
1	Verify if the iFTE abstraction is correct (absence of deadlock).	Automatically verified by ProB model checker.	ProB
2	Verify if every exception that the Server component signals and the iFTE cannot mask is not ignored. It has to be converted or explicit propagated.	After the `‘‘IFTEServer_ABNresp?E1’’` event, if the iFTE could not mask the exception `‘‘E1’’`, the following event has to be either `‘‘ClientIFTE_ABNresp?E2’’`, or `‘‘IFTEServer_req’’`, but the `‘‘ClientIFTE_ABNresp?E2’’` event has to occur before the end of the scenario.	scenarios
3	Verify if the iFTE does not mask exceptions that cannot be masked by it.	After the `‘‘IFTEServer_ABNresp?E’’` event, if the iFTE could not mask the exception `‘‘E’’`, the `‘‘ClientIFTE_resp’’` event would not be possible of occuring in the same scenario.	scenarios
4	Verify if every exception which the Server component signals and the iFTE can mask, can be successful handled, tolerating the fault.	After the `‘‘IFTEServer_ABNresp?E’’` event, if the iFTE could mask the exception `‘‘E’’`, the `‘‘ClientIFTE_resp’’` event would be possible of occuring in the same scenario.	scenarios
5	Verify if the iFTE handles all exceptions signaled by the Server component.	(Property#2 AND Property#4).	scenarios
6	Verify if the iFTE signals or propagates only external exceptions.	When the `‘‘ClientIFTE_ABNresp?E’’` event occurs, the `‘‘E’’` exception has to had been declared as external.	scenarios

Properties which are related only with the consistency of the state of the machine were specified in B-Method notation and had been analysed through the temporal model checking feature of ProB. Beyond this analysis, some properties are related with the consistency of execution scenarios. For these, we have initially used B-Method specification in ProB, as presented in Section 6.1. But, the maintenance of the sequenceHistory caused a high explosion of the number of states. Because this, all properties related with the analysis of the execution scenarios are verified through the scenarios identified by a different tool developed specifically for this reason. It is a simple tool, which from the CSP process algebra, discover all possible combination of execution scenarios. After that, it reads the B-Method specification and combines all possible values of parameters, according to the pre-conditions of each operation. The notation of the CSP and B-Method files are the same of the ProB. The output of this tool is a set of scenarios; based on the sequence of events of these scenarios it is possible to verify inconsistencies (counter-examples) that possibly may occur. The main inconsistencies were specified as properties. Using this complementary tool, the number of possible states discovered by ProB decreased from more than 15,000[1] to 20 states for the abstract model of the iFTE. This reduction is because the combination of the 20 possible states with each variation of scenario (that was specified through a B-Method variable).

As an example of a specification of a property, Property #2 of Table 1 consists on verifying if after receiving an exception that the iFTE cannot mask, the iFTE eventually converts or propagates the exception.

6.1.3 Specification Details

About the formal representation of the iFTE abstraction, we have specified a B-Method machine to represent its stakeholders (client and server components) and the events, which flow among them (calls and returns). With the objective of creating a clear representation of the iFTE behaviour, we have represented them through sets[2]. The interactions among the iFTE and other components (requests and responses of components' services) were represented through B-Method operations. In this way, each B operation represents or a request from a component to another, or a response of a previous request (normal or abnormal return). By convention, the name of the operation is defined as follows: <source component><destination component>< _ ><kind of the event>. The <kind of the event> can be <req> for request, <resp> for a normal response, and <ABNresp> for an abnormal response. For example, the IFTEServer_req represents a service request from the iFTE component to the Server; and the IFTEServer_resp, and IFTEServer_ABNresp operations represent respectively a normal and an abnormal response for the first request. To turn the formal model more flexible, the B-Method machine represents these requests and returns events, without restrict any communication flow of them. That is, we have specified all combinations of services requests and responses among the iFTE and its stakeholders components, for example, a request from the Client to the Server component; and

[1]We do not know the exactly number because we have aborted the verification before its finishing, but is approximately 15,504 states.

[2]B-Method is based on mathematical sets.

the B-Machine permits that a response operation is executed before the respective requesting operation, what in the point of view of the call/return paradigm is forbidden. The valid operations and the correct sequence of execution are guaranteed by restrictions specified in CSP process algebra, which is complementary to the B-Method machine. Following, we detail both the B-Method machine and the CSP behavioural model specifications.

B-Method Specification. Following, we present part of the B-Method machine which represents an iFTE behaviour in a high-level way. Lines 5 to 8 show the main sets defined in the machine. Except for its first element, the `Components` set (Line 5) defines all the stakeholders components of the iFTE; the `none` element was created for implementing reasons: the machine always stores the architectural component that is actually working in a mono-thread execution (`currentComponent` variable in Line 13). If no component is activated, the value of `currentComponent` is assigned as `none`. The `client` and `server` elements represent respectively the components that use and offer services to the iFTE. The `Calls` set (Line 6) defines all possible associations between the iFTE and the other components; by default, the B-Method machine allows all possible combination of associations. The behavioural restrictions just will be defined in CSP process algebra. The `NormalReturns`, and `AbnormalReturns` sets (Lines 7 and 8) represent all possible returns used by the iFTE.

```
1  MACHINE   abstractIFTEModel
2  ...
3  SETS
4          ...
5          Components = {none, client, ifte, server};
6          Calls = {start, client_ifte, client_server, ifte_client, ifte_server,
               server_client, server_ifte};
7          NormalReturns = {norReturn};
8          AbnormalReturns = {abnReturn1, abnReturn2}
9  ...
10 VARIABLES
11         callList,               /*control variable*/
12         currentComponent,       /*control variable*/
13         activedRequests,        /*control variable*/
14 /*        sequenceHistory,        /*control variable*/*/
15         lastABNreturn,          /*control variable*/
16         unrecoverable,          /*control variable*/
17         maskableExceptions,
18         externalExceptions
19 ...
20 OPERATIONS
21 ...
22 /*a client request for the iFTE*/
23 ClientIFTE_req =
24         PRE
25                 client = currentComponent  &
26                 client_ifte /: activedRequests
27         THEN
28             ...
29         END;
```

```
30
31 /*returns normally to the client*/
32 resp <-- ClientIFTE_resp =
33         PRE
34                 size(callList(ifte))>0 &
35                 last(callList(ifte)) = client  &
36                 client_ifte : activedRequests  &
37                 ifte = currentComponent &
38                 ((lastABNreturn = noException) or (lastABNreturn :
                       maskableExceptions)) &
39                 unrecoverable = FALSE
40
41         THEN
42             ...
43         END;
44
45 /*returns an exception to the client*/
46 resp <-- ClientIFTE_ABNresp =
47         PRE
48                 size(callList(ifte))>0 &
49                 last(callList(ifte)) = client  &
50                 client_ifte : activedRequests  &
51                 ifte = currentComponent
52
53         THEN
54             ...
55         END;
56 ...
57 END
```

Lines 11 to 18 of the B-Method machine shows the main variables defined in the model. The six first variables (Lines 11 to 16) were defined for implementation reasons, and are considered "control variables". The `callList` variable (Line 11) is a function from `Components` to `seq(Components)`[3]; for each component, this variable stores the sequence of its callers (in chronological order). This sequence is useful to control when a response operation is valid (after a request) or not. The `activedRequests` variable (Line 13) is a power set of `Calls`, which indicates the requests that do not returned yet. This information is useful, for example to prevent that a service is requested more than once, before receive a return. The `lastABNreturn` (Line 15) stores the last abnormal return which the iFTE knows. Finally, the `unrecoverable` variable (Line 16) indicates if a component can recover from a failure; in other words, it stores if the iFTE have received or raised an exception which cannot be masked. Although there are many "control variables", the the developer does not need to set any one of them. About the variables which are directly useful for the application developer, there are only two of them: the `maskableExceptions`, and `externalExceptions` variables (Lines 17 and 18), which are power sets of `AbnormalReturns` and inform respectively the exceptions that can be masked (successful handled) or externally propagated (failure or interface exception) by the iFTE.

From Line 23 ahead, the machine presents only operations. As discussed before, we used

[3] `seq(Set)` in B-Method indicate a sequence of elements of `Set`.

B-Method operations to represent interactions between the iFTE and its stakeholders (Client and Server). To represent a request from a Client to an iFTE, we have defined the operation ClientIFTE_req (Line 23). This operation can only be executed when the Client is controlling the only execution thread (Line 25), and if it is not waiting for a previous request (Line 26). The other two operations represents the possible returns of the ClientIFTE_req operation: a normal response (resp < −− ClientIFTE_resp, Line 32), or an abnormal one (resp < −− ClientIFTE_ABNresp, Line 46). Both operations have the same first four preconditions: the iFTE component must to have received a service request (Lines 34 and 48); the last component that requested a service of the iFTE was the Client component (Lines 35, and 49); the requested service was not returned yet, that is, it is still actived (Lines 36, and 50); and the iFTE is controlling the processing (Lines 37 and 51). Besides that, the normal return must verify other two assertives: If if the iFTE has received or threw an exception which cannot be masked (Line 38), and if the component is in an inconsistent state (Line 39). For the sake of simplicity, we have showed just three operations, but all operations of the model behaves in a similar way. As can be seen in Appendix A, there are three operations (`*_req`, `*_resp`, and `*_ABNresp`) for each association defined in the Calls set (Line 6).

CSP Specification. After modelling the observable behaviour of an iFTE, it is necessary to represent the interaction rules between it and its stakeholders. Below, we present part of the CSP specification which restricts the sequence of the internal events (B-Method operations) permitted by the iFTE. Its restriction speeds the verification activity of ProB, due to the reduction of the combination's space.

```
1  MAIN = Start -> START2;;
2  START2 = (CLIENT [] Stop -> MAIN);;
3  CLIENT = ClientIFTE_req -> IFTE;;
4
5  -- Client -> iFTE
6  IFTE = ((IFTEServer_req -> SERVER) [] RETURN);;
7  RETURN = ((ClientIFTE_resp?R1 -> START2) [] (ClientIFTE_ABNresp?E1 -> START2))
       ;;
8
9  -- iFTE -> Server
10 SERVER = (SERVER_NORMAL [] SERVER_ABNORMAL);;
11 SERVER_NORMAL = IFTEServer_resp?R1 -> IFTE;;
12 SERVER_ABNORMAL = IFTEServer_ABNresp?E1 -> IFTE;;
```

Lines 1 and 2 of the CSP specification defines the initial process (`MAIN`), which is predefined by ProB. Then, after initiate the model (`Start` operation), the control pass to the Client (`CLIENT` process), or the machine can finish (`Stop` operation). The `[]` symbol represents an external choice of CSP. In this case, any option is valid and the verification tool combines all possibilities. When a Client assumes the control (process `CLIENT` in Line 3), it only can request a service for the iFTE component, which assumes the execution control. As shown in Lines 6 and 7, the iFTE, in turn, can requests another service from the Server component, or return to the Client (normally or abnormally). Following the sequence of execution, if the iFTE requests a service to the Server, the Server can return a normal response, or an abnormal one (Line 10). In both cases, the execution flow returns to the

iFTE component (Lines 11 and 12), which allows another external request to the Server, a normal response, or an abnormal one. The restrictions about the masking, propagation or conversion of exceptions in case of a server abnormal return depends on the specification of a particular iFTE (instantiation of the iFTE abstraction); so, it is specified in the the B-Method machine, through the preconditions of the operations. For the sake of simplicity, this example demonstrates the sequence of events involving the iFTE and only two other components: a Client and a Server. In a realistic system the iFTE can interact with any number of Clients and Servers. This example is also available in Appendix A.

6.2 Formal Verification of the iFTE Structure

In this section, we show how the internal structure of the iFTE was formally specified and verified, and how we analysed the internal propagation of exceptions in the context of a call/return implementation of the iFTE.

6.2.1 Formal Representation of the iFTE Structure

An iFTE was represented in B-Method as a machine, whose operations represents all possible events between its internal elements, both service calls and the respective returns. The B-Method machine does not define any restrictions about the valid sequence of operations. The valid communication scenarios between the iFTE's internal elements are defined through a complementary CSP specification.

The internal components of the iFTE (Provided, Required, Normal, and Abnormal) are represented as elements of the `Components` set. Beyond this, we have defined two other elements, which are used for specifying the external participants of the scenarios: Client, and Server. They represent the components that request and provide services to the specified iFTE, respectively.

Besides operations represent requests and returns of services, their name also identify the source and the destination of the requesting. For example, `ProvNor_req` operation represents a service request that the Provided internal component do for the Normal internal component. The normal and abnormal returns are represented respectively by `ProvNor_resp` and `ProvNor_ABNresp` operations. Reasoning in this way, an association between two internal components of the iFTE is determined through the operations which are allowed or forbidden. This restrictions are defined in CSP.

Besides representing the internal structure of an iFTE, the B-Method machine defines some auxiliary variables for storing information that are useful during the verification process. For example, to prevent the component mask an exception that cannot be masked, there we have defined the `unrecoverable` variable, which indicates when the iFTE cannot recover of an erroneous state.

After modelling the internal structure of the iFTE, the next step is to represent the interaction rules between its internal components. These behavioural restrictions were defined in CSP. Basically, the CSP specification must explicitly represent the valid interaction scenarios between the internal components of the iFTE. These scenarios represent the rules of service calls and how exceptions propagate internally in the iFTE with a call/return

implementation.

More details about the representation in B-Method and CSP are presented in Section 6.2.4.

6.2.2 Exception Propagation Inside the iFTE

Analysing the internal behaviour of the iFTE, we have identified 18 different scenarios, among which nine correspond to a successful execution (one normal and eight masking scenarios), and the other nine correspond to an unsuccessful execution (failure scenarios). The scenarios can also be classified as normal or abnormal. The difference between a normal and an abnormal scenario is respectively the absence or the presence of exceptions in the execution flow.

In simple terms, on a call/return implementation, when an iFTE either raises an exception, or receives one from another iFTE, the exception is automatically propagated (following the flow of the calling stack) to its Provided internal connector. In these cases, the Provided is responsible for evoking the exception handling service from the Abnormal component, which searches for a specific handler. If this handler exists, the Abnormal component can mask the exception (successful execution scenario), or can throw a failure exception (failure scenario). If there is no specific handler, the Abnormal component simply throws a failure exception (failure scenario). In these failure scenarios, all failure exceptions threw by the Abnormal component to the Provided one is propagated to the Client, which can be another iFTE. The only failure scenario that the Provided component resolve without asking the Abnormal component for a specific handling is when the Client component violates a precondition of a service during its requesting. In this case, the Provided directly raises an interface exception to the Client (failure scenario).

Concerning the handlers' specification, the Abnormal component can proceed into three different ways: (i) re-throw the exception; (ii) convert the exception into another one; and (iii) mask the exception. In order to reduce the number of possible handling that the Abnormal component can do for each exception, the formal model of the iFTE turn it possible to inform the way each specific exception can be handled. In the B-Method machine of the iFTE, the software developer can personalise three different aspects: (i) exceptions that the iFTE can raise or propagate (external exceptions); (ii) exceptions that have to be converted; and (iii) exceptions that can be masked by the handler. Exceptions that cannot be masked have to be propagated or converted, depending on the way the developer has defined the iFTE's properties. In order to implement this feature in B-Method notation, it was defined three variables: `maskableExceptions`, `failureExceptions`, and `interfaceExceptions`, which are a power set of `Exceptions`, that is, their elements are exceptions. An exception can be an element of `maskableExceptions` variable, an element of `failureExceptions` variable, an element of `interfaceExceptions`, an element of `maskableExceptions` and another one at the same time, or can be an element of none. In the last case, the exception is treated as an internal exception, which have to be converted to another by the handler (Abnormal internal component).

Since all the exceptional handling is realised by the Abnormal component, both Provided and Required components of any iFTE have a simple and pre-defined behaviour regard-

ing exception propagation. Because this, they can be automatically generated. From the perspective of the software lifecycle, this attribution of responsabilities promotes a clear separation of concerns and improve the component reusability and maintainability.

6.2.3 Verification of Properties of Interest

The behavioural consistency of the iFTE has to be verified in four complementary ways: (i) verify the existence of a valid subset of its predicted scenarios (see Section 3.3); (ii) verify the absence of impredicted scenarios (see Section 3.3); (iii) verify its internal structure (associations between internal elements); and (iv) verify the consistency of the excepcion handling policy (propagation amongst its internal elements). Due to their strict relation with the component behaviour, most off verification uses the scenario identifier tool, which has been presented in Section 5. In order to verify the integrity of the iFTE internal structure and behaviour, we have specified nine properties that are presented in Table 2. The second column of the table shows the objective of verification, while its third and fourth columns describes how the property was specified to achieve the objective and which tool had been used to this. So as can be seen, these properties intend to verify basic aspects of the iFTE, for example the absence of deadlock (Property 1), and to prevent interference between two sequential executions of an iFTE's service (Property 2). Property 2 states that when the Provided component returns a response of a service, all the internal components of the iFTE have to be ready to proceed a request, and all previous requests had to be finished. But besides these behavioural aspects of the iFTE, these properties also defines structural restrictions regarding its call/return implementation (Properties 7, 8, and 9). For clarify the B-Method notation of ProB, `&` represents the logical `AND`, the `/=` symbol means $\neq$, and the `/:` symbol means the opposite of $\in$.

Besides the basic properties of the call/return implementation of the iFTE, which were presented in Table 2, we have also verified some other properties related with exception propagation amongst the iFTE internal elements. Table 3 presents these properties and like in Table 2, its second column explain the objective of verification, while its third column describes how the property was specified to achieve the objective. For example, Properties 1 and 2 verify if the iFTE may ignore an exception that cannot be masked. Property 6 prevents that the iFTE propagates an exception that was not defined as external.

Properties only related with the state of the B-Method machine were specified in B-Method notation and had been analysed through the temporal model checking feature of ProB. To specify this using the B-Method notation of ProB, we have used the special variable `GOAL`, provided by ProB [29]. This variable can be used to define objectives to be achieved, what is indicated through a boolean expression which compares its content (`''GOAL == ...''`), what has to be defined into the `DEFINITIONS` block. Beyond this analysis, some properties are related with the consistency of execution scenarios. For these, we have initially used B-Method specification in ProB. Because the disadvantages of maintaining the `sequenceHistory` variable in a B-Method machine, which was discussed in Section 6.1.2, all properties related with the analysis of the execution scenarios are verified through the complementary tool, developed for identifying scenarios. Using this tool, the number of

Table 2: General Properties of an iFTE.

#	Objective of verification	Specification	Used Tool
1	Verify if the iFTE abstraction is correct (absence of deadlock).	Automatically verified by ProB model checker.	ProB
2	Verify if a service execution does not interfere in the next one (prevent error propagation).	`GOAL == ((activedRequests={start} & currentComponent=client) & (callList(client)/={} or callList(provided)/={} or callList(normal)/={} or callList(abnormal)/={} or callList(required)/={} or callList(server)/={}))`	ProB
3	Verify if the Abnormal component does not interfere in the normal behaviour. This component can just interfere in abnormal scenarios.	``ProvAbn_req'' can be only activated if ``ProvNor_ABNresp'' had been activated.	scenarios
4	The Provided component only request a service of the Abnormal component if an exception had been received from the Normal component.	The ``ClientProv_req -> ProvAbn_req'' sequence cannot be possible.	scenarios
5	The Provided component cannot put at risk a normal return provided by the iFTE.	The following sequences cannot be possible: ``ProvNor_resp -> ClientProv_ABNresp''; ``ProvAbn_resp -> ClientProv_ABNresp''; ``ProvNor_resp -> ProvNor_req''; ``ProvNor_resp -> ProvAbn_req''; ``ProvAbn_resp -> ProvNor_req''; and ``ProvAbn_resp -> ProvAbn_req''	scenarios
6	If the iFTE receive a valid service request from the Client component, it can just fail if the Abnormal component had tried to handle an exception (the handlind can be only search a specific handler).	The ``ClientProv_ABNresp'' can only occur immediately after ``ClientProv_req'' or ``ProvAbn_ABNresp''	scenarios
7	Inside the iFTE, the only element that knows the Server component is the Required. If change the server, it is a single point of maintenance.	`GOAL == (prov_srv /: activedRequests) & (nor_srv /: activedRequests) & (abn_srv /: activedRequests).`	ProB
8	Inside the iFTE, the only two elements that do not have dependencies between them are the Provided and Required components.	`GOAL == (prov_req /: activedRequests) & (req_prov /: activedRequests).`	ProB
9	To simplify an automatic code generation, only the Provided component depends on the Abnormal component. The Normal and Required components just propagate exceptions until the Provided.	`GOAL == (nor_abn /: activedRequests) & (req_abn /: activedRequests).`	ProB

possible states discovered by ProB decreased from more than 100,000[4] to 55 states for the abstract model of the iFTE. This reduction is because the combination of the 55 possible states with each variation of scenario (that was specified through a B-Method variable).

As an example of a specification of a property, to satisfy the objective of verification of Property #2 of Table 2, its `GOAL` could never be satisfied. Any satisfaction of this goal is a counter example that shows the violation of the property The "`beforeLast()`" function represents the penultimate element of the `sequenceHistory` sequence.

[4]We do not know the exactly number because we have aborted the verification before its finishing, but is approximately 144,079,707,346,575 states.

Table 3: Exception propagation properties of an iFTE.

#	Objective of verification	Specification	Used Tool
1	Verify if the iFTE handle all exceptions signaled by the Normal component, still these exceptions cannot be masked.	After the ‘‘ProvNor_ABNresp?E’’ event, if Abnormal cannot mask the exception ‘‘E’’, the ‘‘ProvAbn_resp’’ cannot be possible of occuring.	scenarios
2	Verify if the iFTE handle all exceptions signaled by the Required component, still these exceptions cannot be masked.	After the ‘‘ReqServer_ABNresp?E’’ event, if Abnormal cannot mask the exception ‘‘E’’, the ‘‘ProvAbn_resp’’ cannot be possible of occuring.	scenarios
3	Verify if every maskable exception signaled by the Normal component can be successful handled, tolerating the fault.	After the ‘‘ProvNor_ABNresp?E’’ event, if Abnormal can mask the exception ‘‘E’’, the ‘‘ProvAbn_resp’’ has to be possible of occuring.	scenarios
4	Verify if every maskable exception signaled by the Required component can be successful handled, tolerating the fault.	After the ‘‘ReqServer_ABNresp?E’’ event, if Abnormal can mask the exception ‘‘E’’, the ‘‘ProvAbn_resp’’ has to be possible of occuring.	scenarios
5	Verify if the iFTE promotes separation of concerns, meaning that the only component responsible for handling exceptions is the Abnormal component.	The ‘‘ProvNor_ABNresp?E’’ event has to be followed by the ‘‘ProvAbn_req’’ event, where ‘‘E’’ is a specific exception. The ‘‘AbnNor_ABNresp?E’’ event has to be followed by the ‘‘AbnNor_ABNresp?E’’ event. The ‘‘ProvNor_req −> NorReq_req −> ReqServer_req −> ReqServer_ABNresp?E’’ sequence of events has to be followed by the ‘‘NorReq_ABNresp?E −> ProvNor_ABNresp?E −> ProvAbn_req’’ sequence of events. And the The ‘‘AbnReq_req −> ReqServer_req −> ReqServer_ABNresp?E’’ sequence of events has to be followed by the ‘‘AbnReq_ABNresp?E’’ event.	scenarios
6	Verify if the iFTE signals or propagates only external exceptions.	In all ‘‘ProvAbn_ABNresp?E’’ events, ‘‘E’’ refers to an exception that was declared as external in the B-Method specification.	scenarios
7	Some specific policy of exception handling. For example, if an iFTE fails, could not to be possible to execute other services before restart the element (a new initialisation).	An instantiation-specific scenario to be verified.	scenarios

6.2.4 Specification Details

About the formal representation of the iFTE's internal structure, we have specified a B-Method machine to represent its internal elements and associations among them. With the objective of creating a clear representation of the iFTE, we have used sets[5] to explicitly represented its internal components, interfaces, and associations among components. The interactions among the internal components of iFTE (requests and responses of components' services) were represented through B-Method operations. In this way, each B operation represents or a request from a component to another, or a response of a previous request (normal or abnormal return). By convention, the name of the operation is defined as follows: `<source component><destination component>< _ ><kind of the event>`. The `<kind of the event>` can be `<req>` for request, `<resp>` for a normal response,

[5]B-Method is based on mathematical sets.

and <ABNresp> for an abnormal response. For example, the `ProvNor_req` represents a service request from the Provided component to the Normal; and the `ProvNor_resp`, and `ProvNor_ABNresp` operations represent respectively a normal and an abnormal response for the first request. To turn the formal model more flexible, the B-Method machine represents the internal structure of the iFTE, without restrict any communication flow between its internal components. That is, we have specified all combinations of services requests and responses among the iFTE's internal elements, for example, a request from the Provided to the Required component; and the B-Machine permits that a response operation is executed before the respective requesting operation, what in the point of view of the call/return paradigm is forbidden. The valid operations and the correct sequence of execution are guaranteed by restrictions specified in CSP process algebra, which is complementary to the B-Method machine. Following, we detail both the B-Method machine and the CSP behavioural model specifications.

B-Method Specification. Following, we present part of the B-Method machine which represents an iFTE internally. Lines 5 to 8 show the main sets defined in the machine. Except for its first element, the `Components` set (Line 5) defines all the internal components of the iFTE; the `none` element was created for implementing reasons: the machine always stores the internal component that is actually working in a mono-thread execution (`currentComponent` variable in Line 12). If no component is activated, the value of `currentComponent` is assigned as `none`. The `client` and `server` elements represent respectively the components that use and offer services to the iFTE. The `Calls` set (Line 6) defines all possible associations between the iFTE internal components; by default, the B-Method machine allows all possible combination of associations. The behavioural restrictions just will be defined in CSP process algebra. The `NormalReturns`, and `AbnormalReturns` sets (Lines 7 and 8) represent all possible returns used by the iFTE.

```
1  MACHINE   internalModel
2  ...
3  SETS
4        ...
5        Components = {none, client, provided, normal, abnormal, required,
             server};
6        Calls = {start, cli_prov, prov_nor, prov_abn, prov_req, nor_req,
             nor_prov, nor_abn, abn_nor, abn_req, abn_prov, req_abn, req_nor,
             req_prov, req_srv, prov_srv, nor_srv, abn_srv};
7        NormalReturns = {norReturn};
8        AbnormalReturns = {intAbnReturn, extAbnReturn}
9  ...
10 VARIABLES
11       callList,                /*control variable*/
12       currentComponent,        /*control variable*/
13       activedRequests,         /*control variable*/
14 /*      sequenceHistory,          /*control variable*/*/
15       lastABNreturn,           /*control variable*/
16       lastExternalABNreturn,   /*control variable*/
17       unrecoverable,           /*control variable*/
18       maskableExceptions,
```

```
19         failureExceptions,
20         interfaceExceptions
21         ...
22 OPERATIONS
23 ...
24 /*a client request for the provided internal component*/
25 ClientProv_req =
26         PRE
27                 client = currentComponent &
28                 cli_prov /: activedRequests
29         THEN
30            ...
31         END;
32
33 /*returns normally to the client*/
34 resp <-- ClientProv_resp =
35         PRE
36                 size(callList(provided))>0 &
37                 last(callList(provided)) = client &
38                 cli_prov : activedRequests &
39                 provided = currentComponent
40
41         THEN
42            ...
43         END;
44
45 /*returns an exception to the client*/
46 resp <-- ClientProv_ABNresp =
47         PRE
48                 size(callList(provided))>0 &
49                 last(callList(provided)) = client &
50                 cli_prov : activedRequests &
51                 provided = currentComponent
52         THEN
53            ...
54            ANY val WHERE val : AbnormalReturns & ((val : interfaceExceptions)
                  or (val = raisedExternalException)) & val /= noException
55               THEN
56                  ...
57                  resp := val ||
58                  ...
59               END
60         END;
61 ...
62 END
```

Lines 11 to 20 of the B-Method machine shows the main variables defined in the model. The seven first variables (Lines 11 to 17) are the same "control variables", which where presented in Section 6.1.3 and were defined for implementation reasons. About the variables which are directly useful for the application developer, there are only three of them: the `maskableExceptions`, `failureExceptions`, and `interfaceExceptions` variables (Lines 18 to 20), which are power sets of `AbnormalReturns` and inform respectively the exceptions that

can be masked (successful handled) or externally propagated (failure ou interface exception) by the iFTE.

From Line 25 ahead, the machine presents only operations. As discussed before, we used B-Method operations to represent interactions amongst the iFTE internal elements, as well with the iFTE's stakeholders (Clients and Servers). To represent a request from a Client to an iFTE, we have defined the operation ClientProv_req (Line 25), because the Provided component is the boundary between the iFTE and its external Clients. This operation can only be executed when the Client is controlling the only execution thread (Line 27), and if it is not waiting for a previous request (Line 28). The other two operations represents the possible returns of the ClientProv_req operation: a normal response (resp < −− ClientProv_resp, Line 34), or an abnormal one (resp < −− ClientProv_ABNresp, Line 46). Both operations have the same preconditions to execute: the Provided internal connector must to have received a service request (Lines 36 and 48); the last component that requested a service of the Provided was the Client component (Lines 37, and 49); the requested service was not returned yet, that is, it is still activated (Lines 38, and 50); and Provided internal connector is controlling the processing (Lines 39 and 51). For the sake of simplicity, we have showed just three operations, but all operations of the model behaves in a similar way. As can be seen in Appendix B, there are three operations (`*_req`, `*_resp`, and `*_ABNresp`) for each association defined in the Calls set (Line 6).

Despite they are similar, there is an important difference among the behaviour of these operations. Line 54 of the source-code shows an important one: *the abnormal return of an operation has to be consistent with the role of the internal element of the iFTE.* For example, in Line 54 the `ClientProv_ABNresp` operation just returns or an interface exception, or a failure exception, propagated from the Abnormal component (stored in the `raisedExternalException`) control variable, available in Section B.1).

CSP Specification. After modelling the internal structure of the iFTE, it is necessary to represent the interaction rules between its internal elements. Below, we present part of the CSP specification which restricts the sequence of the internal events (B-Method operations) permitted by the iFTE. Its restriction speeds the verification activity of ProB, due to the reduction of the combinational space.

```
1  MAIN = Start -> START2;;
2  START2 = (CLIENT [] Stop -> MAIN);;
3  CLIENT = ClientProv_req -> PROVIDED;;
4  ...
5  PROVIDED = ((ProvNor_req -> NORMAL) [] INTERFACE_EXCEPTION);;
6  ...
7  NORMAL = (NORMAL_NORMAL_REPONSE [] NORMAL_RAISES_EXCEPTION [] (NorReq_req ->
       REQUIRED));;
8  ...
9  REQUIRED = ReqServer_req -> (EXTERNAL_NOR [] EXTERNAL_ABN);; --external
       service
10 ...
```

Lines 1 and 2 of the CSP specification define the initial process (`MAIN`), which is predefined by ProB. Then, after initiate the model (`Start` operation), the control pass to the

Client (CLIENT process), or the machine can finish (Stop operation). The [] symbol represents an external choice of CSP. In this case, any option is valid and the verification tool combines all possibilities. When a Client assumes the control (process CLIENT), it only can request a service for the Provided internal connector (Line 3), which assume the execution control. As shown in Line 5, the Provided, in turn, can either propagate the requisition to the Normal internal component, or detect and throw an interface exception. Following the sequence of the iFTE internal behaviour, if the requesting is propagated to the Normal component, it can return a normal response, raise an exception, or request an external service through the Required internal connector (Line 7). Finally, if the Required connector receives a service request (Line 9), it requests the service for a Server component, and can receive a normal (EXTERNAL_NOR) or an abnormal (EXTERNAL_ABN) response. For the sake of simplicity, this example do not demonstrate all possible scenarios described in Section 3. All CSP specification of the internal structure of the iFTE is available in Appendix B.

6.3 Formal Verification of High-Level Software Architectures

In this section, we present how a software architecture composed of iFTEs' architectural elements should be verified. This verification allows the developer to analyse the propagation of exceptions among the architectural elements, and to define application-specific architectural policies for exception flow.

6.3.1 Formal Representation of High-Level Software Architectures

To formally represent software architectures, we have done two complementary models: a B-Method machine and a CSP specification to guide the machine. The specification of software architectures in B-Method consisted on representing all architecture's information, which is relevant to understand the model: (i) architectural elements (components and connectors with their interfaces); and (ii) the way its elements are disposed together (the architecture configuration). In our case, every architectural elements are iFTEs and in B-Method are represented as elements of the `ArchitecturalElements` set. The provided interfaces of the architectural elements are also represented as elements of a set: the `Interfaces` set. The association among architectural elements and interfaces are assigned through a function from `ArchitecturalElements` to `Interfaces`. Finally, the architecture configuration is specified in terms of dependencies between iFTEs. These dependencies are also represented through a function from `ArchitecturalElements` to a power set of `ArchitecturalElements`.

Another important issue, due to our emphasis on exception propagation, is the representation of the types of exceptions that each architectural element can throw (raise or propagate) and mask (transparently to the element's client). The architectural exceptions are represented as elements of a set (`AbnormalReturns`). For indicating the exceptions that each iFTE can mask and that each interface can throw, we have defined two more mathematical functions in B-Method notation. In order to make it possible to represent exception handling by subsumption, all exceptions can be organised in a type hierarchy, which is implemented through another relation, specified between exceptions.

All these mentioned functions and other variables of the B-Method machine will be detailed in Section 6.3.4.

The events (calls and returns) among architectural elements are also represented through B-Method operations: one for the service requesting, and other two for its possible returns (normal and abnormal). But to allow a component for requesting a service from another one, the machine had to have defined a dependency between the source and the destination components of the requisition. Despite this restriction about dependencies among components, concerning the interaction among architectural elements, the B-Method machine does not impose other restrictions on the sequence of executed operations. These "advanced restrictions" on the way architectural components communicate are specified using the CSP process algebra.

The high-level view of the software architecture (omitting the iFTE's internal elements) reduces the complexity of the logical view of the architecture. In this way, it is possible to analyse the exception propagation of components with a "gray-box" view of the system components. It also facilitates the specification of exception flow policies with high-level specification units, which can be detailed in a future development phase (a top-down incremental design). These exception propagation policies are also specified in CSP. An example of a recommended architectural policy of exception flow is:

> *After an iFTE receives an abnormal return of an external requesting, it have to try again (possibly with another server) trying to mask the error.*

This policy forces the element to request another service to substitute the failed one (or re-try the same) at least once.

More details about the representation of the B-Method machine and the CSP process algebra are presented in Section 6.3.4.

6.3.2 Exception Propagation between iFTEs

From the point of view of a software architecture, there are just five scenarios of exception propagation: two correspond to the successful execution scenarios, and three correspond to the unsuccessful execution scenarios. When an iFTE receives no exception from its dependents, it can either return normally ("no exception" - the normal scenario, successful execution), or return abnormally ("exception raising" - an abnormal scenario, unsuccessful execution). On the other hand, when an architectural element receives exceptions from its server components, it can behaves in three different ways: (i) return normally ("masking the exception" - an abnormal scenario, but a successful execution); (ii) returns the same exception ("propagation" - an abnormal scenario, unsuccessful execution); and (iii) returns another exception ("exception conversion" - an abnormal scenario, unsuccessful execution).

Remembering that in order to reduce the state space of verification, the way that an architectural element deals with an exception is defined in the B-Method machine. In the B-Method machine of the iFTE, the architect can personalise four different aspects: (i) exceptions that each iFTE can raise or propagate (external exceptions); (ii) exceptions that the iFTE has to convert into another one; (iii) exceptions that can be masked by the handler;

and (iv) components that does not raise any exception (but can handle and propagate). Exceptions that cannot be masked have to be propagated or converted, depending on the way the architect has defined the iFTE's properties. In order to implement these four personalised aspects in B-Method notation, it was defined three variables: `faultfreeComponents`, `maskableExceptions`, and `interfaceExceptions`. The first one is a power set of ArchitecturalElements, that is, their elements are architectural elements. The `maskableExceptions` is a function from `AbstractElements` to a power set of `AbnormalReturns` and relates for each iFTE, the set of exceptions that it can mask. Finally, the `interfaceExceptions` variable is a function from `Interfaces` to a power set of `AbnormalReturns` and relates for each interface, the set of exceptions that can be propagated (external exceptions). The elements of the `faultfreeComponents` and `maskableExceptions` variables (components and exceptions) are verified in the preconditions of the B-Method operations; and the elements of `interfaceExceptions` refines the exceptions that can be threw by each provided interface.

6.3.3 Verification of Properties of Interest

In order to verify the integrity of systems constructed using iFTEs as building blocks, we have specified seven properties that should be verified for fault-tolerant software architectures based on iFTEs. Table 4 presents these properties. The second column of the table shows the objective of verification, while its third column describes how the property was specified to achieve the objective. So as can be seen, these properties intend to verify the basic aspects of fault-tolerant architectures and other assumptions to simplify the architecture. For example, Property 1 verifies the absence of deadlock, while Property 2 prevents iFTEs execute concurrently (we assume sequential execution of the architectural elements). Property 2 states that when an iFTE requests a service, it waits the respective response (normal or abnormal). Only after receive this response, it can requests another service. This assumption about the absence of concurrence intends to simplify the specification of small and common systems, which does not involve concurrence. As presented in Section 8, an ongoing work is to extend this work for dealing with concurrent components. Property 3 states that an interface (provided or required) only throws exceptions that are explicitly defined in its specification. It is an important information, because the exceptions defined in the provided interfaces have to be handled by the iFTE. The specified properties also deal with elementary properties of software architectures, for example Property 7, which states that an architectural element only receives requests through its provided interfaces. The events presented in Table 4 are expressed in a "CSP-like" notation, where `''?''` separates the name of the event (before it) and the value returned by it (when there is one), and `''!''` identify the arguments of the event. The order of the arguments is dicted by the B-Method operation, presented in Section 6.3.4 and listed in full in Appendix C.

Besides the basic properties of an iFTE-based software architecture, which were presented in Table 4, it was also verified ten other properties related with exception propagation among iFTEs in the software architecture. Table 5 presents these properties and like in Table 4, its second column explain the objective of verification, while its third column describes how the property was specified to achieve the objective. For example, Properties 1 and 2 verify if the iFTE elements of the architecture handle all exception declared

Table 4: Properties of an architectural configuration with iFTE elements.

#	**Objective of verification**	**Specification**	**Used Tool**
1	Verify if the way the iFTE components are configured in the software architecture produces no are free of deadlock.	Automatically verified by ProB model checker.	ProB
2	Verify if there is iFTE components executing concurrently in the software architecture. It is necessary, because we are temporarily assuming in this work that there is no concurrency in the software architecture.	It is not possible to have two ``Service_req!a!b'' events, without a ``Service_resp!a!b!i1'' or a ``Service_ABNresp!a!b!i1'' event occur between them. When ``a'' is the source of the request, and ``b'' is the destination of the request.	ProB preconditions
3	Verify if there are exceptions, which are propagated and are not explicity defined in the specification of the interface.	If occur a ``Service_ABNresp!a!b!i1?E'' event, ``E'' has to be an external exception of the ``b'' architectural element.	scenarios
4	The iFTEs in the software architecture have to communicate in a call/return way.	A ``Service_resp!a!b!i1'' or a ``Service_ABNresp!a!b!i1'' event can only occur after a ``Service_req!a!b!i1''	scenarios
5	Verify if the iFTEs only mask exceptions defined as maskable.	After a ``Service_ABNresp!a!b!i1?E'' event, if ``E'' could not be masked by the ``a'' component, it is not possible to execute the ``Service_resp!a!x!y?E'' event in this scenario, for any ``x'' component and ``y'' interface.	scenarios
6	All the architectural elements have to be compatible with the architectural restrictions (communication rules).	The following events are only possible if ``a'' depends on ``b'': ``Service_req!a!b!i1?E'', ``Service_resp!a!b'', and ``Service_ABNresp!a!b!i1?E''.	scenarios
7	Verify if each service request of the iFTE architectural elements are done through a valid provided interface.	The following events are only possible if ``b'' provides the ``i1'' interface: ``Service_req!a!b!i1?E'', ``Service_resp!a!b!i1'', and ``Service_ABNresp!a!b!i1?E''.	scenarios

in their `I_ReceivedExceptions` interfaces. Property 3 is equivalent to (*Property* 1 *AND Property* 2) Property 4 prevents that an iFTE propagates an exception which was not defined in its interfaces. But besides these general aspects of exception propagation among iFTE architectural elements, it is possible to define restrictions which are application-specific, for example Property 5 has to be specified according to a specific objective of a specific application. This property turns it possible to analyse the tracing of a specific exception in a scenario of exception propagation. It is useful for verifying the transformation of the exception during the flow. Properties 6 and 7 have to be specified according to a specific objective of a specific application. These properties turn it possible to verify if a specific scenario of exception propagation can occur or not in a specific application. It is useful for verifying if the architecture restricts a valid scenario or allows an invalid one.

All properties were specified in B-Method notation and had been analysed through animating the model (B-Method machine and CSP specification) in ProB model checker.

Table 5: Exception propagation properties of an architectural configuration with iFTE elements.

#	Objective of verification	Specification	Used Tool
1	Verify if the iFTEs handle all exceptions that cannot be masked and are declared in its required interfaces.	If ``E'' is an exception that ``a'' cannot mask, the ``Service_ABNresp!a!b!z?E'' event have to be followed by the ``Service_ABNresp!x!a!y?E2'' event, for any ``x'' and ``y'', where ``E2'' is an external exception declared on ``y''.	scenarios
2	Verify if the iFTEs handle all exceptions that can be masked and are declared in its required interfaces.	If ``E'' is an exception that ``a'' can mask, have to exist a scenario that has both the ``Service_ABNresp!a!b!z?E'' and ``Service_resp!x!a!y'' events, for some ``x'', ``y'', and ``z''.	scenarios
3	Verify the absence of handling (neither propagation, conversion, nor masking) for some exceptions.	*Property* 1 *AND Property* 2	scenarios
4	Verify if all exceptions that are propagated, are explicitly defined in the specification of the interface.	Verify if there is no scenario where occur the ``Service_ABNresp!a!b!i1?E'' event, where E is an exception that is not assigned to the ``I1'' interface.	scenarios
5	Verify the specification of a specific pattern of interaction (application-specific).	Verify if all sequence of operations follow the pattern that was specified. For example, a mandatory retry (``Service_req!a!b2!i2'') after receiving an abnormal return (``Service_ABNresp!a!b1!i1?E'').	scenarios
6	Verify the possibility of occurring desired application-specific scenarios.	Verify if there is at least one scenario where occur the desired sequence of operations: ``op1!parameters?return −> op2!parameters?return −> ...''.	scenarios
7	Verify the absence of undesired application-specific scenarios.	Verify if there is no scenario where occur the specified sequence of operations: ``op1!parameters?return −> op2!paramethers?return −> ...''.	scenarios
8	Verify if all external exceptions flows in the architecture (prevent useless exceptions).	Verify if there is some external exception that is never equals for ``E'' in the ``Service_ABNresp!a!b!x?E'' events, for any ``a'', ``b'', or ``x''.	scenarios
9	Verify if the iFTEs do not mask exceptions that cannot be masked by them.	If ``E'' is an exception that ``a'' cannot mask, a scenario cannot present a ``Service_ABNresp!a!b!z?E'' before a ``Service_resp!x!a!y'' event, for any ``x'' and ``y''.	scenarios
10	Tracing of an exception in predefined application-specific scenarios of exception propagation.	Use the simulation feature of ProB tool.	ProB

Beyond this initial analysis, we can use the feature of temporal model checking of ProB, in order to verify the satisfaction (or not) of all properties. To verify this using the B-Method notation of ProB, we have used the special variable `GOAL`, provided by ProB [29]. This variable can be used to define objectives to be achieved. This is indicated through an expression of attribution into the `DEFINITIONS` block of the machine. As an example, the definition of Property #2 of Table 4 was specified as follows: `GOAL == (card(activedRequests(currentComponent)) > 1)`. As explained in details in Section 6.3.4, the `activedRequests` variable stores the set of requests that a specific component

had done and do not finished yet. To satisfy the objective of the property, this `GOAL` could never be satisfied. Any satisfaction of this goal is a counter example that shows the violation of the property To verify the exception propagation properties, it was necessary to analyse the sequence of executed operations of the B-Method machine. As explained in Section 6.3.1, this sequence represents all requests and returns that happened among iFTEs in the software architecture. The traceability of the sequence of executed operations is achieved through the complementary tool, developed to analyse the execution scenarios of the CSP process algebra end the pre-conditions of the B-Method operations.

6.3.4 Specification Details

To provide a clear understanding, Figure 35 presents an example of a simple software architecture. This example consists on six iFTE architectural elements, two of them are connectors. To simplify the model, it is possible to represent the abnormal interfaces in a implicit way. In these cases, the architectural elements have to switch the stereotypes $<< iFTComponent >>$ and $<< iFTConnector >>$ for the $iFTEComponent$ and $iFTEConnector$, respectively (Figure 36). There is a client component (Client1), and three server components (Server1, Server2, and Server1and2). The Client1 component requires two interfaces: Interface1 and Interface2, which are provided by the servers components. The Server1 component provides the Interface1 interface, the Server2 component provides the Interface2, and the Server1and2 provides both interfaces, acting as a redundant staff for dependability reasons. The C1_S1 connector is responsible for intermediate the access between the Client1 component and the providers of the Interface1 interface (Server1 and Server1and2). This connector is also responsible to switch the responsibility request from a server to another, in case of failure.

Now we will present how this software architecture was specified in B-Method and CSP.

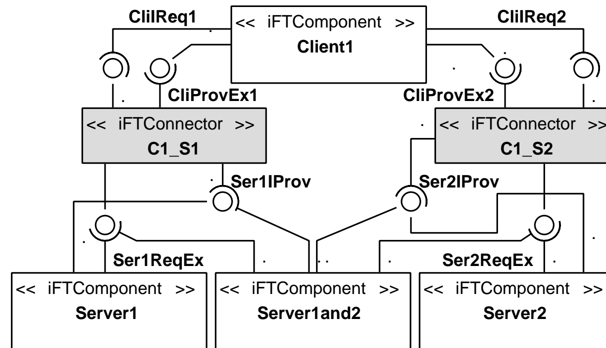

Figure 35: A Simple software architecture (complete view).

B-Method Specification. Following, we present part of the B-Method machine which represents the software architecture presented in Figure 35 in a high-level way (without the internal details of each iFTE element). Lines 5 to 8 presents the main sets which were

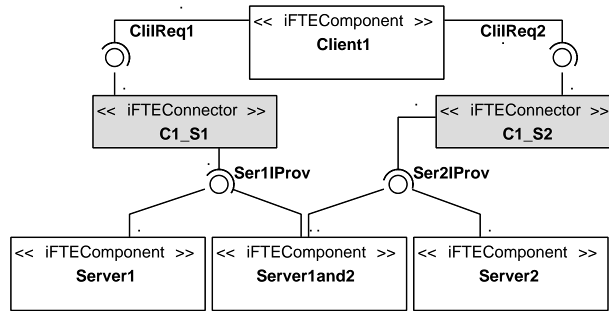

Figure 36: A Simple software architecture (simplified view).

specified in this model. The first two sets (Lines 5 and 6) represents respectively the normal and abnormal types that flow in the software architecture. The `ArchitecturalElements` set (Line 7) defines the list of components and connectors of the architecture, according to a specific system. And the Interfaces set (Line 8) defines the list of every interfaces existing in the software architecture.

```
1  MACHINE hiLevelArchitecture
2  ...
3  SETS
4          ...
5          NormalReturns = {norReturn};
6          AbnormalReturns = {noException, abnReturn1, abnReturn2};
7          ArchitecturalElements = {none, client1, server1, server1and2, server2,
               c1_s1, c1_s2};
8          Interfaces = {interface1, interface2}
9  VARIABLES
10         ...
11         dependencies,
12         faultfreeComponents,
13         ...
14         componentInterfaces,
15         interfaceExceptions,
16         maskableExceptions,
17         exceptionHierarchy,
18         handlingBySubsumption
19 ...
20 OPERATIONS
21 ...
22 Service_req(from, to, interface) =
23         PRE
24                 from : ArchitecturalElements & to : ArchitecturalElements & to
                       : dependencies(from) &
25                 from = currentComponent & to /: activedCalls(from) &
26                 interface : Interfaces & interface : componentInterfaces(to)
27         THEN
28                 ...
29         END;
30
```

```
31 resp <-- Service_resp(from, to, interface) =
32         PRE
33                 from : ArchitecturalElements & to : ArchitecturalElements & to
                       : dependencies(from) &
34                 to : activedCalls(from)  & to = currentComponent &
35                 interface : Interfaces & interface : componentInterfaces(to) &
36                 lastRequisition(from|->to) = interface &
37                 tp /: unrecoverable
38         THEN
39                 ...
40         END;
41
42 resp <-- Service_ABNresp(from, to, interface) =
43         PRE
44                 from : ArchitecturalElements & to : ArchitecturalElements & to
                       : dependencies(from) &
45                 ... // if the component is not defined as ``fault-free'' &
46                 to : activedCalls(from)  & to = currentComponent &
47                 interface : Interfaces & interface : componentInterfaces(to) &
48                 lastRequisition(from|->to) = interface &
49         THEN
50                 ...
51         END
52 ...
53 END
```

Despite all information about architectural elements have been defined through B-Method sets, the way as they become related was not defined in the same way. Dependencies among components, the list of interfaces that each component provides, exceptions that each interface throws, exceptions that each component can mask, and the typing hierarchy of exceptions are defined in the B-Method machine respectively through the `dependencies` (Line 11), `componentInterfaces` (Line 14), `interfaceExceptions` (Line 15), `maskableExceptions` (Line 16), and `exceptionHierarchy` (Line 17) variables. The `faultfreeComponents` variable (Line 12) is a power set of `ArchitecturalElements` and stores the set of architectural elements which are fault-free, that is, these architectural elements do not raise exceptions. Finally, the `handlingBySubsumption` (Line 18) is a Boolean variable which indicates if the machine will (or will not) handle exceptions by subsumption. These variables are considered in the operations pre-conditions to indicate when a specific request of service is allowed or denied.

Except by the Start and Stop operations, which respectively initialises and finishes the model execution, there are just other three operations, which are presented in the B-Method machine. For the sake of simplicity, parts of its pre-conditions were simplified and showed just its meaning. For a component requests a service of another one, it has to evoke the `Service_req(from, to, interface)` operation (Line 22), where `from` is the source component of the requesting, `to` is the destination component, and `interface` is the interface through the service is requested. Lines 24 to 26 presents the preconditions to a component request a service of another one. As presented in Line 24, a service can just be requested when there is a dependency, which is explicitly defined, between the source component (`from` argument of the B-Method operation) and the destination component (`to` argument

of the B-Method operation). Besides that, Line 25 states that the requesting can only occur if the the source component is not waiting for a previous requesting to the destination component. Finally, to prevent other inconsistencies of the software architecture, Line 26 states that the destination component (`to` argument of the B-Method operation) has to provide the interface which was requested by the source component (`from` argument of the B-Method operation).

The possible responses of a request are also represented through operations. The `Service_resp(from, to, interface)` operation (Line 31) represents the normal return, and the `Service_ABNresp(from, to, service)` operation (Line 42) represents the abnormal one. These two operations also verify the dependence between components (Lines 33 and 44), verify if the source component had requested the service before (Line 34, and 46), verify if the component provides the specified interface (Lines 35 and 47), and guarantee that the component only responds if a request had been done from 'from' to 'to' through the 'interface' interface (Lines 36 and 48). Beyond this, the return operations verify other particularities. For example, if a component has received an abnormal return, it only can return normally if the exception which was received can be masked (directly or by subsumption), as stated in Line 37.

CSP Specification. Differently from the internal representation of the iFTE, in this high-level software architecture model, the basic restrictions of communication are specified directly in the B-Method machine. These restrictions are defined through the provided interfaces and dependencies among components. As explained before the preconditions of the B-Method operations are responsible for verifying these structural restrictions. These restrictions have to be done in B-Method because they are state-related, that is, to verify them it is necessary to analyse the elements of some sets, and the values of some variables. The CSP model is used in two complementary ways: (i) to reduce the state space of the verification process; and (ii) to define specific policies for exception handling and exception propagation. The reduction of the state space is achieved defining a component for starting all interaction scenario. This component, called "the top-level client component" behaves like the graphical interface of an application. Line 1 of the following CSP code defines the `client1` component as a "top-level". About the policies for exception handling, Lines 8 and 9 of the CSP code represents a policy specified specifically for this application. It is important stress that all other restrictions, which were defined in B-Method are still valid. Lines 8 and 6 states that after a component receives an abnormal return, despite the exception is maskable, it cannot provide a normal return; unless the component request another external service and receive a normal response. The addition of this policy for exception handling means that for this specific application, the Abnormal component of an iFTE element cannot implement itself the services provided by external components.

```
1 MAIN = Start!client1 -> REQUEST1;;
2 REQUEST1 = ((Service_req!from!to!interface -> EXEC) [] Stop -> MAIN);;
3
4 -- a -> a_b
5 EXEC = (RETURN1 [] REQUEST2);;
6 RETURN1 = ((Service_resp!from!to!interface?N -> REQUEST1) [] (Service_ABNresp!
```

```
    from!to!interface?E -> REQUEST1));;
7
8 ABN_EXEC = (RETURN1_ERR [] REQUEST2);;
9 RETURN1_ERR = (Service_ABNresp!from!to!interface?E -> REQUEST1);; --if receive
     an exception, it is not possible to return normally without another
    external request
10
11 -- a_b -> b
12 REQUEST2 = (Service_req!from!to!interface -> RETURN2);;
13 RETURN2 = ((Service_resp!from!to!interface?N -> EXEC) [] (Service_ABNresp!from
    !to!interface?E -> ABN_EXEC));;
```

6.4 Formal Verification of Detailed Software Architectures

As the high-level software architecture, the detailed specification of the software architecture also represents all the architectural elements, which can be iFTEs. But in this new model, the iFTE elements are represented in a more detailed way. This detailing consists on specifying the internal elements of each iFTE architectural element (components and connectors). In this way, it is possible to verify the architectural properties, analysing the development view of the software architecture (lower granularity components). Because it encloses the characteristics of both high-level architecture specification (Section 6.3) and the internal structure of the iFTE (Section 6.2), the detailed specification of the software architecture explicitly combines these models and proceed a combined verification of them.

6.4.1 Formal Representation of Detailed Software Architectures

As mentioned in Section 6.3, to formally represent software architectures, we have to represent all architecture's information, which is relevant to understand the model: (i) architectural elements (components and connectors with their interfaces); and (ii) the way its elements are disposed together (the architecture configuration). Section 6.3 also has presented that these architectural information is modelled through a B-Method machine. The machine of the detailed view of the software architecture is similar to the machine of the high-level view. Both have four mathematical sets, whose utility was explained in Section 6.3.1: `ArchitecturalElements`, `Interfaces`, `NormalReturns`, and `AbnormalReturns`. The association among architectural elements and interfaces are assigned through a function from `ArchitecturalElements` to `Interfaces`. Finally, the architecture configuration is specified in terms of dependencies between iFTEs. These dependencies are also represented through a function from `ArchitecturalElements` to a power set of `ArchitecturalElements`.

Despite the architectural elements of the software architecture, the detailed view have to represent the internal details of each architectural element. For this, we have used the feature of extending B-Method machines, which ProB implements. In this way, it is possible to reuse other machines to add behaviour into another one. So, for each iFTE architectural element, we extends the software architecture with the behaviour of the B-Method machine of the element (the iFTE structure discussed in Section 6.2). For example, a software architecture with two iFTE elements have to extends two other B-Method machines of iFTE Structure.

Because all events (calls and returns) among architectural elements are represented through B-Method operations, the detailed view of the software architecture has to control both the architectural behaviour (all elements) and internal behaviour (iFTE elements). The operations related to the internal behaviour of the iFTE are inherited by the machine of the software architecture, through the extension mechanism.

Besides the inclusion of the internal behaviour of the iFTE elements of the architecture, it is necessary to coordinate the execution sequence of the machine's operations. This coordination is specified in CSP and has to control when an architectural interface can be executed, and when an internal operation can be executed. In other words, the CSP process specification has to enclose both the coordination of the high-level architecture specification, and the coordination of the iFTE structure specification.

More details about the representation in B-Method and CSP algebra are presented in Section 6.4.4.

6.4.2 Exception Propagation between iFTEs

Had to the high-level and detailed specifications of the software architecture deal with the same application artifact (the software architecture), the execution scenarios of these two models are compatible. That is, abstracting the way the internal components of the iFTE elements interact each other, the detailed specification has the same scenarios of the high-level one. But instead of the high-level specification have five possible scenarios, as presented in Section 6.3.2, the detailed specification of the software architecture presents one more, because the scenario where an architectural element raises an exception is splitted in two scenarios, according the way the exception is raised: (i) an interface exception, which only involves the Provided component; and (ii) a failure exception, which involves the Provided, Normal, and Abnormal internal components.

Including the new scenario which is specific of the detailed specification model, the software architecture has six scenarios of exception propagation: two correspond to the successful execution scenarios, and four correspond to the unsuccessful execution scenarios. When an iFTE receives no exception from its dependents, it can either return normally ("no exception" - the normal scenario, successful execution), or return abnormally ("exception raising" - an abnormal scenario, unsuccessful execution). But as discussed before, an exception can be raised as an interface exception, or as a failure exception. On the other hand, when an architectural element receives exceptions from its server components, it can behaves in three different ways: (i) return normally ("masking" - an abnormal scenario, but successful execution); (ii) returns the same exception ("propagation" - an abnormal scenario, unsuccessful execution); and (iii) returns another exception ("exception conversion" - an abnormal scenario, unsuccessful execution).

Remembering that as presented in Section 6.3.2, to reduce the state space of verification, it is also possible to specify the same handling aspects of the high-level specification: (i) exceptions that each iFTE can raise or propagate (external exceptions); (ii) exceptions that the iFTE has to convert into another one; (iii) exceptions that can be masked by the handler; and (iv) components that does not raise any exception (but can handle and propagate).

6.4.3 Verification of Properties of Interest

Had to the high-level and detailed specifications of the software architecture deal with the same application artifact (the software architecture), the execution scenarios of these two models are compatible. So, we have verified in the architecture detailed model, the same properties of the high-level architecture model, which were presented in Section 6.3.3. There are seven properties for verifying the integrity of systems which use iFTEs as building blocks (Table 4), and seven other properties related with exception propagation among iFTEs in the software architecture (Table 5). An important characteristic of the detailed specification model of software architectures is the necessity of verifying the each iFTE architectural element internally, with the properties presented in Section 6.2.3.

These properties intend to verify the basic aspects of fault-tolerant architectures and other assumptions to simplify the architecture. For example, Property 1 of Table 4 verifies the absence of deadlock, while Property 3 of Table 4 states that an interface (provided or required) only throws exceptions that are explicitly defined in its specification. It is an important information, because the exceptions defined in the provided interfaces have to be handled by the iFTE. Besides that, the specified properties also deal with elementary properties of software architectures, for example Property 7 of Table 4 states that an architectural element only receives requests through its provided interfaces.

The properties related with exception propagation (of Table 5) verify both general aspects, and specific ones. Some examples of general aspects of the exception handling and exception propagation: (i) to verify if the iFTE elements of the architecture handle all exception declared in their `I_ReceivedExceptions` interfaces; and (ii) to prevent that an iFTE propagates an exception which was not defined in its interfaces. Some examples of aspects of the exception handling and exception propagation, which are specific of the application: (i) to analyse the tracing of a specific exception in a scenario of exception propagation; and (ii) to verify if a specific scenario of exception propagation can occur or not in a specific configuration.

As presented in Section 6.3.3, all properties were specified in B-Method notation and had been analysed through animating the model (B-Method machine and CSP specification) in ProB model checker. Beyond this initial analysis, we can use the feature of temporal model checking of ProB, in order to verify the satisfaction (or not) of all properties.

6.4.4 Specification Details

To provide a clear understanding of the model, Figure 37 presents an example of a simple client/server software architecture. This example consists on three iFTE architectural elements: two components and a connector. The Client component requires two interfaces: Interface1 and Interface2, which are provided by the server. The Server component provides the Interface1 and Interface2 interfaces. The Connector connector is responsible for intermediate the access between the Client and the Server components.

Now we will present how this software architecture was specified in B-Method and CSP.

B-Method Specification. Following, we present part of the B-Method machine which represents the software architecture presented in Figure 37 in a detailed way (with the

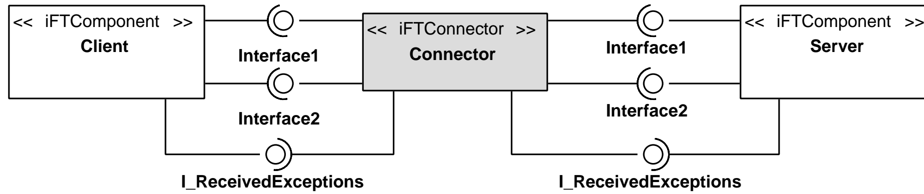

Figure 37: A Simple client/server software architecture.

internal details of each iFTE element). As can be seen in this listing, the B-Method machine of the detailed model of the software architecture is almost equals to the B-Method machine of the high-level specification, presented in Section 6.3.1. The only difference is the inclusion of the internal behaviour of each iFTE architectural element. Lines 5 to 9 presents the main sets which were specified in this model. Note they are the same sets of the high-level architecture.

```
1 MACHINE clientServerConfiguration
2 ...
3 SETS
4         ...
5         NormalReturns = {normalReturn};
6         AbnormalReturns = {noException, abnReturn1, abnReturn2};
7         ArchitecturalElements = {none, clientComponent, connector,
              serverComponent};
8         DetailedArchitecturalElements = {connector, serverComponent};
9         Interfaces = {interface1, interface2}
10 EXTENDS
11    conn,
12    server
13 VARIABLES
14        ...
15        dependencies,
16        faultfreeComponents,
17        ...
18        componentInterfaces,
19        interfaceExceptions,
20        maskableExceptions,
21        exceptionHierarchy,
22        handlingBySubsumption
23 ...
24 OPERATIONS
25 ...
26 Service_req(from, to, interface) =
27        PRE
28                from : ArchitecturalElements & to : ArchitecturalElements & to
                      : dependencies(from) &
29                from = currentComponent & to /: activedCalls(from) &
30                interface : Interfaces & interface : componentInterfaces(to)
31        THEN
32                ...
33        END;
```

```
34
35 resp <-- Service_resp(from, to, interface) =
36         PRE
37                 from : ArchitecturalElements & to : ArchitecturalElements & to
                       : dependencies(from) &
38                 to : activedCalls(from)  & to = currentComponent &
39                 interface : Interfaces & interface : componentInterfaces(to) &
40                 lastRequisition(from|->to) = interface &
41                 to /: unrecoverable
42         THEN
43                 ...
44         END;
45
46 resp <-- Service_ABNresp(from, to, interface) =
47         PRE
48                 from : ArchitecturalElements & to : ArchitecturalElements & to
                       : dependencies(from) &
49                 ... // if the component is not defined as ''fault-free'' &
50                 to : activedCalls(from)  & to = currentComponent &
51                 interface : Interfaces & interface : componentInterfaces(to) &
52                 lastRequisition(from|->to) = interface &
53         THEN
54                 ...
55         END
56 ...
57 END
```

Had to the similarity between the high-level B-Method machine and the detailed architecture one, the architectural service requests and the respective responses are also represented through the same operations. The `Service_req(from, to, interface)` operation (Line 26) represents a service request, where `from` is the source architectural component of the requesting (not an internal element of the iFTE), `to` is the destination architectural component, and `interface` is the interface through the service is requested. The `Service_resp(from, to, interface)` operation (Line 35) represents the normal return, and the `Service_ABNresp(from, to, service)` operation (Line 46) represents the abnormal one. But beyond the architectural services, the detailed model has to include the internal behaviour of its desired iFTE architectural elements.

For including an internal behaviour model for each architectural element, we would use the B-Method inclusion mechanism (`includes`) with machine renaming [1], which in UML notation is equivalent to an attribute of a class. But because ProB model checker does not support this feature yet [29], we have used the extension mechanism (`extends`) to provide the same effect. In B-Method, an extension is a kind of inclusion, where all operations are publicly visible. The disadvantage of the extension approach is the fact that we have to adjust the B-Method machine presented in Section 6.2 for each iFTE element of the architecture, which will be internally detailed. The contents of these machines are almost the same, varying the exceptions' names, which are specific for the component, and the operations' names. The renaming of operations is needed because all operations of the iFTE internal model are inherited with public visibility by the architecture machine. To facilitate the process of renaming, we have adopted a default terminology: <`the initial`

`name of the operation>_<the name of the architectural element>`. For example, the `ClientProv_req` operation starts to be `ClientProv_req_Server` for the Server component, and `ClientProv_req_Connector` for the Connector connector.

As we just wanted to detail the internal elements of two iFTE architectural elements of our example of Figure 37 (`connector` and `serverComponent` at Line 8), the B-Method machine only extends two iFTE structure machines (Lines 11 and 12). In this example, the detailed specification of the client/server architecture presented in Figure 37 knows the internal functioning of two iFTE elements: the Server component and the Connector connector.

CSP Specification. As in the high-level architecture specification, the basic restrictions of communication are specified directly in the B-Method machine. These restrictions are defined through the provided interfaces and dependencies among components, and verified by the preconditions of the B-Method operations. In the high-level architecture specification, the CSP specification has two complementary roles: (i) to reduce the state space of the verification process; and (ii) to define specific policies for exception handling and exception propagation. Besides this, the detailed specification model is also responsible for coordinate the execution of both the architectural services (public interfaces of the architectural elements), and the internal services of the desired iFTE architectural elements. The main difficulty of modelling the architecture detailed view was to coordinate this execution sequence, considering the context switch between this two abstraction levels. To implement this, the specifier can use the CSP specification of the high-level architecture as a start point. From this model, it is necessary to consider the internal operations of all iFTE architectural elements which were detailed. In this way, as the following listing shows, the resultant CSP specification joins both the architectural specification (high-level), and many iFTE structural specification (iFTE internally). As can be seen, the main role of the CSP specification is to guarantee that the internal operations of an iFTE element can be just executed when the respective architectural element had been executed.

```
1  -- architectural flow
2  ...
3  -- external service request
4  -- REQUEST1 = ((Service_req!from!to!interface -> EXEC_CONN) [] Stop -> MAIN);;
5  -- EXEC_CONN = (ABN_CONN [] REQUEST2_CONN);;
6  REQUEST1 = ((Service_req!client!to!interface -> CONN) [] Stop -> MAIN);;
7  ...
8  -- component internally
9  CONN = StartCONN -> (CLIENT [] StopCONN -> CONN);;
10 ...
11 INTERFACE_EXCEPTION = ClientProv_ABNresp?E -> Stop1 -> ABN_CONN;;
12 ...
13 -- REQUIRED = ReqServer_req -> (EXTERNAL_NOR [] EXTERNAL_ABN);; --external
      service
14 REQUIRED = ReqServer_req -> SERVER_PROV_NOR_REQ;; --external service
15 ...
16 SERVER_PROV_NOR_REQ = StartAcMGR1 -> (CLIENT2_NOR_REQ [] StopAcMGR1 ->
      SERVER_PROV_NOR_REQ);;
```

```
17 ...
18 INTERFACE_EXCEPTION_AcMGR1 = ClientProv_ABNresp_AcMGR1?E -> Stop_AcMGR1 ->
       EXTERNAL_ABN;;
19 ...
20 NORMAL_AcMGR1_NORMAL_REPONSE = ProvNor_respAcMGR1?R1 -> ClientProv_respAcMGR1.
       R1-> StopAcMGR1 -> EXTERNAL_NOR;;
21 ...
```

Lines 4 and 6 show the difference between the requesting service before (as in the high-level architecture, Line 4) and after (in the detailed software architecture, Line 6) the iFTE internal details. Note that in second case, when the client component requests a service to the connector connector has its internal operations activated (CONN in Line 6), instead of abstract them and activate directly its returns or external requests (Line 4). From Line 9 ahead, the specification refers to the internal operations of the iFTE architectural elements. At Line 9, before use the connector, it has to be initialised (StartCONN operation), what corresponds to the Start operation in the B-Method machine of the iFTE structure.

Lines 13 and 14 of the CSP listing show how the internal behaviour of the component have been became explicit. Note that instead of provide a normal or abnormal return directly (as showed in Line 13), when occur an external request, the internal component propagates the requisition to the other architectural element (Lines 14 and 16), which in this case is the server component (SERVER_PROV_NOR_REQ). The return of the execution is presented in Lines 11, 18, and 20, that respectively corresponds to an interface exception (ABN_CONN), a failure exception (EXTERNAL_ABN), and a normal return (EXTERNAL_NOR).

6.5 Instantiation of the Models for a Specific Application

Exception handling is an inherently application-specific technique for structuring error recovery in software systems [2]. Hence, the properties of interest discussed in the previous sections have to be verified whenever the software architecture (including its iFTE architectural elements) is specified according to a specific application. Hereafter, we refer this adjust of the formal models (B-Method machines and CSP specification files) to a specific application as “instantiation of the models”.

The process of instantiating the iFTE for an application comprises two steps: first, specify the overall system architecture using iFTEs, and then specify each of the iFTEs in terms of its details. Due to the particular way that each software architecture defines the communication among its elements, the first step requires updating both the B-Method machine and CSP specification. For example, the listings which were presented in Section 6.3.4 (B-Method and CSP) corresponds to the specification of the software architecture presented in Figure 35, and had to be adjusted according to this specific application. In this way, the properties which refer to application-specific elements (Properties 5, 6, and 7 of Table 5) should be also adjusted.

Concerning the specification of software architectures, as presented in Sections 6.3.1, and 6.4.1, it is necessary to provide the following information: (i) the set of architectural elements (components and connectors); (ii) the provided interfaces of each architectural element; (iii) the dependencies among the elements (architecture configuration itself); (iv) the

set of architectural exceptions (exceptions which flow among architectural elements); (v) the list of exceptions which each interface can throw; and (vi) the exception typing hierarchy (used for handling by subsumption). These information are defined into the B-Method machine.

Alternatively, the CSP file can also specify some specific communication patterns among architectural elements. These patterns can be used for specifying handling scenarios, or for realising collaborative protocols.

On the other hand, to specify the iFTE internally it is not necessary to change the CSP specification listing presented in Section 6.2.4. It occurs because all iFTE abstractions have the same behavioural restrictions, which are inherent of the abstraction itself. In these cases, the adjust of the B-Method machine consists on provide specific information about its possible return values, as presented in Section 6.2.4. Though the internal exceptions are not visible at the architecture level, they might represent the internal details of the iFTE architectural elements, when the architect consider it a useful information (a fine-grained version of the software architecture, or an iFTE itself to the software factory).

6.6 Influences of Formal Verification on Design

The promise of formal methods is to detect errors early in the design cycle, when they are less costly to fix [26]. Had to the anticipation of some design decision, in a development process which uses some formal verification approach, the analysis phase normally finds a higher number of problems.

Besides the analysis concern, the design phase is also benefited for the formal verification. Writing the formal specification focuses the design early on important questions. For example, a distributed financial system needs to be able to communicate so securely in order to distribute keys. Since these keys should not be sent in the clear, they must themselves be encrypted. This means that the systems that participate in the interaction must already share a key. How is this key distributed? The key could be built into an "leader machine", but then how is this code distributed and loaded in a secure manner? There must be some bootstrapping mechanism to get a key loaded into an secure distributed application.

Another advantage of doing a formal model is the ability to examine the model for design errors, even before doing a formal analysis. During the formal specification of a system, the architect has to think about intrinsic details of the application. For example, restrict the way the components interact among them, specify reconfiguration scenarios to improve the system availability, thinking about the exceptions each component have to handle, according to the propagation flow of exceptions, and so on.

Errors discovered in the formal specification, probably would be propagated in following development phases. It is cheaper correct these errors in specification time. Besides that, it is possible to automatically generate some software development assets, from the formal model of a system. Examples of these asses are: source-code, test cases, dynamic diagrams, as UML sequence diagrams. This automation facilitates still more the activities related with the evolution of the system.

7 Case Study

We undertook a case study to evaluate the benefits of employing the iFTE for structuring the software architecture of an enterprise financial system with strict dependability requirements. The case study also aimed to assess the ease of verifying some properties of interest in a system that uses iFTEs as building blocks.

The case study consisted on developing part of a financial system with strict dependability requirements using the proposed approach (hereafter called simply “target system”). The system belongs to the domain of banking applications and was being developed by a medium-sized Brazilian company located in São Paulo, Brazil, which is specialised in banking automation. The subsystem employed in this case study is responsible for registering and controlling the delivery of chequebooks, account contracts, and credit limits, and consists of seven basic operations:

1. Request Chequebooks. The customer requests a chequebook (in person, by phone, through the Internet, etc.).
2. Deliver Chequebooks. The system manages the delivery of previously requested chequebooks.
3. Cancel Cheques. In cases of loss, theft, or another specific reason, the customer can cancel chequebooks.
4. Process Cheques. The processing of a cheque occurs during a deposit in cheque. These cheques are restrained and processed for future payment.
5. Cancel Accounting Contract. At any time, the customer can lose the credit of his account. In these cases, the contract of the customer must be cancelled.
6. Change Loan Credit. Depending on necessity and credit conditions, the customer can receive an additional credit limit.
7. Change Business Rules. Changes a financial rule within the bank. Such a change affects all the system services that depend on the modified rule. Economic rules are established by the Central Bank of Brazil.

Operations #1-3 can be initiated by an operator or by a customer. Operations #4-7 can only be initiated by an operator of the system. Operations Cancel Accounting Contract, Cancel Cheques, and Change Business Rules have very strict availability requirements and should be online 24/7. The first two operations must be available in order to avoid illicit use of someone’s credit, for example, in the case of theft of a chequebook. Operation Change Business Rules should not be unavailable for long. Otherwise, changes in economic rules would take a long time to be put into effect. This could result in the bank being fined for not adhering to the rules established by the Central Bank. The value of the fine is proportional to the time the service was unavailable.

The operations presented above were described as use cases. The specification of each use case includes the input information expected by the system, normal, alternative, and

exceptional scenarios, and assertions (pre- and post-conditions). Exceptional scenarios were derived from violations of assertions and alternative scenarios triggered by errors. Hereafter, due to space constraints, we focus our attention on activities related to the design of the system's software architecture. A more detailed specification of the target system is available elsewhere [16].

The case study was conducted in two phases: (i) specification of the software architecture (Section 7.1); and (ii) verification of the software architecture (Section 7.2). The first phase followed the MDCE+ method [16], an extension of the UML Components process [12] that includes activities for specifying a system's exceptional behaviour. The case study was planned by the authors. Phase one was conducted by two other developers, one of them a specialist in the domain of the application. Phase two was conducted by one of the authors.

7.1 Specification of the Software Architecture

The specified system adheres to a peer-to-peer architectural style, and has 15 components, two of them were replicated, for a total of 17 component's instances. Moreover, because we have adopted an explicit connector representation, 16 connectors were also identified. Since these architectural elements could potentially raise exceptions, the system was structured using iFTEs. Figure 38 shows a simplified view of the architecture of the target system. The architecture adheres to a relaxed layered style where two transversal layers, `Communication` and `Utility`, are accessed by all other the "regular" layers. Both layers are actually part of an infrastructure that is reused across many systems developed within the company. The `Data` layer provides access to the database. Components in the `Utility` layer provide fine-grained services, for example, statistical functions. The `Communication` layer implements communication bridges with other systems, including the legacies ones. The `System` and `Business` layers implement the application-dependent and application-independent business rules, respectively. Components in the `Business` layer can be reused across different applications from the same domain, since they are application-independent. Finally, the `Boundary` layer is the only interaction point between the system and its users.

The components in each layer and their provided and required interfaces were identified and specified following the MDCE+ method. This specification contains the list of exceptions which each component raises and the behaviour of the associated exception handlers. The specified system has 15 components, two of them were replicated, for a total of 17 component instances. Moreover, because we have adopted an explicit connector representation, 16 connectors were also identified. Since these architectural elements potentially raise and catch exceptions, the system was structured using iFTEs.

Because all architectural elements could potentially raise or catch exceptions, all the 33 iFTE architectural elements were structured as iFTEs. Figure 39 shows part of the 33 iFTE elements of the overall system. Two instances of `EconomicRules_Ctr` are responsible for receiving and processing the requests for changing economic rules. `Accounts_Ops` controls the execution of accounting use cases' scenarios. For the sake of simplicity, in the rest of the section, the explanation focuses on these components.

The system can fail in a variety of ways and for a number of different reasons. For example, the `Account_Ops` component alone signals nine different types of exceptions. In

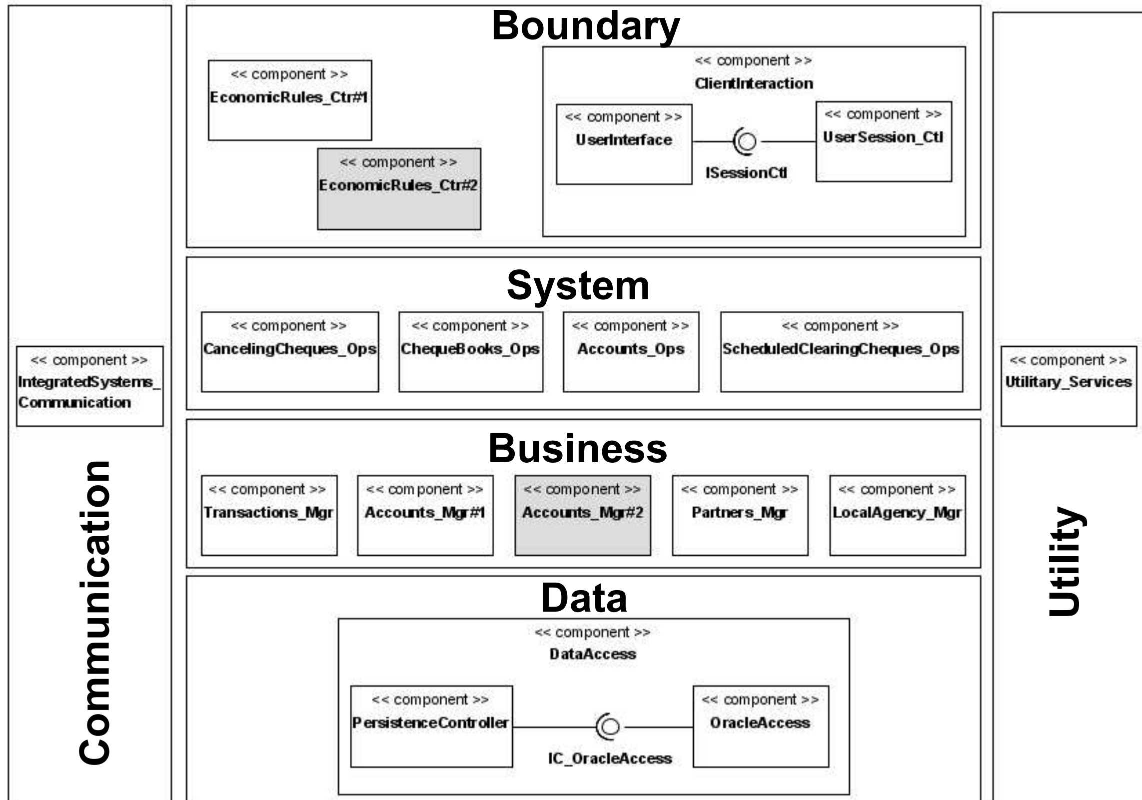

Figure 38: Layered architecture of the target system

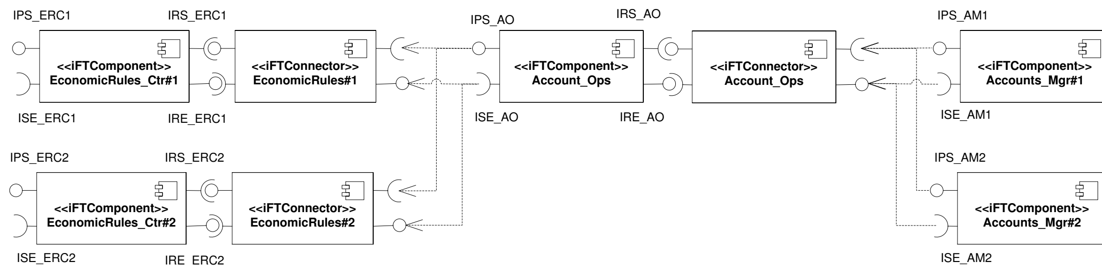

Figure 39: Architecture Detailed View (with iFTE)

total, there are more than 20 types of exceptions that flow between architectural components. This large number of exceptions stems from a development policy adopted by the organisation where the case study was conducted. According to this policy, even very similar errors and situations that are not normally treated as errors generate new exception types. Table 6 shows the exceptions which `Account_Ops` may signal.

Table 6: Some exceptions signaled by the `Account_Ops` component.

#	Exception	Description
1	`InvalidInputDataException`	An Interface exception that encapsulates all exceptions related to the provision of invalid fields. For example, an agency number does not belong to any valid agency.
2	`BusinessRulesViolationException`	A Failure exception that encapsulates all exceptions related to the violation of some business rule. For example, try a transaction in an account, which has its contract already been canceled.
3	`TransactionNotFinishedException`	A Generic exception that encapsulates all exceptions related to some generic failure. For example, the occurrence of an unexpected failure exception, as a `NullPointerException` in Java.
4	`CommunicationFailureException`	An exception which is raised when some architectural element identify a failure in the communication link.
5	`InsufficientFundsException`	A business-specific exception, which is raised when the system tries to execute a transaction which needs funds and it is not available.
6	`AuthenticationFailureException`	Exception which is raised when occurs a failure during the autentication process of one user or agency.
7	`InsufficientPrivilegesException`	Exception which is raised when a valid user (which have been successfully authenticated) tries to execute a transaction which he has not the privileges.
8	`InsufficientDataException`	Exception which is threw when a transaction needs the user actualise its data register.
9	`TransactionAlreadyExecutedException`	Exception which is threw when a transaction is executed twice or more times in a predefined timeout.

7.2 Verification of the Software Architecture

Although exception handling is a well-known mechanism for structuring forward error recovery in software systems, the actual exception handling performed within an application is inherently application-specific [3]. Since exception handling is an application-specific technique, the properties of the software architecture have to be verified when the architectural abstraction is instantiated for a particular application. As presented in Section 6.5, this instantiation consists on representing the particularities of a specific application into the formal models presented in Section 6.

During the instantiation process, which includes the instantiation of the properties, we have also specified some application-specific properties. For example, Property 7 of the internal propagation of the iFTE (Table 3) was specified as follows: *When an iFTE element fails, it is not possible to execute other services before stop and restart the failed element.* This was verified through the following property: After a ``ClientProv_ABNresp'' event it is only possible to execute the ``Stop'' operation. Besides that, other application-specific properties were also specified for this application, for example analyse the possibility to occur application-specific scenarios (Property 6 of the architecture configuration, Table 5).

The verification process of the software architecture, which is presented here, consists of two activities: (i) verification of each iFTE element of the software architecture (iFTComponents and iFTConnectors); and (ii) verification of the software architecture in terms of exception propagation amongst iFTE elements. After these two activities, it is possible to analyse the results of the case study. This section describes how the verification was conducted, and the next section discusses the experience obtained with it.

After instantiating the iFTE architectural abstraction into 31 different architectural elements (33 including the redundant copies), we have conducted the verification of their basic and internal properties, which were presented in Sections 6.1.2, and 6.2.3.

Aiming to increase gradually the complexity of the formal models, we have also verified the software architecture in two steps: (i) verification of its high-level view, abstracting from the internal details of each iFTE element; and (ii) verification of its development view, considering the internal details of the iFTE elements. In both steps, we have verified the same set properties, as presented in Sections 6.3.3 and 6.4.3 (Tables 4, and 5).

Besides the general properties regarding the verification of software architectures, we have also analysed the viability of some specific scenarios that were considered critical for this application. As an example, analysing the part of the system presented in Figure 39, the architects identified the following scenario, to improve the system availability through the use of redundant instances: *when the* `AccountOps_Business_conn` *connector requests a service to the* `Accounts_Mgr#1` *component and this component fails, the connector has to request an equivalent service for the* `Accounts_Mgr#2` *component, before throwing an exception.*

The following subsections describe how verification was conducted. Section 7.3 discusses the results of the case study.

7.2.1 Verification of the iFTE Abstraction for each Architectural Element

First, we instantiated the iFTE abstraction model (Section 6.1) to each element of the architecture. To carry through instantiating the iFTE model, we need to define the elements of two sets, and to initialise two variables. The sets are: `AbnormalReturns`, and `NormalReturns`, and the variables are: `maskableExceptions`, and `externalExceptions`. Lines 4 and 5 of the following code shows how we have defined the elements of the `NormalReturns` and `AbnormalReturns` sets for the `Account_Ops` iFTE component. Lines 9 and 11 shows how this iFTE component can deal with the defined exceptions. Analysing this specification, although `InsufficientFundsException` and `authenticationFailureException` are defined in the list of abnormal types understood by the iFTE (Line 5), these exceptions are not considered external (not in `externalExceptions` in Line 11) and cannot be propagated.

```
1 MACHINE account_Ops
2 SETS
3 ...
4 NormalReturns = {normalReturn};
5 AbnormalReturns = {invalidInputDataException, businessRulesViolationException,
      transactionNotFinishedException, communicationFailureException,
    insufficientFundsException, authenticationFailureException,
```

```
    insufficientPrivilegesException , insufficientDataException ,
    transactionAlreadyExecutedException , invalidAccountException ,
    duplicatedAccountException}
6 ...
7 INITIALISATION
8 ...
9        maskableExceptions := {insufficientFundsException ,
            insufficientDataException , transactionAlreadyExecutedException ,
            invalidAccountException , duplicatedAccountException} ||
10 ...
11       externalExceptions := {invalidInputDataException ,
            businessRulesViolationException , transactionNotFinishedException ,
            communicationFailureException , insufficientFundsException ,
            authenticationFailureException , insufficientPrivilegesException ,
            insufficientDataException , transactionAlreadyExecutedException} ||
12 ...
```

After its instantiation, the model was verified using ProB model checker and a specific tool for analysing scenarios (see Section 6.1.2). It took thirty seconds on average for each component. During the verification process of the individual components, we identified a couple of spelling errors in instantiations, but after correct them, all the properties presented in Section 6.1.2 were successfully verified.

7.2.2 Verification of the iFTE Structure for each Architectural Element

After verifying the iFTE abstraction, we have performed a detailed verification of the iFTE architectural elements. This verification could be discarded, but in this case study we have decided to be as critical as possible. So, we first instantiated the iFTE structural model to each element of the architecture. To instantiate the structural model of the iFTE it is necessary exactly the same information, which was specified in the iFTE abstraction models: to define the elements of the `AbnormalReturns` and `NormalReturns` sets, and to initialise the `maskableExceptions` and `externalExceptions` variables. Thus, the instantiation process of the structural models consisted on a "copy and paste" procedure from the abstract model of each iFTE. The following code presents all the changes, which are needed to instantiate a structural model of an iFTE. As can be seen, the difference between the abstract model and the internal structure model of an iFTE is the operations and properties of each one, which is fix and do not need to be instantiated.

```
1 MACHINE account_Ops
2 SETS
3 ...
4 NormalReturns = {normalReturn};
5 AbnormalReturns = {invalidInputDataException , businessRulesViolationException ,
     transactionNotFinishedException , communicationFailureException ,
    insufficientFundsException , authenticationFailureException ,
    insufficientPrivilegesException , insufficientDataException ,
    transactionAlreadyExecutedException , invalidAccountException ,
    duplicatedAccountException}
6 ...
7 INITIALISATION
```

```
8 ...
9       maskableExceptions := {insufficientFundsException ,
           insufficientDataException , transactionAlreadyExecutedException ,
           invalidAccountException , duplicatedAccountException} ||
10 ...
11      externalExceptions := {invalidInputDataException ,
           businessRulesViolationException , transactionNotFinishedException ,
           communicationFailureException , insufficientFundsException ,
           authenticationFailureException , insufficientPrivilegesException ,
           insufficientDataException , transactionAlreadyExecutedException} ||
12 ...
```

After its instantiation, the model was verified using ProB model checker and a specific tool for analysing scenarios (see Section 6.1.2). It took five minutes on average for each component. The verification of each individual architectural component showed that the abstract model and the internal structure model of the iFTE are consistent, because all properties presented in Section 6.2.3 were successfully verified, after fixing some writing errors that happened during the manual instantiation of the model.

7.2.3 Verification of the High-Level Model of the Architecture

As presented in Section 6.3.1, to instantiate the high-level model of the software architecture, it is necessary to represent the following information, which can vary from an application to another one: (i) architectural elements (components and connectors with their interfaces); (ii) the way its elements are disposed together (the architecture configuration); (iii) possible abnormal returns of each interface; and (iv) the exception type hierarchy.

As can be seen in the following code, the architectural elements and their interfaces are represented through the `ArchitecturalElements` and `Interfaces` sets (Lines 6, and 7, respectively). But the association among a component and its provided interface is defined in the `componentInterfaces` variable, which is initialised in Line 13. The configuration of the architecture (the way they are disposed together) is defined through the `dependencies` variable (Line 11). Finally, the `AbnormalReturns` set (Line 5) defines all the exceptions which flow in the architecture; and the type hierarchy of these exceptions is defined through the `exceptionHierarchy` variable, initialised in Line 16; and Line 17 defines that the architect wants to handle exceptions by subsumption. The `maskableExceptions` variable (Line 15) defines for each architectural element, which exceptions it can mask. For example, Line 15 states that the `AccountOps_Business_conn` connector can mas two exceptions: `transactionNotFinishedException`, and `communicationFailureException`. They can be masked through the two redundant components (`Accounts_Mgr#1` and `Accounts_Mgr#2`) which the connector interacts with.

```
1 MACHINE systemConfiguration
2 SETS
3     ...
4     NormalReturns = {norReturn};
5     AbnormalReturns = {... , invalidAgencyException , invalidAccountException ,
         alreadyCanceledException , transactionNotFinishedException ,
         communicationFailureException , ...};
```

```
6     ArchitecturalElements = {... , economicRules_Ctr_1 , economicRules_Ctr_2 ,
          accounts_Ops , accounts_Mgr_1 , accounts_Mgr_2 ,
          economicRules_System_conn1 , economicRules_System_conn2 ,
          accountOps_Business_conn , ...};
7     Interfaces = {IC_EconomicRules , IC_ER_SystemReq , IC_AccountContract ,
          IC_AdditionalCredit , IC_A_BusinessReq , IC_Account_Mgt , ...}
8 ...
9 INITIALISATION
10    ...
11    dependencies := {economicRules_Ctr_1 |-> {economicRules_System_conn},
          accountOps_Business_conn |-> {transactions_Mgr , accounts_Mgr_1 ,
          accounts_Mgr_2 , partners_Mgr , localAgency_Mgr}, ...} ||
12    ...
13    componentInterfaces := {economicRules_Ctr_1 |-> {IC_EconomicRules},
          accounts_Ops |-> {IC_AdditionalCredit , IC_AccountContract}, ...} ||
14    interfaceExceptions := {invalidAgencyException , invalidAccountException ,
          alreadyCanceledException , ...} ||
15    maskableExceptions := {accountOps_Business_conn |-> {
          transactionNotFinishedException , communicationFailureException}, ...}
          ||
16    exceptionHierarchy := { invalidAgencyException |-> {interfaceException},
          invalidAccountException |-> {interfaceException},
          alreadyCanceledException |-> {failureException}, ...} ||
17    handlingBySubsumption := TRUE
18 ...
```

After instantiation, the model was verified using ProB model checker and a specific tool for analysing scenarios (see Section 6.1.2). It took on average 40 minutes to verify all the system architecture. The little time is justifiable because in this model we abstract all internal details of the architectural elements. Beyond the properties presented in Section 6.3, that were successfully verified, we have also analysed the possibility to occur one application-specific scenario: *if the* `Accounts_Mgr#1` *component fails, have to be possible for the* `AccountOps_Business_conn` *connector request the same service for the redundant component* `Accounts_Mgr#2`. The model checker presented a sequence of events where this scenario occurs.

7.2.4 Verification of the Detailed Software Architecture

For the sake of simplicity and to speed the verification process, we have defined various detailed models of the software architecture, each one with just three detailed architectural elements. With these three detailed elements, it was possible to verify in a lower-level way some important interaction scenarios, for example the reconfiguration scenario, which was just mentioned. In this way, the three detailed architectural elements were: the `Accounts_Mgr#1` component, the `AccountOps_Business_conn` connector, and the `Accounts_Mgr#2` component. Although it does not represent all the system architecture in just one model, detailing three architectural elements had been sufficient to express all critical propagation scenarios.

For expressing the characteristics of the two first models, the detailed architecture model is the most laborious to instantiate. Due to its necessity of synchronisation between the architectural operations and the internal iFTE operations, we proceed the instantiation in

three steps: (i) instantiate each detailed iFTE element; (ii) instantiate the architecture in a high-level way; and (iii) join both models and synchronise them. The first two steps can be proceed as presented before (Sections 7.2.2 abd 7.2.3), except by the joining of both models in a single one. Part of this merged B-Method machine is following presented. The three architectural elements which were detailed in this scenario were the `EconomicRules_Ctr#1`, `EconomicRules1_System_conn`, and `Accounts_Ops` architectural elements (Line 6).

```
1 MACHINE iFTEPropagation
2 ...
3    NormalReturns = {norReturn};
4    AbnormalReturns = {..., invalidAgencyException, invalidAccountException,
         alreadyCanceledException, ...};
5    ArchitecturalElements = {..., economicRules_Ctr_1, economicRules_Ctr_2,
         accounts_Ops, accounts_Mgr_1, accounts_Mgr_2,
         economicRules_System_conn1, economicRules_System_conn2,
         accountOps_Business_conn, ...};
6    DetailedArchitecturalElements = {economicRules_Ctr_1,
         economicRules_System_conn1, accounts_Ops};
7    Interfaces = {IC_EconomicRules, IC_ER_SystemReq, IC_AccountContract,
         IC_AdditionalCredit, IC_A_BusinessReq, IC_Account_Mgt, ...}
8 ...
9 EXTENDS
10   economicRulesCtr1,
11   economicRules1Systemconn,
12   accountsOps
13 ...
```

After the inclusion of the internal models in Lines 10 and 12, all operations of `economicRulesCtr1` `economicRules1Systemconn` and `accountsOps` M-Method machines are visible in the high-level software architecture. In order to prevent the specification of invalid scenarios, it was necessary to organise the order that these added operations can execute. Operations that represent the internal execution of an element have just to be available after the respective architectural element has received a service request. This restrictions of communication is specified in CSP process algebra.

As presented in Section 6.4, the CSP complementary file is composed by both high-level architecture model and many iFTE internal models. Consequently, to instantiate it, it is necessary to define the join point between each service request at the architectural level and the internal behaviour of the respective iFTE element. Following, we present a CSP specification which details three components of the high-level software architecture: `EconomicRules_Ctr#1` component, the `EconomicRules1_System_conn` connector (`conn`), and the `Accounts_Ops` component.

```
1 -- architectural flow
2 ...
3 -- external service request
4 -- REQUEST1 = ((Service_req!from!to!interface -> EXEC_CONN) [] Stop -> MAIN);;
5 -- EXEC_CONN = (ABN_CONN [] REQUEST2_CONN);;
6 REQUEST1 = ((Service_req!economicRules_Ctr_1!to!interface -> CONN) [] Stop ->
     MAIN);;
7 ...
8 -- component internally
```

```
9 CONN = StartCONN -> (ECONOMIC_RULES_CTR_1 [] StopCONN -> CONN);;
10 ...
11 INTERFACE_EXCEPTION = EconomicRulesCtr1Prov_ABNresp?E -> Stop1 -> ABN_CONN;;
12 ...
13 REQUIRED = ReqServer_req -> ACOPS_PROV_NOR_REQ;; --external service
14 ...
15 ACOPS_PROV_NOR_REQ = StartAcMGR1 -> (ECONOMIC_RULES_CTR_1_NOR_REQ []
     StopAcMGR1 -> ACOPS_PROV_NOR_REQ);;
16 ...
17 INTERFACE_EXCEPTION_AcMGR1 = EconomicRulesCtr1Prov_ABNresp_AcMGR1?E ->
     Stop_AcMGR1 -> EXTERNAL_ABN;;
18 ...
19 NORMAL_AcMGR1_NORMAL_REPONSE = ProvNor_respAcMGR1?R1 ->
     EconomicRulesCtr1Prov_respAcMGR1.R1-> StopAcMGR1 -> EXTERNAL_NOR;;
20 ...
```

After its instantiation, the model was verified using ProB model checker and a specific tool for analysing scenarios (see Section 6.1.2). It was spent on average 50 minutes to verify each version of the detailed software architecture. We have verified the same properties of the architectural flow, including the reconfiguration scenario explained in Section 7.2.3.

Related with the properties that were verified, after fixing some misspellings errors, all of them were successfully satisfied. The most serious error which were found in the model, was a violation of the Property 4 of Table 5 (Section 6.3.3), which is related with exception propagation and prevents that an exception are forgotten by the architect. This omission the exception handling was caused by the following fault during the instantiation of the model: *the* EconomicRules_Ctr#1 *component raised the* `DuplicatedAccountException` *exception to the* AccountOps_Business_conn *connector, which could not mask it and propagated to the* Accounting_Ops *component. But although this exception is non-maskable (the* Accounting_Ops *component cannot return normally after have received it), the architect forgot to initialise the* `externalExceptions` *variable, which indicate the exceptions that could be thrown by the* Accounting_Ops *component. Then, the* Accounting_Ops *component neither could return normally (because it has received a non-maskable exception), nor could return abnormally (no declared exceptions).* Because this, the Property 1 of Table 4 (Section 6.3.3) was also be violated, because the system had been in deadlock. Besides that, during the verification of these models, we have traced some scenarios of exception propagation. The goal of these tracings was initially understand the architecture, but after, this feature has helped us to trace exceptions amongst different architectural elements.

7.3 Case Study Evaluation

For specifying and verifying the architecture of the system, including identifying components and interfaces, was necessary an effort of about 100 developer/hour.

Overall, this case study has shown how exceptions are handled and propagated on the proposed architectural approach by enforcing a clear separation between the application, and the infrastructure that might affect the application, in our case, exceptions and exception handling that are employed for tolerating faults.

During the verification of the software architecture, it was clear the iFTE helped the

architect to distinguish clearly the role of each flow of the architecture (normal and abnormal). This clear distinction favoured the specification in parallel of both architectural views, which have been made by distinct developers.

Besides that, the verification results provide valuable information about the system properties, and help to correct the specification of the system architecture in earlier stages of the software development. Although most of these problems are simple to correct, failure to address them can result in problems that are harder to correct in later phases of development.

The results do not demonstrate the universal usefulness of iFTE in the construction of software systems with strict dependability requirements. They do show, however, that the iFTE approach is useful in some cases, and justify further studies that can provide more substantial evidence, mainly considering components executing concurrently.

Concerning the developer effort for instantiating the models, we consider that the instantiation of the two iFTE models and the high-level architectural model were easy and fast. However, the effort necessary to instantiate the detailed model of the software architecture was great and error prone, mainly the definition of the CSP process algebra. But it is important to stress that all the tasks for instantiating models can be done automatically, by means of a case tool, which is still under construction.

To analyse the scalability of the proposed solution, we have adopted two approaches: (i) to compare the proposed solution with the Aereal framework: we compared the results after the same specification had been verified in both solutions; and (ii) to compare the difference between the number of states in a “pure B” and a “B + CSP” solution: we verified in ProB the B specified machines as with as without the support of the CSP restrictions.

After generate the Alloy [26] specification with Aereal, we have tried to simulate all the software architecture in the Alloy Analyser tool. But to our surprise, the application could not reach the end of the verification, because ran out of memory in a computer with 1GB of RAM memory. The main reason of this high number of possible states was because Alloy is based on mathematical set theory, similar to B-Method. Although the high amount of time (approximately one hour) needed to verify the system in ProB (with B and CSP), the case tool successful completed its goal.

Finally, analysing the case study results in the perspective of reducing the number of faults added to the system during its development, the iFTE has improved the system dependability in four complementary points of view:

1. **Complexity management**. The iFTE is a simple model with well-defined interfaces. So, it provides a way to abstract the internal complexity of the idealised fault-tolerant component;

2. **Separation of concerns**. For being based on the concept of the idealised fault-tolerant component, presented in Section 2.3, and for keep the Normal and Abnormal components separately, the iFTE model provides a semantic and physical separation between the normal and abnormal concerns. This separation improves the software understandability, since the software developer can concentrate efforts on understanding and developing each concern separately;

3. **Component adaptation**. When reusing components, the developer can add new faults during the component adaptation in the new context of use. Therefore, the iFTE has elements with the specific role to deal with architectural mismatches: the Provided and Required connectors.

4. **Architectural verification**. As pointed out by many authors [5, 14, 35], the architecture of a software system has a strong impact on the ability of the system to meet its intended quality requirements, for example, security, reliability, availability, and performance. Thus, if a system should be reliable and exception handling is one of the mechanisms that will be used to achieve this goal, it may be valuable to consider exception handling-related issues during architectural design [10]. The iFTE model support the software architect during the rigorous architectural design, providing specific interfaces to signal exceptions at the architectural level. This explicit separation prevents the flow of exceptions is totally neglected in the beginning of the software life cycle. Moreover, worry about the abnormal behaviour of the system in a high-level of abstraction helps to specify recovery mechanisms that involve more than one architectural component in a collaborative way [20].

8 Conclusions and Future Work

This paper has presented an architectural abstraction for structuring fault-tolerant systems, which is able to handle architectural mismatches in the presence of software component failures. The idealised fault-tolerant architectural element (iFTE), which is based on the peer-to-peer architectural style, relies on exception handling for detecting and eliminating erroneous states from the system. In addition to formally specifying and verifying the architectural abstraction using the B-Method and CSP, we have also presented how exception propagation can be analysed in software architectures that are based on the iFTE. Although the case study used for evaluating the overall approach was a snapshot of a real system, we were able to demonstrate nevertheless that compared with other similar approaches, the iFTE provides an appropriate abstraction for modelling and analysing fault-tolerant software architectures. However, since the exception handling mechanism is application dependent, the failure analysis of the overall system is restricted by the coverage of the architectural properties associated with the iFTE. Although these properties provide the basis to identify faults that might be caused by the instantiation of the iFTE, nonetheless a thorough failure analysis is necessary for localising further faults that are not particularly related to the iFTE.

Developers of systems with strict dependability requirements benefit from using the proposed solution in the following ways: (i) by better documenting their decisions about exceptions that flow amongst architectural elements; (ii) by making explicit their assumptions about the policy of exception handling that will be used during system implementation; and (iii) by checking structural constraints imposed by architectural styles. Moreover, iFTE explicitly separates the normal and abnormal behaviour, parallelling the specify and the development of both parts. This feature promotes better understandability and maintainability because concerns do not get cluttered in a single development unit.

Since a limitation of the proposed solution is the lack of concurrency within the architectural abstraction, as future work, we intend to deal with concurrent systems, which have more complex exception handling strategies and resolution of exceptions. Tool support for analysing the propagation of exceptions is still central to our work, for that, we are developing support for automatically extracting a XMI specification from the software architecture. The additional information about exception flow and handling policies is being specified through meta tags, a standard extension point of the XMI scheme. With this tool, we expect to ease the verification task, which is yet mostly conducted by software developers.

The second area of future work is further evaluation. Supported by tools, we would like to gain more substantial experience in specifying the exceptional behaviour of dependable critical systems with complex architectural styles. Furthermore, we still have not attempted to evaluate the usefulness of the proposed approach in the context of software evolution. We believe it can be a valuable tool to detect inconsistencies and conflicts introduced by software architecture evolution in a system's abnormal behaviour.

9 Acknowledgments

Patrick Brito is supported by Fapesp/Brazil, under grant 06/02116-2. Fernando Castor Filho is supported by Fapesp/Brazil, under grant 04/10663-8. Cecília Rubira is partially supported by CNPq/Brazil under grant 351592/97-0, and by Fapesp/Brazil, under grant 2004/10663-8. Patrick Brito, Fernando Castor Filho, and Cecília Rubira are partially supported by FINEP/Brazil under grant 1843/04 of CompGov, which is a project for a Shared Library of Components for e-Government.

References

[1] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Sorensen. The b-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development (VDM '91) - Volume 2*, pages 398–405, London, UK, 1991. Springer-Verlag.

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[3] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[5] Len Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[6] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *In Lecture Notes on Concurrency and Petri Nets*, LNCS 3095. Springer-Verlag, 2004.

[7] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, School of Computing, Queen's University, March 2004.

[8] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[9] Michael J. Butler and Michael Leuschel. Combining csp and b for specification and property verification. In *Proceeding of Formal Methods*, LNCS 3582, pages 221–236. Springer, 2005.

[10] Fernando Castor Filho, Patrick Henrique da Silva Brito, and Cecília Mary F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, October 2006.

[11] Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília M. F. Rubira. An architectural-level exception-handling system for component-based applications. In *Proceedings of the 1st Latin-American Symposium on Dependable Computing*, LNCS 2847, pages 321–340, 2003.

[12] John Chessman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. 2000.

[13] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[14] Paul C. Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. Addison-Wesley, 2003.

[15] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97. Blackwell, 1989.

[16] Patrick Henrique da Silva Brito, Camila Ribeiro Rocha, Fernando Castor Filho, Eliane Martins, and Cecília M. F. Rubira. A method for modeling and testing exceptions in component-based software development. In *Proceedings of the 2nd Latin American Symposium on Dependable Computing*, LNCS 3747, pages 61–79, 2005.

[17] P. A. de Castro Guerra, Cecilia Mary Fischer Rubira, and R. de Lemos. A fault-tolerant software architecture for component-based systems. In *Architecting Dependable Systems. Lecture Notes in Computer Science*, LNCS 2677, pages 129–149. Springer, Berlin, Germany, 2003.

[18] R. de Lemos and A. Romanovsky. Coordinated atomic actions in modelling objects cooperation. In *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 152, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, 2006.

[20] Rogério de Lemos and A. Romanovsky. Exception handling in a cooperative object-oriented approach. In *Proc. of the 2nd IEEE ISORC'99*, May 1999.

[21] Rogerio de Lemos and Alexander Romanovsky. Exception handling in the software lifecycle. *International Journal of Computer Science and Engineering*, 16(2):167–181, 2001.

[22] C. Gacek and R. de Lemos. Architectural description of dependable software systems. In *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, pages 127–142. Springer, London, UK, 2006.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] John B. Goodenough. Exceptional handling: Issues and a proposed notation. *CACM*, 18(12), 1975.

[25] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[26] Brant Hashii. Lessons learned using alloy to formally specify mls-pca trusted security architecture. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 86–95, New York, NY, USA, 2004. ACM Press.

[27] V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences.*, 2001.

[28] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[29] M. Leuschel and Michael J. Butler. Prob: A model checker for b. In *Proceedings of FME'2003*, LNCS 2805, pages 855–874. Springer-Verlag, Pisa, Italy, 2004.

[30] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):221–232, 1975.

[31] D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In *Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, July 2003.

[32] Cecília Mary F. Rubira, Rogério de Lemos, Gisele Ferreira, and Fernando Castor Filho. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.

[33] Aaron Shui, Sadaf Mustafiz, and Jorg Kienzle. Exception-aware requirements elicitation with use cases. In *Recent Advances in Exception Handling Techniques*, LNCS 4119, pages 221–242. Springer-Verlag, 2006. To appear.

[34] Kevin Simons and Judith Stafford. Cmeh: Container-managed exception handling for increased assembly robustness. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, LNCS 3054, pages 122–129, May 2004.

[35] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[36] R. N. Taylor, N. Medvidovic, K.M. Anderson, Jr. E. J. Whitehead, and J.E. Robbins. A component- and message- based architectural style for GUI software. In *Proceedings of the 17th International Conference on Software Engineering*, pages 295–304, April 1995.

[37] G.J. Vecellio, M.M. Thomas, and R.M. Sanders. Container services for high confidence software. In *Proceedings of the 7th ECOOP Workshop on Component-Oriented Programming*, June 2002.

[38] Westley Weimer and George Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of OOPSLA'2004*, pages 419–433, Vancouver, Canada, October 2004.

A iFTE Abstraction Model

A.1 B-Method Machine

```
1  MACHINE abstractModel
2
3  /* ================================ */
4  SETS
5        BOperations = {client_ifte_req , client_ifte_nor , client_ifte_abn ,
6                       client_server_req , client_server_nor , client_server_abn ,
7                       ifte_client_req , ifte_client_nor , ifte_client_abn ,
8                       ifte_server_req , ifte_server_nor , ifte_server_abn ,
9                       server_client_req , server_client_nor , server_client_abn ,
10                      server_ifte_req , server_ifte_nor , server_ifte_abn };
11       Components = {none , client , ifte , server };
12       Calls = {start , client_ifte , client_server , ifte_client , ifte_server ,
             server_client , server_ifte };
13       NormalReturns = {norReturn};
14       AbnormalReturns = {noException , abnReturn1 , abnReturn2}
15
16
17
18
19 /* ================================ */
20 VARIABLES
21       callList ,                /*control variable*/
22       currentComponent ,        /*control variable*/
23       activedRequests ,         /*control variable*/
24 /*       sequenceHistory ,         /*control variable*/*/
25       lastABNreturn ,           /*control variable*/
26       unrecoverable ,           /*control variable*/
27       maskableExceptions ,
28       externalExceptions
29
30
31
32
33 /* ================================ */
34 /*DEFINITIONS
35    GOAL == ( ... ) */
36
37
38
39 /* ================================ */
40 INVARIANT
41               callList : Components --> seq(Components) & /*for each
                     component , this variable stores the list of its callers (in
                      chronological order)*/
42               currentComponent : Components & /*the component that is
                     actually working*/
43               activedRequests : POW(Calls) & /*a set of calls that were not
                     finished*/
```

```
44                sequenceHistory : seq(BOperations) & /*store the sequence of
                      operations, in order to provide traceability*/
45                lastABNreturn : AbnormalReturns & /*indicate which exception
                      have influenced the iFTE */
46                unrecoverable : BOOL &/*indicates if a component can recover
                      itself of a failure. In other words, if it have received an
                       exception which cannot be masked*/
47                maskableExceptions : POW(AbnormalReturns) & /*exceptions which
                      can be handled*/
48                externalExceptions : POW(AbnormalReturns) /*external exceptions
                      */
49
50
51
52 /* ================================ */
53 INITIALISATION
54        callList := {none |-> <>, client |-> <>, ifte |-> <>, server |-> <>} ||
55        currentComponent := none ||
56        activedRequests := {} ||
57        sequenceHistory := <> ||
58        lastABNreturn := noException ||
59        unrecoverable := FALSE ||
60        maskableExceptions := {abnReturn1} ||
61        externalExceptions := {abnReturn1, abnReturn2}
62
63
64
65 /* ================================ */
66 OPERATIONS
67
68 /*0*/
69 Start =
70        PRE
71                none = currentComponent & card(activedRequests)=0
72        THEN
73                currentComponent := client ||
74                activedRequests := activedRequests \/ {start} ||
75                sequenceHistory := <> ||
76                lastABNreturn := noException ||
77                unrecoverable := FALSE
78        END;
79
80
81
82 /*00*/
83 Stop =
84        PRE
85                client = currentComponent & start : activedRequests & card(
                      activedRequests)=1
86        THEN
87                activedRequests := activedRequests - {start} ||
88                currentComponent := none ||
89                sequenceHistory := <> ||
90                skip
```

```
91        END;
92
93
94
95
96 /*10*/
97 ClientIFTE_req =
98        PRE
99                client = currentComponent  &
100               client_ifte /: activedRequests
101       THEN
102               currentComponent := ifte ||
103               activedRequests := activedRequests \/ {client_ifte} ||
104               callList(ifte) := callList(ifte) <- client ||
105               sequenceHistory := [client_ifte_req]
106       END;
107
108
109 resp <-- ClientIFTE_resp =
110       PRE
111               size(callList(ifte))>0 &
112               last(callList(ifte)) = client  &
113               client_ifte : activedRequests  &
114               ifte = currentComponent &
115               ((lastABNreturn = noException) or (lastABNreturn :
                      maskableExceptions)) &
116               unrecoverable = FALSE
117
118       THEN
119               activedRequests := activedRequests - {client_ifte} ||
120               currentComponent := client ||
121               callList(ifte) := front(callList(ifte)) ||
122               resp := norReturn ||
123               sequenceHistory := sequenceHistory <- client_ifte_nor
124       END;
125
126
127 resp <-- ClientIFTE_ABNresp =
128       PRE
129               size(callList(ifte))>0 &
130               last(callList(ifte)) = client  &
131               client_ifte : activedRequests  &
132               ifte = currentComponent
133
134       THEN
135               activedRequests := activedRequests - {client_ifte} ||
136               currentComponent := client ||
137               callList(ifte) := front(callList(ifte)) ||
138               sequenceHistory := sequenceHistory <- client_ifte_abn ||
139               ANY val WHERE val : AbnormalReturns & (val : externalExceptions
                      ) /*& val /= noException*/
140                   THEN
141                       IF val /: maskableExceptions
142                           THEN
```

```
143                             unrecoverable := TRUE
144                         END ||
145                         resp := val ||
146                         lastABNreturn := val
147                     END
148         END;
149
150
151
152
153 /*20*/
154 ClientServer_req =
155         PRE
156                 client = currentComponent  &
157                 client_server /: activedRequests
158         THEN
159                 currentComponent := server ||
160                 activedRequests := activedRequests \/ {client_server} ||
161                 callList(server) := callList(server) <- client ||
162                 sequenceHistory := [client_server_req]
163         END;
164
165
166 resp <-- ClientServer_resp =
167         PRE
168                 size(callList(server))>0 &
169                 last(callList(server)) = client  &
170                 client_server : activedRequests  &
171                 server = currentComponent
172
173         THEN
174                 activedRequests := activedRequests - {client_server} ||
175                 currentComponent := client ||
176                 callList(server) := front(callList(server)) ||
177                 resp := norReturn ||
178                 sequenceHistory := sequenceHistory <- client_server_nor
179         END;
180
181
182 resp <-- ClientServer_ABNresp =
183         PRE
184                 size(callList(server))>0 &
185                 last(callList(server)) = client  &
186                 client_server : activedRequests  &
187                 server = currentComponent
188
189         THEN
190                 activedRequests := activedRequests - {client_server} ||
191                 currentComponent := client ||
192                 callList(server) := front(callList(server)) ||
193                 sequenceHistory := sequenceHistory <- client_server_abn ||
194                 ANY val WHERE val : AbnormalReturns & (val : externalExceptions
                        ) /*& val /= noException*/
195                     THEN
```

```
196                         IF val /: maskableExceptions
197                            THEN
198                               unrecoverable := TRUE
199                         END ||
200                         resp := val ||
201                         lastABNreturn := val
202                      END
203        END;
204
205
206
207
208 /*30*/
209 IFTEClient_req =
210        PRE
211                ifte = currentComponent  &
212                ifte_client /: activedRequests
213        THEN
214                currentComponent := client ||
215                activedRequests := activedRequests \/ {ifte_client} ||
216                callList(client) := callList(client) <- ifte ||
217                sequenceHistory := [ifte_client_req]
218        END;
219
220
221 resp <-- IFTEClient_resp =
222        PRE
223                size(callList(client))>0 &
224                last(callList(client)) = ifte  &
225                ifte_client : activedRequests  &
226                client = currentComponent
227
228        THEN
229                activedRequests := activedRequests - {ifte_client} ||
230                currentComponent := ifte ||
231                callList(client) := front(callList(client)) ||
232                resp := norReturn ||
233                sequenceHistory := sequenceHistory <- ifte_client_nor
234        END;
235
236
237 resp <-- IFTEClient_ABNresp =
238        PRE
239                size(callList(client))>0 &
240                last(callList(client)) = ifte  &
241                ifte_client : activedRequests  &
242                client = currentComponent
243
244        THEN
245                activedRequests := activedRequests - {ifte_client} ||
246                currentComponent := ifte ||
247                callList(client) := front(callList(client)) ||
248                sequenceHistory := sequenceHistory <- ifte_client_abn ||
```

```
249                 ANY val WHERE val : AbnormalReturns & (val : externalExceptions
                    ) /*& val /= noException*/
250                     THEN
251                         IF val /: maskableExceptions
252                             THEN
253                                 unrecoverable := TRUE
254                         END ||
255                         resp := val ||
256                         lastABNreturn := val
257                     END
258         END;
259
260
261
262
263
264 /*40*/
265 IFTEServer_req =
266         PRE
267                 ifte = currentComponent &
268                 ifte_server /: activedRequests
269         THEN
270                 currentComponent := server ||
271                 activedRequests := activedRequests \/ {ifte_server} ||
272                 callList(server) := callList(server) <- ifte ||
273                 sequenceHistory := [ifte_server_req]
274         END;
275
276
277 resp <-- IFTEServer_resp =
278         PRE
279                 size(callList(server))>0 &
280                 last(callList(server)) = ifte &
281                 ifte_server : activedRequests &
282                 server = currentComponent
283
284         THEN
285                 activedRequests := activedRequests - {ifte_server} ||
286                 currentComponent := ifte ||
287                 callList(server) := front(callList(server)) ||
288                 resp := norReturn ||
289                 sequenceHistory := sequenceHistory <- ifte_server_nor
290         END;
291
292
293 resp <-- IFTEServer_ABNresp =
294         PRE
295                 size(callList(server))>0 &
296                 last(callList(server)) = ifte &
297                 ifte_server : activedRequests &
298                 server = currentComponent
299
300         THEN
301                 activedRequests := activedRequests - {ifte_server} ||
```

```
302                 currentComponent := ifte ||
303                 callList(server) := front(callList(server)) ||
304                 sequenceHistory := sequenceHistory <- ifte_server_abn ||
305                 ANY val WHERE val : AbnormalReturns & (val : externalExceptions
                        ) /*& val /= noException*/
306                     THEN
307                         IF val /: maskableExceptions
308                             THEN
309                                 unrecoverable := TRUE
310                         END ||
311                         resp := val ||
312                         lastABNreturn := val
313                     END
314         END;
315
316
317
318
319 /*50*/
320 ServerClient_req =
321         PRE
322                 server = currentComponent &
323                 server_client /: activedRequests
324         THEN
325                 currentComponent := client ||
326                 activedRequests := activedRequests \/ {server_client} ||
327                 callList(client) := callList(client) <- server ||
328                 sequenceHistory := [server_client_req]
329         END;
330
331
332 resp <-- ServerClient_resp =
333         PRE
334                 size(callList(client))>0 &
335                 last(callList(client)) = server &
336                 server_client : activedRequests &
337                 client = currentComponent
338
339         THEN
340                 activedRequests := activedRequests - {server_client} ||
341                 currentComponent := server ||
342                 callList(client) := front(callList(client)) ||
343                 resp := norReturn ||
344                 sequenceHistory := sequenceHistory <- server_client_nor
345         END;
346
347
348 resp <-- ServerClient_ABNresp =
349         PRE
350                 size(callList(client))>0 &
351                 last(callList(client)) = server &
352                 server_client : activedRequests &
353                 client = currentComponent
354
```

```
355         THEN
356                 activedRequests := activedRequests - {server_client} ||
357                 currentComponent := server ||
358                 callList(client) := front(callList(client)) ||
359                 sequenceHistory := sequenceHistory <- server_client_abn ||
360                 ANY val WHERE val : AbnormalReturns & (val : externalExceptions
                        ) /*& val /= noException*/
361                     THEN
362                         IF val /: maskableExceptions
363                             THEN
364                                 unrecoverable := TRUE
365                         END ||
366                         resp := val ||
367                         lastABNreturn := val
368                     END
369         END
370 END
```

A.2 CSP Process Algebra

```
1 --SPECIFICATION:
2 MAIN = Start -> START2;;
3 START2 = (CLIENT [] Stop -> MAIN);;
4 CLIENT = ClientIFTE_req -> IFTE;;
5
6
7 -- Client -> iFTE
8 IFTE = ((IFTEServer_req -> SERVER) [] RETURN);;
9 RETURN = ((ClientIFTE_resp?R1 -> START2) [] (ClientIFTE_ABNresp?E1 -> START2))
      ;;
10
11 -- iFTE -> Server
12 SERVER = (SERVER_NORMAL [] SERVER_ABNORMAL);;
13 SERVER_NORMAL = IFTEServer_resp?R1 -> IFTE;;
14 SERVER_ABNORMAL = IFTEServer_ABNresp?E1 -> IFTE;;
```

B iFTE Internal Structure Model

B.1 B-Method Machine

```
1  MACHINE internalModel
2
3  /* ============================================ */
4  SETS
5          BOperations = {cliProv, cliProv_nor, cliProv_abn, provNor, provNor_nor,
               provNor_abn, provAbn, provAbn_nor, provAbn_abn, provReq,
              provReq_nor, provReq_abn, norReq, norReq_nor, norReq_abn, norProv,
              norProv_nor, norProv_abn, norAbn, norAbn_nor, norAbn_abn, abnNor,
              abnNor_nor, abnNor_abn, abnReq, abnReq_nor, abnReq_abn, abnProv,
              abnProv_nor, abnProv_abn, reqAbn, reqAbn_nor, reqAbn_abn, reqNor,
              reqNor_nor, reqNor_abn, reqProv, reqProv_nor, reqProv_abn, reqSrv,
              reqSrv_nor, reqSrv_abn}; /*Each defined operation and their
              possible returns*/
6          Components = {none, client, provided, normal, abnormal, required,
              server};
7          Calls = {start, cli_prov, prov_nor, prov_abn, prov_req, nor_req,
              nor_prov, nor_abn, abn_nor, abn_req, abn_prov, req_abn, req_nor,
              req_prov, req_srv, prov_srv, nor_srv, abn_srv};
8          NormalReturns = {norReturn};
9          AbnormalReturns = {noException, internalException1, failureException1,
              interfaceException1}
10
11
12
13 /* ============================================ */
14 VARIABLES
15         callList,                /*control variable*/
16         currentComponent,        /*control variable*/
17         activedRequests,         /*control variable*/
18 /*        sequenceHistory,         /*control variable*/*/
19         lastABNreturn,           /*control variable*/
20         lastExternalABNreturn,   /*control variable*/
21         unrecoverable,           /*control variable*/
22         maskableExceptions,
23         failureExceptions,
24         interfaceExceptions
25
26
27
28
29 /* ============================================ */
30 /*DEFINITIONS
31     GOAL == (...)*/
32
33
34
35 /* ============================================ */
36 INVARIANT
```

```
37                  callList : Components --> seq(Components) & /*for each
                        component, this variable stores the list of its callers (in
                        chronological order)*/
38                  currentComponent : Components & /*the component that is
                        actually working*/
39                  activedRequests : POW(Calls) & /*a set of calls that were not
                        finished*/
40                  sequenceHistory : seq(BOperations) & /*store the sequence of
                        operations, in order to provide traceability*/
41                  maskableExceptions : POW(AbnormalReturns) & /*exceptions which
                        can be handled*/
42                  failureExceptions : POW(AbnormalReturns) & /*failure exceptions
                        */
43                  interfaceExceptions : POW(AbnormalReturns) & /*interface
                        exceptions (can be signalled by the Provided)*/
44                  lastABNreturn : AbnormalReturns & /*indicate which exception
                        had been threw by some internal element */
45                  lastExternalABNreturn : AbnormalReturns & /*indicate which
                        exception had been threw by the Abnormal*/
46                  unrecoverable : BOOL /*indicates if a component can recover
                        itself of a failure. In other words, if it have received an
                        exception which cannot be masked*/
47
48
49
50 /* ============================== */
51 INITIALISATION
52          callList := {none |-> <>, client |-> <>, provided |-> <>, normal |->
                <>, abnormal |-> <>, required |-> <>, server |-> <>} ||
53          activedRequests := {} ||
54          currentComponent := none ||
55          sequenceHistory := <> ||
56          maskableExceptions := {internalException1} ||
57          failureExceptions := {failureException1} ||
58          interfaceExceptions := {interfaceException1} ||
59          lastABNreturn := noException ||
60          lastExternalABNreturn := noException ||
61          unrecoverable := FALSE
62
63
64
65 /* ============================== */
66 OPERATIONS
67
68 /*0*/
69 Start =
70          PRE
71                  none = currentComponent & card(activedRequests)=0
72          THEN
73                  currentComponent := client ||
74                  activedRequests := activedRequests \/ {start} ||
75                  sequenceHistory := <> ||
76                  lastABNreturn := noException ||
77                  lastExternalABNreturn := noException ||
```

```
78                  unrecoverable := FALSE
79          END;
80
81
82
83  /*00*/
84  Stop =
85          PRE
86                  client = currentComponent  &  start : activedRequests  &  card(
                        activedRequests)=1
87          THEN
88                  activedRequests := activedRequests - {start} ||
89                  currentComponent := none ||
90                  sequenceHistory := <> ||
91                  skip
92          END;
93
94
95
96
97  /*10*/
98  ClientProv_req =
99          PRE
100                 client = currentComponent  &
101                 cli_prov  /: activedRequests
102         THEN
103                 currentComponent := provided  ||
104                 activedRequests := activedRequests \/ {cli_prov} ||
105                 callList(provided) := callList(provided) <- client  ||
106                 sequenceHistory := [cliProv]
107         END;
108
109
110 resp <-- ClientProv_resp =
111         PRE
112                 size(callList(provided))>0 &
113                 last(callList(provided)) = client  &
114                 cli_prov : activedRequests  &
115                 provided = currentComponent
116
117         THEN
118                 activedRequests := activedRequests - {cli_prov} ||
119                 currentComponent := client  ||
120                 callList(provided) := front(callList(provided)) ||
121                 resp := norReturn ||
122                 sequenceHistory := sequenceHistory <- cliProv_nor
123         END;
124
125
126 resp <-- ClientProv_ABNresp =
127         PRE
128                 size(callList(provided))>0 &
129                 last(callList(provided)) = client  &
130                 cli_prov : activedRequests  &
```

```
131                 provided = currentComponent
132
133         THEN
134                 activedRequests := activedRequests - {cli_prov} ||
135                 currentComponent := client ||
136                 callList(provided) := front(callList(provided)) ||
137                 sequenceHistory := sequenceHistory <- cliProv_abn ||
138                 ANY val WHERE val : AbnormalReturns & ((val :
                        interfaceExceptions) or (val = lastExternalABNreturn)) &
                        val /= noException
139                       THEN
140                           IF val /: maskableExceptions
141                              THEN
142                                 unrecoverable := TRUE
143                           END ||
144                           resp := val ||
145                           lastABNreturn := val
146                       END
147         END;
148
149
150
151
152 /*20*/
153 ProvNor_req =
154         PRE
155                 provided = currentComponent  &
156                 prov_nor /: activedRequests
157         THEN
158                 currentComponent := normal ||
159                 activedRequests := activedRequests \/ {prov_nor} ||
160                 callList(normal) := callList(normal) <- provided ||
161                 sequenceHistory := sequenceHistory <- provNor
162         END;
163
164 resp <-- ProvNor_resp =
165         PRE
166                 size(callList(normal))>0 &
167                 last(callList(normal)) = provided  &
168                 prov_nor : activedRequests  &
169                 normal = currentComponent
170
171         THEN
172                 activedRequests := activedRequests - {prov_nor} ||
173                 currentComponent := provided ||
174                 callList(normal) := front(callList(normal)) ||
175                 resp := norReturn ||
176                 sequenceHistory := sequenceHistory <- provNor_nor
177         END;
178
179
180 resp <-- ProvNor_ABNresp =
181         PRE
182                 size(callList(normal))>0 &
```

```
183                 last(callList(normal)) = provided  &
184                 prov_nor : activedRequests  &
185                 normal = currentComponent
186
187         THEN
188                 activedRequests := activedRequests - {prov_nor} ||
189                 currentComponent := provided ||
190                 callList(normal) := front(callList(normal)) ||
191                 sequenceHistory := sequenceHistory <- provNor_abn ||
192                 ANY val WHERE val : AbnormalReturns & val /= noException & val
                        /: interfaceExceptions
193                     THEN
194                         IF val /: maskableExceptions
195                             THEN
196                                 unrecoverable := TRUE
197                         END ||
198                         resp := val ||
199                         lastABNreturn := val
200                     END
201         END;
202
203
204
205
206 /*30*/
207 ProvAbn_req =
208         PRE
209                 provided = currentComponent  &
210                 prov_abn /: activedRequests &
211                 lastABNreturn /= noException
212         THEN
213                 currentComponent := abnormal ||
214                 activedRequests := activedRequests \/ {prov_abn} ||
215                 callList(abnormal) := callList(abnormal) <- provided ||
216                 sequenceHistory := sequenceHistory <- provAbn
217         END;
218
219 resp <-- ProvAbn_resp =
220         PRE
221                 ((lastABNreturn = noException) or (lastABNreturn :
                        maskableExceptions)) &
222                 unrecoverable = FALSE &
223                 size(callList(abnormal))>0 &
224                 last(callList(abnormal)) = provided  &
225                 prov_abn : activedRequests  &
226                 abnormal = currentComponent
227
228         THEN
229                 lastABNreturn := noException ||
230                 activedRequests := activedRequests - {prov_abn} ||
231                 currentComponent := provided ||
232                 callList(abnormal) := front(callList(abnormal)) ||
233                 resp := norReturn ||
234                 sequenceHistory := sequenceHistory <- provAbn_nor
```

```
235         END;
236
237
238 resp <-- ProvAbn_ABNresp =
239         PRE
240                 size(callList(abnormal))>0 &
241                 last(callList(abnormal)) = provided &
242                 prov_abn : activedRequests &
243                 abnormal = currentComponent
244
245         THEN
246                 activedRequests := activedRequests - {prov_abn} ||
247                 currentComponent := provided ||
248                 callList(abnormal) := front(callList(abnormal)) ||
249                 sequenceHistory := sequenceHistory <- provAbn_abn ||
250                 ANY val WHERE val : AbnormalReturns & val : failureExceptions &
                        val /= noException & val /: interfaceExceptions
251                       THEN
252                          IF val /: maskableExceptions
253                             THEN
254                                 unrecoverable := TRUE
255                          END ||
256                          resp := val ||
257                          lastExternalABNreturn := val
258                       END
259         END;
260
261
262 /*40*/
263 ProvReq_req =
264         PRE
265                 provided = currentComponent &
266                 prov_req /: activedRequests
267         THEN
268                 currentComponent := required ||
269                 activedRequests := activedRequests \/ {prov_req} ||
270                 callList(required) := callList(required) <- provided ||
271                 sequenceHistory := sequenceHistory <- provReq
272         END;
273
274 resp <-- ProvReq_resp =
275         PRE
276
277                 size(callList(required))>0 &
278                 last(callList(required)) = provided &
279                 prov_req : activedRequests &
280                 required = currentComponent
281
282         THEN
283                 activedRequests := activedRequests - {prov_req} ||
284                 currentComponent := provided ||
285                 callList(required) := front(callList(required)) ||
286                 resp := norReturn ||
287                 sequenceHistory := sequenceHistory <- provReq_nor
```

```
288         END;
289
290
291 resp <-- ProvReq_ABNresp =
292         PRE
293                 size(callList(required))>0 &
294                 last(callList(required)) = provided  &
295                 prov_req : activedRequests  &
296                 required = currentComponent
297
298         THEN
299                 activedRequests := activedRequests - {prov_req} ||
300                 currentComponent := provided ||
301                 callList(required) := front(callList(required)) ||
302                 sequenceHistory := sequenceHistory <- provReq_abn ||
303                 ANY val WHERE val : AbnormalReturns & val /= noException & val
                        /: interfaceExceptions
304                       THEN
305                           IF val /: maskableExceptions
306                              THEN
307                                  unrecoverable := TRUE
308                           END ||
309                           resp := val ||
310                           lastABNreturn := val
311                       END
312         END;
313
314
315
316
317
318 /*50*/
319 NorReq_req =
320         PRE
321                 normal = currentComponent  &
322                 nor_req /: activedRequests
323         THEN
324                 currentComponent := required ||
325                 activedRequests := activedRequests \/ {nor_req} ||
326                 callList(required) := callList(required) <- normal ||
327                 sequenceHistory := sequenceHistory <- norReq
328         END;
329
330 resp <-- NorReq_resp =
331         PRE
332                 size(callList(required))>0 &
333                 last(callList(required)) = normal  &
334                 nor_req : activedRequests  &
335                 required = currentComponent
336
337         THEN
338                 activedRequests := activedRequests - {nor_req} ||
339                 currentComponent := normal ||
340                 callList(required) := front(callList(required)) ||
```

```
341                 resp := norReturn ||
342                 sequenceHistory := sequenceHistory <- norReq_nor
343         END;
344
345
346 resp <-- NorReq_ABNresp =
347         PRE
348                 size(callList(required))>0 &
349                 last(callList(required)) = normal &
350                 nor_req : activedRequests &
351                 required = currentComponent
352
353         THEN
354                 activedRequests := activedRequests - {nor_req} ||
355                 currentComponent := normal ||
356                 callList(required) := front(callList(required)) ||
357                 sequenceHistory := sequenceHistory <- norReq_abn ||
358                 ANY val WHERE val : AbnormalReturns & val /= noException & val
                        /: interfaceExceptions
359                    THEN
360                        IF val /: maskableExceptions
361                           THEN
362                              unrecoverable := TRUE
363                        END ||
364                        resp := val ||
365                        lastABNreturn := val
366                    END
367         END;
368
369 /*60*/
370 NorProv_req =
371         PRE
372                 normal = currentComponent &
373                 nor_prov /: activedRequests
374         THEN
375                 currentComponent := provided ||
376                 activedRequests := activedRequests \/ {nor_prov} ||
377                 callList(provided) := callList(provided) <- normal ||
378                 sequenceHistory := sequenceHistory <- norProv
379         END;
380
381 resp <-- NorProv_resp =
382         PRE
383                 size(callList(provided))>0 &
384                 last(callList(provided)) = normal &
385                 nor_prov : activedRequests &
386                 provided = currentComponent
387
388         THEN
389                 activedRequests := activedRequests - {nor_prov} ||
390                 currentComponent := normal ||
391                 callList(provided) := front(callList(provided)) ||
392                 resp := norReturn ||
393                 sequenceHistory := sequenceHistory <- norProv_nor
```

```
394         END;
395
396
397 resp <-- NorProv_ABNresp =
398         PRE
399                 size(callList(provided))>0 &
400                 last(callList(provided)) = normal  &
401                 nor_prov : activedRequests  &
402                 provided = currentComponent
403
404         THEN
405                 activedRequests := activedRequests - {nor_prov} ||
406                 currentComponent := normal ||
407                 callList(provided) := front(callList(provided)) ||
408                 sequenceHistory := sequenceHistory <- norProv_abn ||
409                 ANY val WHERE val : AbnormalReturns & val /= noException
410                       THEN
411                          IF val /: maskableExceptions
412                             THEN
413                                 unrecoverable := TRUE
414                          END ||
415                          resp := val ||
416                          lastABNreturn := val
417                       END
418         END;
419
420
421 /*70*/
422 NorAbn_req =
423         PRE
424                 normal = currentComponent  &
425                 nor_abn /: activedRequests
426         THEN
427                 currentComponent := abnormal ||
428                 activedRequests := activedRequests \/ {nor_abn} ||
429                 callList(abnormal) := callList(abnormal) <- normal ||
430                 sequenceHistory := sequenceHistory <- norAbn
431         END;
432
433 resp <-- NorAbn_resp =
434         PRE
435                 ((lastABNreturn = noException) or (lastABNreturn :
                        maskableExceptions)) &
436                 unrecoverable = FALSE &
437                 size(callList(abnormal))>0 &
438                 last(callList(abnormal)) = normal  &
439                 nor_abn : activedRequests  &
440                 abnormal = currentComponent
441
442         THEN
443                 lastABNreturn := noException ||
444                 activedRequests := activedRequests - {nor_abn} ||
445                 currentComponent := normal ||
446                 callList(abnormal) := front(callList(abnormal)) ||
```

```
447                 resp := norReturn ||
448                 sequenceHistory := sequenceHistory <- norAbn_nor
449         END;
450
451
452 resp <-- NorAbn_ABNresp =
453         PRE
454                 size(callList(abnormal))>0 &
455                 last(callList(abnormal)) = normal  &
456                 nor_abn : activedRequests  &
457                 abnormal = currentComponent
458
459         THEN
460                 activedRequests := activedRequests - {nor_abn} ||
461                 currentComponent := normal ||
462                 callList(abnormal) := front(callList(abnormal)) ||
463                 sequenceHistory := sequenceHistory <- norAbn_abn ||
464                 ANY val WHERE val : AbnormalReturns & val : failureExceptions &
                        val /= noException & val /: interfaceExceptions
465                       THEN
466                          IF val /: maskableExceptions
467                             THEN
468                                unrecoverable := TRUE
469                          END ||
470                          resp := val ||
471                          lastExternalABNreturn := val
472                       END
473         END;
474
475
476
477 /*80*/
478 AbnNor_req =
479         PRE
480                 abnormal = currentComponent  &
481                 abn_nor /: activedRequests
482         THEN
483                 currentComponent := normal ||
484                 activedRequests := activedRequests \/ {abn_nor} ||
485                 callList(normal) := callList(normal) <- abnormal ||
486                 sequenceHistory := sequenceHistory <- abnNor
487         END;
488
489 resp <-- AbnNor_resp =
490         PRE
491                 size(callList(normal))>0 &
492                 last(callList(normal)) = abnormal  &
493                 abn_nor : activedRequests  &
494                 normal = currentComponent
495
496         THEN
497                 activedRequests := activedRequests - {abn_nor} ||
498                 currentComponent := abnormal ||
499                 callList(normal) := front(callList(normal)) ||
```

```
500                resp :=  norReturn ||
501                sequenceHistory := sequenceHistory <- abnNor_nor
502         END;
503
504
505 resp <-- AbnNor_ABNresp =
506         PRE
507                size(callList(normal))>0 &
508                last(callList(normal)) = abnormal  &
509                abn_nor : activedRequests  &
510                normal = currentComponent
511
512         THEN
513                activedRequests := activedRequests - {abn_nor} ||
514                currentComponent := abnormal ||
515                callList(normal) := front(callList(normal)) ||
516                sequenceHistory := sequenceHistory <- abnNor_abn ||
517                ANY val WHERE val : AbnormalReturns & val /= noException & val
                       /: interfaceExceptions
518                    THEN
519                       IF val /: maskableExceptions
520                          THEN
521                              unrecoverable := TRUE
522                       END ||
523                       resp := val ||
524                       lastABNreturn := val
525                    END
526         END;
527
528
529
530
531 /*90*/
532 AbnReq_req =
533         PRE
534                abnormal = currentComponent  &
535                abn_req /: activedRequests
536         THEN
537                currentComponent := required ||
538                activedRequests := activedRequests \/ {abn_req} ||
539                callList(required) := callList(required) <- abnormal ||
540                sequenceHistory := sequenceHistory <- abnReq
541         END;
542
543 resp <-- AbnReq_resp =
544         PRE
545                size(callList(required))>0 &
546                last(callList(required)) = abnormal  &
547                abn_req : activedRequests  &
548                required = currentComponent
549
550         THEN
551                activedRequests := activedRequests - {abn_req} ||
552                currentComponent := abnormal ||
```

```
553                 callList(required) := front(callList(required)) ||
554                 resp := norReturn ||
555                 sequenceHistory := sequenceHistory <- abnReq_nor
556         END;
557
558
559 resp <-- AbnReq_ABNresp =
560         PRE
561                 size(callList(required))>0 &
562                 last(callList(required)) = abnormal  &
563                 abn_req : activedRequests  &
564                 required = currentComponent
565
566         THEN
567                 activedRequests := activedRequests - {abn_req} ||
568                 currentComponent := abnormal ||
569                 callList(required) := front(callList(required)) ||
570                 sequenceHistory := sequenceHistory <- abnReq_abn ||
571                 ANY val WHERE val : AbnormalReturns & val /= noException & val
                        /: interfaceExceptions
572                     THEN
573                         IF val /: maskableExceptions
574                             THEN
575                                 unrecoverable := TRUE
576                         END ||
577                         resp := val ||
578                         lastABNreturn := val
579                     END
580         END;
581
582
583 /*100*/
584 AbnProv_req =
585         PRE
586                 abnormal = currentComponent  &
587                 abn_prov /: activedRequests
588         THEN
589                 currentComponent := provided ||
590                 activedRequests := activedRequests \/ {abn_prov} ||
591                 callList(provided) := callList(provided) <- abnormal ||
592                 sequenceHistory := sequenceHistory <- abnProv
593         END;
594
595 resp <-- AbnProv_resp =
596         PRE
597                 size(callList(provided))>0 &
598                 last(callList(provided)) = abnormal  &
599                 abn_prov : activedRequests  &
600                 provided = currentComponent
601
602         THEN
603                 activedRequests := activedRequests - {abn_prov} ||
604                 currentComponent := abnormal ||
605                 callList(provided) := front(callList(provided)) ||
```

```
606                 resp := norReturn ||
607                 sequenceHistory := sequenceHistory <- abnProv_nor
608         END;
609
610
611 resp <-- AbnProv_ABNresp =
612         PRE
613                 size(callList(provided))>0 &
614                 last(callList(provided)) = abnormal  &
615                 abn_prov : activedRequests  &
616                 provided = currentComponent
617
618         THEN
619                 activedRequests := activedRequests - {abn_prov} ||
620                 currentComponent := abnormal ||
621                 callList(provided) := front(callList(provided)) ||
622                 sequenceHistory := sequenceHistory <- abnProv_abn ||
623                 ANY val WHERE val : AbnormalReturns & val /= noException
624                        THEN
625                            IF val /: maskableExceptions
626                               THEN
627                                   unrecoverable := TRUE
628                            END ||
629                            resp := val ||
630                            lastABNreturn := val
631                        END
632         END;
633
634
635
636
637 /*110*/
638 ReqAbn_req =
639         PRE
640                 required = currentComponent  &
641                 req_abn /: activedRequests
642         THEN
643                 currentComponent := abnormal ||
644                 activedRequests := activedRequests \/ {req_abn} ||
645                 callList(abnormal) := callList(abnormal) <- required ||
646                 sequenceHistory := sequenceHistory <- reqAbn
647         END;
648
649 resp <-- ReqAbn_resp =
650         PRE
651                 ((lastABNreturn = noException) or (lastABNreturn :
                        maskableExceptions)) &
652                 unrecoverable = FALSE &
653                 size(callList(abnormal))>0 &
654                 last(callList(abnormal)) = required  &
655                 req_abn : activedRequests  &
656                 abnormal = currentComponent
657
658         THEN
```

```
659                 lastABNreturn := noException ||
660                 activedRequests := activedRequests - {req_abn} ||
661                 currentComponent := required ||
662                 callList(abnormal) := front(callList(abnormal)) ||
663                 resp := norReturn ||
664                 sequenceHistory := sequenceHistory <- reqAbn_nor
665         END;
666
667
668 resp <-- ReqAbn_ABNresp =
669         PRE
670                 size(callList(abnormal))>0 &
671                 last(callList(abnormal)) = required  &
672                 req_abn : activedRequests  &
673                 abnormal = currentComponent
674
675         THEN
676                 activedRequests := activedRequests - {req_abn} ||
677                 currentComponent := required ||
678                 callList(abnormal) := front(callList(abnormal)) ||
679                 sequenceHistory := sequenceHistory <- reqAbn_abn ||
680                 ANY val WHERE val : AbnormalReturns & val : failureExceptions &
                         val /= noException & val /: interfaceExceptions
681                          THEN
682                              IF val /: maskableExceptions
683                                  THEN
684                                      unrecoverable := TRUE
685                              END ||
686                              resp := val ||
687                              lastExternalABNreturn := val
688                          END
689         END;
690
691
692 /*120*/
693 ReqNor_req =
694         PRE
695                 required = currentComponent  &
696                 req_nor /: activedRequests
697         THEN
698                 currentComponent := normal ||
699                 activedRequests := activedRequests \/ {req_nor} ||
700                 callList(normal) := callList(normal) <- required ||
701                 sequenceHistory := sequenceHistory <- reqNor
702         END;
703
704 resp <-- ReqNor_resp =
705         PRE
706                 size(callList(normal))>0 &
707                 last(callList(normal)) = required  &
708                 req_nor : activedRequests  &
709                 normal = currentComponent
710
711         THEN
```

```
712                  activedRequests := activedRequests - {req_nor} ||
713                  currentComponent := required ||
714                  callList(normal) := front(callList(normal)) ||
715                  resp := norReturn ||
716                  sequenceHistory := sequenceHistory <- reqNor_nor
717          END;
718
719
720
721 resp <-- ReqNor_ABNresp =
722          PRE
723                  size(callList(normal))>0 &
724                  last(callList(normal)) = required  &
725                  req_nor : activedRequests  &
726                  normal = currentComponent
727
728          THEN
729                  activedRequests := activedRequests - {req_nor} ||
730                  currentComponent := required ||
731                  callList(normal) := front(callList(normal)) ||
732                  sequenceHistory := sequenceHistory <- reqNor_abn ||
733                  ANY val WHERE val : AbnormalReturns & val /= noException & val
                         /: interfaceExceptions
734                        THEN
735                            IF val /: maskableExceptions
736                               THEN
737                                   unrecoverable := TRUE
738                            END ||
739                            resp := val ||
740                            lastABNreturn := val
741                        END
742          END;
743
744
745 /*130*/
746 ReqProv_req =
747          PRE
748                  required = currentComponent  &
749                  req_prov /: activedRequests
750          THEN
751                  currentComponent := provided ||
752                  activedRequests := activedRequests \/ {req_prov} ||
753                  callList(provided) := callList(provided) <- required ||
754                  sequenceHistory := sequenceHistory <- reqProv
755          END;
756
757 resp <-- ReqProv_resp =
758          PRE
759                  size(callList(provided))>0 &
760                  last(callList(provided)) = required  &
761                  req_prov : activedRequests  &
762                  provided = currentComponent
763
764          THEN
```

```
765                 activedRequests := activedRequests - {req_prov} ||
766                 currentComponent := required ||
767                 callList(provided) := front(callList(provided)) ||
768                 resp := norReturn ||
769                 sequenceHistory := sequenceHistory <- reqProv_nor
770         END;
771
772
773 resp <-- ReqProv_ABNresp =
774         PRE
775                 size(callList(provided))>0 &
776                 last(callList(provided)) = required &
777                 req_prov : activedRequests &
778                 provided = currentComponent
779
780         THEN
781                 activedRequests := activedRequests - {req_prov} ||
782                 currentComponent := required ||
783                 callList(provided) := front(callList(provided)) ||
784                 sequenceHistory := sequenceHistory <- reqProv_abn ||
785                 ANY val WHERE val : AbnormalReturns & val /= noException
786                         THEN
787                             IF val /: maskableExceptions
788                                 THEN
789                                     unrecoverable := TRUE
790                             END ||
791                             resp := val ||
792                             lastABNreturn := val
793                         END
794         END;
795
796
797
798
799 /*140*/
800 ReqServer_req =
801         PRE
802                 required = currentComponent &
803                 req_srv /: activedRequests
804         THEN
805                 currentComponent := server ||
806                 activedRequests := activedRequests \/ {req_srv} ||
807                 callList(server) := callList(server) <- required ||
808                 sequenceHistory := sequenceHistory <- reqSrv
809         END;
810
811 resp <-- ReqServer_resp =
812         PRE
813                 size(callList(server))>0 &
814                 last(callList(server)) = required &
815                 req_srv : activedRequests &
816                 server = currentComponent
817
818         THEN
```

```
819                 activedRequests := activedRequests - {req_srv} ||
820                 currentComponent := required ||
821                 callList(server) := front(callList(server)) ||
822                 resp := norReturn ||
823                 sequenceHistory := sequenceHistory <- reqSrv_nor
824         END;
825
826
827 resp <-- ReqServer_ABNresp =
828         PRE
829                 size(callList(server))>0 &
830                 last(callList(server)) = required &
831                 req_srv : activedRequests &
832                 server = currentComponent
833
834         THEN
835                 activedRequests := activedRequests - {req_srv} ||
836                 currentComponent := required ||
837                 callList(server) := front(callList(server)) ||
838                 sequenceHistory := sequenceHistory <- reqSrv_abn ||
839                 ANY val WHERE val : AbnormalReturns & val /= noException & val
                        /: interfaceExceptions
840                     THEN
841                         IF val /: maskableExceptions
842                             THEN
843                                 unrecoverable := TRUE
844                         END ||
845                         resp := val ||
846                         lastABNreturn := val
847                     END
848         END
849
850 END
```

B.2 CSP Process Algebra

```
1 --SPECIFICATION:
2 MAIN = Start -> START2;;
3 START2 = (CLIENT [] Stop -> MAIN);;
4 CLIENT = ClientProv_req -> PROVIDED;;
5
6
7 -- *********************
8 -- ***NORMAL EXECUTION***
9 -- *********************
10
11 -- Client -> Provided
12 PROVIDED = ((ProvNor_req -> NORMAL) [] INTERFACE_EXCEPTION);;
13 INTERFACE_EXCEPTION = ClientProv_ABNresp?E -> Stop -> MAIN;;
14
15 -- Provided -> Normal
```

```
16 NORMAL = (NORMAL_NORMAL_REPONSE [] NORMAL_RAISES_EXCEPTION [] (NorReq_req ->
       REQUIRED));;
17 NORMAL_NORMAL_REPONSE = ProvNor_resp?R1 -> ClientProv_resp.R1-> START2;;
18 NORMAL_RAISES_EXCEPTION = ProvNor_ABNresp?IE -> HANDLING(IE);; --internal
       exception
19
20 -- Provided -> Normal -> Required
21 REQUIRED = ReqServer_req -> (EXTERNAL_NOR [] EXTERNAL_ABN);; --external
       service
22 EXTERNAL_NOR = ReqServer_resp?R2 -> (REQUIRED [] (NorReq_resp.R3 -> NORMAL));;
23 EXTERNAL_ABN = ReqServer_ABNresp?EE -> NorReq_ABNresp.EE -> ProvNor_ABNresp.EE
       -> HANDLING(EE);;
24
25
26 -- **************
27 -- ***HANDLING***
28 -- **************
29
30 -- Provided -> Abnormal
31 HANDLING(E) = ProvAbn_req -> (SUCCESS_OR_UNSUCCESS);;
32 SUCCESS = (ABN_NOR [] ABN_REQ [] (ProvAbn_resp.R2-> ClientProv_resp.R3 ->
       START2));; --handling
33 ABN_NOR = AbnNor_req -> (ABN_NOR_NORMAL [] ABN_NOR_INTERNAL_ESCEPTION []
       ABN_NOR_REQUIRED);; --normal or external service
34 ABN_REQ = AbnReq_req -> ABN_REQ_REQUIRED;;--external services
35 UNSUCCESS = ProvAbn_ABNresp?E1 -> ClientProv_ABNresp.E1 -> Stop -> MAIN;;
36 SUCCESS_OR_UNSUCCESS = SUCCESS [] UNSUCCESS;;
37
38 -- Abnormal -> Normal
39 ABN_NOR_NORMAL = (ABN_NOR_NORMAL_NORMAL_REPONSE []
       ABN_NOR_NORMAL_RAISES_EXCEPTION [] (NorReq_req -> ABN_NOR_REQUIRED));;
40 ABN_NOR_NORMAL_NORMAL_REPONSE = AbnNor_resp?R1 -> SUCCESS_OR_UNSUCCESS;; --
       can be an abort
41 ABN_NOR_NORMAL_RAISES_EXCEPTION = AbnNor_ABNresp?IE -> SUCCESS_OR_UNSUCCESS;;
       --internal exception - it is possible to try again
42
43 -- Abnormal -> Normal -> Req
44 ABN_NOR_REQUIRED = ReqServer_req -> (ABN_NOR_EXTERNAL_NOR []
       ABN_NOR_EXTERNAL_ABN);; --external service
45 ABN_NOR_EXTERNAL_NOR = ReqServer_resp?R2 -> (ABN_NOR_REQUIRED [] (NorReq_resp.
       R3 -> ABN_NOR_NORMAL));;
46 ABN_NOR_EXTERNAL_ABN = ReqServer_ABNresp?EE -> NorReq_ABNresp.EE ->
       AbnNor_ABNresp.EE -> SUCCESS_OR_UNSUCCESS;;
47
48 -- Abnormal -> Required
49 ABN_REQ_REQUIRED = ReqServer_req -> (ABN_REQ_EXTERNAL_NOR []
       ABN_REQ_EXTERNAL_ABN);; --external service
50 ABN_REQ_EXTERNAL_NOR = ReqServer_resp?R2 -> (ABN_REQ_REQUIRED [] (AbnReq_resp.
       R3 -> SUCCESS_OR_UNSUCCESS));; -- it is possible to execute an external
       service (e.g. logging) despite the fault has not been masked.
51 ABN_REQ_EXTERNAL_ABN = ReqServer_ABNresp?EE -> AbnReq_ABNresp.EE ->
       SUCCESS_OR_UNSUCCESS;; -- it is possible an exception decurrent of an
       external service (e.g. logging), but the fault has been masked before it.
```

C High-level Architecture Model

C.1 B-Method Machine

```
1  MACHINE hiLevelArchitecture
2
3  /* ================================ */
4  SETS
5          EventType = {request, response};
6          NormalReturns = {norReturn};
7          AbnormalReturns = {noException, abnReturn1, abnReturn2};
8          ArchitecturalElements = {none, client1, server1a, server1b, server2,
               c1_s1, c1_s2};
9          Interfaces = {interface1, interface2}
10
11 /* ================================ */
12 VARIABLES
13         currentComponent, /*control variable*/
14         activedCalls,     /*control variable*/
15         initialComponent, /*control variable*/
16         dependences,
17         faultfreeComponents,
18         componentsReceivedException, /*control variable*/
19         lastRequisition, /*control variable*/
20         unrecoverable,   /*control variable*/
21 /*        sequenceHistory, /*control variable*/*/
22         componentInterfaces,
23         interfaceExceptions,
24         componentExceptions,
25         maskableExceptions,
26         exceptionHierarchy,
27         handlingBySubsumption
28
29
30
31 /* ================================ */
32 /*DEFINITIONS
33    GOAL == (...)
34
35
36
37
38 /* ================================ */
39 INVARIANT
40                 currentComponent : ArchitecturalElements & /*the component that
                      is actually working*/
41                 activedCalls : ArchitecturalElements --> POW(
                    ArchitecturalElements) & /*a set of calls that were not
                    finished*/
42                 initialComponent : ArchitecturalElements & /*used to stop the
                    interation*/
43                 dependences : ArchitecturalElements +-> POW(
                    ArchitecturalElements) & /*materialization of the
```

```
            configuration*/
44        faultfreeComponents : POW(ArchitecturalElements) & /*a set of
            the components that does not fail - default, connectors are
             bastionized*/
45        componentsReceivedException :POW(ArchitecturalElements) & /*a
            set of components that received exceptions*/
46        lastRequisition : ArchitecturalElements*ArchitecturalElements
            +-> Interfaces & /*the last interface REQUESTED from a
            component to other*/
47        unrecoverable : POW(ArchitecturalElements) & /*architectural
            elements which received at least one external and not
            maskable exception*/
48        sequenceHistory : seq(ArchitecturalElements*
            ArchitecturalElements*EventType*AbnormalReturns) & /*store
            the sequence of operations, in order to provide
            traceability*/
49
50        componentInterfaces : ArchitecturalElements +-> POW(Interfaces)
             & /*interfaces provided for a specific component*/
51        interfaceExceptions : Interfaces +-> POW(AbnormalReturns) & /*
            exceptions that an interface may throws*/
52        componentExceptions : ArchitecturalElements +-> POW(
            AbnormalReturns) & /*exceptions that a component may throw
            */
53        maskableExceptions : ArchitecturalElements +-> POW(
            AbnormalReturns) & /*exceptions that a component may mask*/
54        exceptionHierarchy : AbnormalReturns --> POW(AbnormalReturns) &
             /* subtype --> supertypeS*/
55        handlingBySubsumption : BOOL /*a flag to indicate if is to
            handle exceptions with subsumption*/
56
57
58
59 /* ================================ */
60 INITIALISATION
61     currentComponent := none ||
62     activedCalls := {none |-> {}, client1 |-> {}, server1a |-> {}, server1b
          |-> {}, server2 |-> {}, c1_s1 |-> {}, c1_s2 |-> {}} ||
63     initialComponent := none ||
64     dependences := {client1 |-> {c1_s1, c1_s2}, server1a |-> {}, server1b
         |-> {}, server2 |-> {}, c1_s1 |-> {server1a, server1b}, c1_s2 |-> {
         server2}} ||
65     faultfreeComponents := {c1_s1, c1_s2} ||
66     componentsReceivedException := {} ||
67     lastRequisition := {} ||
68     unrecoverable := {} ||
69     sequenceHistory := <>||
70     componentInterfaces := {client1 |-> {}, server1a |-> {interface1},
         server1b |-> {interface1, interface2}, server2 |-> {interface2},
         c1_s1 |-> {interface1, interface2}, c1_s2 |-> {interface2}} ||
71     interfaceExceptions := {interface1 |-> {abnReturn1}, interface2 |-> {
         abnReturn1, abnReturn2}} ||
72     componentExceptions := {client1 |-> {}, server1a |-> {abnReturn1},
         server1b |-> {abnReturn1}, server2 |-> {abnReturn1}, c1_s1 |-> {
```

```
            abnReturn1}, c1_s2 |-> {abnReturn1}} ||
73        maskableExceptions := {client1 |-> {abnReturn1}, server1a |-> {
            abnReturn1}, server1b |-> {abnReturn1}, server2 |-> {abnReturn1},
            c1_s1 |-> {abnReturn1}, c1_s2 |-> {abnReturn1}} ||
74        exceptionHierarchy := {noException |-> {}, abnReturn1 |-> {},
            abnReturn2 |-> {abnReturn1}} ||
75        handlingBySubsumption := FALSE
76
77
78
79
80
81 /* ================================ */
82 OPERATIONS
83
84 /* */
85 Start(from) =
86        PRE
87                from : ArchitecturalElements &
88                none = currentComponent
89        THEN
90                initialComponent := from ||
91                currentComponent := from ||
92                componentsReceivedException := {} ||
93                unrecoverable := {} ||
94                activedCalls(none) := activedCalls(none) \/ {none} ||
95                sequenceHistory := <>
96        END;
97
98
99
100
101
102
103 /* */
104 Stop =
105        PRE
106                initialComponent = currentComponent & none : activedCalls(
                    none) & card(activedCalls(none))=1
107        THEN
108                activedCalls(none) := activedCalls(none) - {none} ||
109                currentComponent := none ||
110                skip
111        END;
112
113
114
115
116
117
118 /* */
119 Service_req(from, to, interface) =
120        PRE
```

```
121                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
122                 from = currentComponent  & to /: activedCalls(from) &
123                 interface : Interfaces & interface : componentInterfaces(to)
124         THEN
125                 currentComponent := to ||
126                 activedCalls(from) := activedCalls(from) \/ {to} ||
127                 lastRequisition(from|->to) := interface ||
128                 sequenceHistory := sequenceHistory <- (from |-> to |-> request
                        |-> noException)
129         END;
130
131
132
133
134
135
136
137 resp <-- Service_resp(from, to, interface) =
138         PRE
139                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
140                 to : activedCalls(from)  & to = currentComponent &
141                 interface : Interfaces & interface : componentInterfaces(to) &
142                 lastRequisition(from|->to) = interface &
143                 to /: unrecoverable
144
145         THEN
146                 activedCalls(from) := activedCalls(from) - {to} ||
147                 currentComponent := from ||
148                 componentsReceivedException := componentsReceivedException - {
                        to} || /*the operation finished - context cleaning*/
149                 ANY val WHERE val : NormalReturns
150                     THEN
151                         resp := val
152                     END ||
153                 sequenceHistory := sequenceHistory <- (from |-> to |-> response
                         |-> noException)
154         END;
155
156
157
158
159
160
161
162
163
164 resp <-- Service_ABNresp(from, to, interface) =
165         PRE
166                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
167                 ((to /: faultfreeComponents) or (to :
                        componentsReceivedException)) &
```

```
168                 to : activedCalls(from)  &
169                 to = currentComponent &
170                 interface : Interfaces & interface : componentInterfaces(to) &
171                 lastRequisition(from|->to) = interface
172
173         THEN
174                 activedCalls(from) := activedCalls(from) - {to} ||
175                 currentComponent := from ||
176                 componentsReceivedException := componentsReceivedException \/ {
                        from} || /*destination of the exception*/
177                 ANY val WHERE val : AbnormalReturns & val : interfaceExceptions
                        (interface)
178                 THEN
179                     IF not ((val : maskableExceptions(from)) or (((
                            exceptionHierarchy(val) /\ maskableExceptions(from)) /=
                            {}) & (handlingBySubsumption = TRUE)))
180                         THEN
181                             unrecoverable := unrecoverable \/ {from}
182                     END ||
183                     resp := val ||
184                     sequenceHistory := sequenceHistory <- (from |-> to |->
                            response |-> val)
185                 END
186         END
187 END
```

C.2 CSP Process Algebra

```
1 MAIN = Start!client1 -> REQUEST1;;
2 REQUEST1 = ((Service_req!from!to!interface -> EXEC) [] Stop -> MAIN);;
3
4
5 -- a -> a_b
6 EXEC = (RETURN1 [] REQUEST2);;
7 RETURN1 = ((Service_resp!from!to!interface?N -> REQUEST1) [] (Service_ABNresp!
     from!to!interface?E -> REQUEST1));;
8
9 ABN_EXEC = (RETURN1_ERR [] REQUEST2);;
10 RETURN1_ERR = (Service_ABNresp!from!to!interface?E -> REQUEST1);; --if receive
      an exception, it is not possible to return normally without another
     external request
11
12 -- a_b -> b
13 REQUEST2 = (Service_req!from!to!interface -> RETURN2);;
14 RETURN2 = ((Service_resp!from!to!interface?N -> EXEC) [] (Service_ABNresp!from
     !to!interface?E -> ABN_EXEC));;
```

D Architecture Detailed Model

D.1 B-Method Machine

```
1 MACHINE configuration
2
3 /* ================================ */
4 SETS
5         EventType = {request, response};
6         NormalReturns = {normalReturn};
7         AbnormalReturns = {noException, abnReturn1, abnReturn2};
8         ArchitecturalElements = {none, clientComponent, connector,
              serverComponent};
9         DetailedArchitecturalElements = {connector, serverComponent};
10        Interfaces = {interface1, interface2}
11 /* ================================ */
12 EXTENDS
13    conn,
14    server
15
16 /* ================================ */
17 VARIABLES
18        currentComponent, /*control variable*/
19        activedCalls,     /*control variable*/
20        initialComponent, /*control variable*/
21        dependences,
22        faultfreeComponents,
23        componentsReceivedException, /*control variable*/
24        lastRequisition, /*control variable*/
25        unrecoverable,   /*control variable*/
26 /*       sequenceHistory, /*control variable*/*/
27        componentInterfaces,
28        interfaceExceptions,
29        componentExceptions,
30        maskableExceptions,
31        exceptionHierarchy,
32        handlingBySubsumption
33
34
35
36 /* ================================ */
37 /*DEFINITIONS
38    GOAL == (goal)*/
39
40
41
42
43 /* ================================ */
44 INVARIANT
45                currentComponent : ArchitecturalElements & /*the component that
                      is actually working*/
46                activedCalls : ArchitecturalElements --> POW(
                     ArchitecturalElements) & /*a set of calls that were not
```

```
                finished*/
47          initialComponent : ArchitecturalElements & /*used to stop the
                interation*/
48          dependences : ArchitecturalElements +-> POW(
                ArchitecturalElements) & /*materialization of the
                configuration*/
49          faultfreeComponents : POW(ArchitecturalElements) & /*a set of
                the components that does not fail - default, connectors are
                 bastionized*/
50          componentsReceivedException :POW(ArchitecturalElements) & /*a
                set of components that received exceptions*/
51          lastRequisition : ArchitecturalElements*ArchitecturalElements
                +-> Interfaces & /*the last interface REQUESTED from a
                component to other*/
52          unrecoverable : POW(ArchitecturalElements) & /*architectural
                elements which received at least one external and not
                maskable exception*/
53          sequenceHistory : seq(ArchitecturalElements*
                ArchitecturalElements*EventType*AbnormalReturns) & /*store
                the sequence of operations, in order to provide
                traceability*/
54
55          componentInterfaces : ArchitecturalElements +-> POW(Interfaces)
                 & /*interfaces provided for a specific component*/
56          interfaceExceptions : Interfaces +-> POW(AbnormalReturns) & /*
                exceptions that as interface may throws*/
57          componentExceptions : ArchitecturalElements +-> POW(
                AbnormalReturns) & /*exceptions that a component may throw
                */
58          maskableExceptions : ArchitecturalElements +-> POW(
                AbnormalReturns) & /*exceptions that a component may mask*/
59          exceptionHierarchy : AbnormalReturns --> POW(AbnormalReturns) &
                 /* subtype --> supertypeS*/
60          handlingBySubsumption : BOOL /*a flag to indicate if is to
                handle exceptions with subsumption*/
61
62
63
64 /* ================================ */
65 INITIALISATION
66      currentComponent := none ||
67      activedCalls := {none |-> {}, clientComponent |-> {}, connector |-> {},
             serverComponent |-> {}} ||
68      initialComponent := none ||
69      dependences := {clientComponent |-> {connector}, connector |-> {
            serverComponent}, serverComponent |-> {}} ||
70      faultfreeComponents := {} || /*Nao utilizar*/
71      componentsReceivedException := {} ||
72      lastRequisition := {} ||
73      unrecoverable := {} ||
74      componentInterfaces := {clientComponent |-> {}, connector |-> {
            interface1, interface2}, serverComponent |-> {interface1,
            interface2}} ||
75      sequenceHistory := <>||
```

```
76         interfaceExceptions := {interface1 |-> {abnReturn1}, interface2 |-> {
               abnReturn1, abnReturn2}} ||
77         componentExceptions := {clientComponent |-> {}, connector |-> {
               abnReturn1}, serverComponent |-> {abnReturn1, abnReturn2}} ||
78         maskableExceptions := {clientComponent |-> {abnReturn1}, connector |->
               {abnReturn1}, serverComponent |-> {}} ||
79         exceptionHierarchy := {noException |-> {}, abnReturn1 |-> {},
               abnReturn2 |-> {abnReturn1}} ||
80         handlingBySubsumption := FALSE
81
82
83
84
85
86 /* ================================ */
87 OPERATIONS
88
89 /* */
90 Start(from) =
91         PRE
92                 from : ArchitecturalElements &
93                 none = currentComponent
94         THEN
95                 initialComponent := from ||
96                 currentComponent := from ||
97                 componentsReceivedException := {} ||
98                 unrecoverable := {} ||
99                 activedCalls(none) := activedCalls(none) \/ {none} ||
100                sequenceHistory := <>
101        END;
102
103
104
105
106
107
108 /* */
109 Stop =
110        PRE
111                initialComponent = currentComponent & none : activedCalls(
                       none) & card(activedCalls(none))=1
112        THEN
113                activedCalls(none) := activedCalls(none) - {none} ||
114                currentComponent := none ||
115                skip
116        END;
117
118
119
120
121
122
123 /* */
124 Service_req(from, to, interface) =
```

```
125         PRE
126                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
127                 from = currentComponent  & to /: activedCalls(from) &
128                 interface : Interfaces & interface : componentInterfaces(to)
129         THEN
130                 currentComponent := to ||
131                 activedCalls(from) := activedCalls(from) \/ {to} ||
132                 lastRequisition(from|->to) := interface ||
133                 sequenceHistory := sequenceHistory <- (from |-> to |-> request
                        |-> noException)
134         END;
135
136
137
138
139
140
141
142 resp <-- Service_resp(from, to, interface) =
143         PRE
144                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
145                 to : activedCalls(from)  & to = currentComponent &
146                 interface : Interfaces & interface : componentInterfaces(to) &
147                 lastRequisition(from|->to) = interface &
148                 to /: unrecoverable
149
150         THEN
151                 activedCalls(from) := activedCalls(from) - {to} ||
152                 currentComponent := from ||
153                 componentsReceivedException := componentsReceivedException - {
                        to} || /*the operation finished - context cleaning*/
154                 ANY val WHERE val : NormalReturns
155                    THEN
156                        resp := val
157                    END ||
158                 sequenceHistory := sequenceHistory <- (from |-> to |-> response
                         |-> noException)
159         END;
160
161
162
163
164
165
166
167
168
169 resp <-- Service_ABNresp(from, to, interface) =
170         PRE
171                 from : ArchitecturalElements & to : ArchitecturalElements & to
                        : dependences(from) &
```

```
172                 ((to /: faultfreeComponents) or (to :
                        componentsReceivedException)) &
173                 to : activedCalls(from)  &
174                 to = currentComponent &
175                 interface : Interfaces & interface : componentInterfaces(to) &
176                 lastRequisition(from|->to) = interface
177
178         THEN
179                 activedCalls(from) := activedCalls(from) - {to} ||
180                 currentComponent := from ||
181                 componentsReceivedException := componentsReceivedException \/ {
                        from} || /*destination of the exception*/
182                 ANY val WHERE val : AbnormalReturns & val : interfaceExceptions
                        (interface)
183                 THEN
184                     IF not ((val : maskableExceptions(from)) or (((
                            exceptionHierarchy(val) /\ maskableExceptions(from)) /=
                            {}) & (handlingBySubsumption = TRUE)))
185                         THEN
186                             unrecoverable := unrecoverable \/ {from}
187                     END ||
188                     resp := val ||
189                     sequenceHistory := sequenceHistory <- (from |-> to |->
                            response |-> val)
190                 END
191         END
192 END
```

D.2 B-Method Machine of the Conn iFTE

```
1 MACHINE conn
2
3 /* ================================ */
4 SETS
5         BOperations = {cliProv , cliProv_nor , cliProv_abn , provNor, provNor_nor ,
               provNor_abn , provAbn, provAbn_nor , provAbn_abn , provReq ,
              provReq_nor , provReq_abn , norReq , norReq_nor , norReq_abn , norProv ,
              norProv_nor , norProv_abn , norAbn, norAbn_nor , norAbn_abn , abnNor ,
              abnNor_nor , abnNor_abn , abnReq, abnReq_nor , abnReq_abn , abnProv ,
              abnProv_nor , abnProv_abn , reqAbn , reqAbn_nor , reqAbn_abn , reqNor ,
              reqNor_nor , reqNor_abn , reqProv , reqProv_nor , reqProv_abn , reqSrv ,
              reqSrv_nor , reqSrv_abn }; /*Each defined operation and their
              possible returns*/
6         Components = {noneCONN, clientCONN , providedCONN , normalCONN,
              abnormalCONN , requiredCONN , serverCONN};
7         Calls = {start1 , cli_prov , prov_nor , prov_abn , prov_req , nor_req ,
              nor_prov , nor_abn , abn_nor , abn_req , abn_prov , req_abn , req_nor ,
              req_prov , req_srv , prov_srv , nor_srv , abn_srv };
8         NormalReturnsCONN = {norReturnCONN};
9         AbnormalReturnsCONN = {noExceptionCONN, internalExceptionCONN1 ,
              failureExceptionCONN1 , interfaceExceptionCONN1}
10
```

```
11
12
13 /* ============================== */
14 VARIABLES
15         callListCONN,                  /*control variable*/
16         currentComponentCONN,      /*control variable*/
17         activedRequestsCONN,       /*control variable*/
18         sequenceHistoryCONN,       /*control variable*/
19         lastABNreturnCONN,             /*control variable*/
20         lastExternalABNreturnCONN,     /*control variable*/
21         unrecoverableCONN,             /*control variable*/
22         maskableExceptionsCONN,
23         failureExceptionsCONN,
24         interfaceExceptionsCONN
25
26
27
28 /* ============================== */
29 /*DEFINITIONS
30    GOAL == (...)*/
31
32
33
34 /* ============================== */
35 INVARIANT
36                 callListCONN : Components --> seq(Components) & /*for each
                        component, this variable stores the list of its callers (in
                         chronological order)*/
37                 currentComponentCONN : Components & /*the component that is
                        actually working*/
38                 activedRequestsCONN : POW(Calls) & /*a set of calls that were
                        not finished*/
39                 sequenceHistoryCONN : seq(BOperations) & /*store the sequence
                        of operations, in order to provide traceability*/
40                 maskableExceptionsCONN : POW(AbnormalReturnsCONN) & /*
                        exceptions which can be handled*/
41                 failureExceptionsCONN : POW(AbnormalReturnsCONN) & /*failure
                        exceptions*/
42                 interfaceExceptionsCONN : POW(AbnormalReturnsCONN) & /*
                        interface exceptions (can be signalled by the Provided)*/
43                 lastABNreturnCONN : AbnormalReturnsCONN & /*indicate which
                        exception had been threw by some internal element */
44                 lastExternalABNreturnCONN : AbnormalReturnsCONN & /*indicate
                        which exception had been threw by the Abnormal*/
45                 unrecoverableCONN : BOOL /*indicates if a component can recover
                         itself of a failure. In other words, if it have received
                        an exception which cannot be masked*/
46
47
48
49 /* ============================== */
50 INITIALISATION
51         callListCONN := {noneCONN |-> <>, clientCONN |-> <>, providedCONN |->
               <>, normalCONN |-> <>, abnormalCONN |-> <>, requiredCONN |-> <>,
```

```
            serverCONN |−> <>} ||
52        activedRequestsCONN := {} ||
53        currentComponentCONN := noneCONN ||
54        sequenceHistoryCONN := <> ||
55        maskableExceptionsCONN := {internalExceptionCONN1} ||
56        failureExceptionsCONN := {failureExceptionCONN1} ||
57        interfaceExceptionsCONN := {interfaceExceptionCONN1} ||
58        lastABNreturnCONN := noExceptionCONN ||
59        lastExternalABNreturnCONN := noExceptionCONN ||
60        unrecoverableCONN := FALSE
61
62
63
64
65 /* ================================ */
66 OPERATIONS
67
68 /* */
69 Start_Conn =
70        PRE
71                noneCONN = currentComponentCONN & card(activedRequestsCONN)=0
72        THEN
73                currentComponentCONN := clientCONN ||
74                activedRequestsCONN := activedRequestsCONN \/ {start1} ||
75                sequenceHistoryCONN := <> ||
76                lastABNreturnCONN := noExceptionCONN ||
77                lastExternalABNreturnCONN := noExceptionCONN ||
78                unrecoverableCONN := FALSE
79
80        END;
81
82
83
84 /* */
85 Stop_Conn =
86        PRE
87                clientCONN = currentComponentCONN & start1 :
                    activedRequestsCONN & card(activedRequestsCONN)=1
88        THEN
89                activedRequestsCONN := activedRequestsCONN − {start1} ||
90                currentComponentCONN := noneCONN ||
91                sequenceHistoryCONN := <> ||
92                skip
93        END;
94
95
96
97 /*10*/
98 ClientProv_req_Conn =
99        PRE
100               clientCONN = currentComponentCONN &
101               cli_prov /: activedRequestsCONN
102       THEN
103               currentComponentCONN := providedCONN ||
```

```
104                 activedRequestsCONN := activedRequestsCONN \/ {cli_prov} ||
105                 callListCONN(providedCONN) := callListCONN(providedCONN) <-
                        clientCONN ||
106                 sequenceHistoryCONN := [cliProv]
107         END;
108
109 resp <-- ClientProv_resp_Conn =
110         PRE
111                 size(callListCONN(providedCONN))>0 &
112                 last(callListCONN(providedCONN)) = clientCONN  &
113                 cli_prov : activedRequestsCONN  &
114                 providedCONN = currentComponentCONN
115
116         THEN
117                 activedRequestsCONN := activedRequestsCONN - {cli_prov} ||
118                 currentComponentCONN := clientCONN ||
119                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                        ||
120                 resp := norReturnCONN ||
121                 sequenceHistoryCONN := sequenceHistoryCONN <- cliProv_nor
122         END;
123
124
125 resp <-- ClientProv_ABNresp_Conn =
126         PRE
127                 size(callListCONN(providedCONN))>0 &
128                 last(callListCONN(providedCONN)) = clientCONN  &
129                 cli_prov : activedRequestsCONN  &
130                 providedCONN = currentComponentCONN
131
132         THEN
133                 activedRequestsCONN := activedRequestsCONN - {cli_prov} ||
134                 currentComponentCONN := clientCONN ||
135                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                        ||
136                 sequenceHistoryCONN := sequenceHistoryCONN <- cliProv_abn ||
137                 ANY val WHERE val : AbnormalReturnsCONN & ((val :
                        interfaceExceptionsCONN) or (val =
                        lastExternalABNreturnCONN)) & val /= noExceptionCONN
138                     THEN
139                         IF val /: maskableExceptionsCONN
140                             THEN
141                                 unrecoverableCONN := TRUE
142                         END ||
143                         resp := val ||
144                         lastABNreturnCONN := val
145                     END
146         END;
147
148
149
150
151 /*20*/
152 ProvNor_req_Conn =
```

```
153         PRE
154                 providedCONN = currentComponentCONN  &
155                 prov_nor /: activedRequestsCONN
156         THEN
157                 currentComponentCONN := normalCONN ||
158                 activedRequestsCONN := activedRequestsCONN \/ {prov_nor} ||
159                 callListCONN(normalCONN) := callListCONN(normalCONN) <-
                        providedCONN ||
160                 sequenceHistoryCONN := sequenceHistoryCONN <- provNor
161         END;
162
163 resp <-- ProvNor_resp_Conn =
164         PRE
165                 size(callListCONN(normalCONN))>0 &
166                 last(callListCONN(normalCONN)) = providedCONN  &
167                 prov_nor : activedRequestsCONN  &
168                 normalCONN = currentComponentCONN
169
170         THEN
171                 activedRequestsCONN := activedRequestsCONN - {prov_nor} ||
172                 currentComponentCONN := providedCONN ||
173                 callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
174                 resp := norReturnCONN ||
175                 sequenceHistoryCONN := sequenceHistoryCONN <- provNor_nor
176         END;
177
178
179 resp <-- ProvNor_ABNresp_Conn =
180         PRE
181                 size(callListCONN(normalCONN))>0 &
182                 last(callListCONN(normalCONN)) = providedCONN  &
183                 prov_nor : activedRequestsCONN  &
184                 normalCONN = currentComponentCONN
185
186         THEN
187                 activedRequestsCONN := activedRequestsCONN - {prov_nor} ||
188                 currentComponentCONN := providedCONN ||
189                 callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
190                 sequenceHistoryCONN := sequenceHistoryCONN <- provNor_abn ||
191                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                        noExceptionCONN & val /: interfaceExceptionsCONN
192                     THEN
193                         IF val /: maskableExceptionsCONN
194                             THEN
195                                 unrecoverableCONN := TRUE
196                         END ||
197                         resp := val ||
198                         lastABNreturnCONN := val
199                     END
200         END;
201
202
203
204
```

```
205
206 /*30*/
207 ProvAbn_req_Conn =
208         PRE
209                 providedCONN = currentComponentCONN  &
210                 prov_abn  /: activedRequestsCONN &
211                 lastABNreturnCONN /= noExceptionCONN
212         THEN
213                 currentComponentCONN := abnormalCONN ||
214                 activedRequestsCONN := activedRequestsCONN \/ {prov_abn} ||
215                 callListCONN(abnormalCONN) := callListCONN(abnormalCONN) <-
                        providedCONN ||
216                 sequenceHistoryCONN := sequenceHistoryCONN <- provAbn
217         END;
218
219 resp <-- ProvAbn_resp_Conn =
220         PRE
221                 ((lastABNreturnCONN = noExceptionCONN) or (lastABNreturnCONN :
                        maskableExceptionsCONN)) &
222                 unrecoverableCONN = FALSE &
223                 size(callListCONN(abnormalCONN))>0 &
224                 last(callListCONN(abnormalCONN)) = providedCONN  &
225                 prov_abn : activedRequestsCONN  &
226                 abnormalCONN = currentComponentCONN
227
228         THEN
229                 lastABNreturnCONN := noExceptionCONN ||
230                 activedRequestsCONN := activedRequestsCONN - {prov_abn} ||
231                 currentComponentCONN := providedCONN ||
232                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                        ||
233                 resp := norReturnCONN ||
234                 sequenceHistoryCONN := sequenceHistoryCONN <- provAbn_nor
235         END;
236
237
238 resp <-- ProvAbn_ABNresp_Conn =
239         PRE
240                 size(callListCONN(abnormalCONN))>0 &
241                 last(callListCONN(abnormalCONN)) = providedCONN  &
242                 prov_abn : activedRequestsCONN  &
243                 abnormalCONN = currentComponentCONN
244
245         THEN
246                 activedRequestsCONN := activedRequestsCONN - {prov_abn} ||
247                 currentComponentCONN := providedCONN ||
248                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                        ||
249                 sequenceHistoryCONN := sequenceHistoryCONN <- provAbn_abn ||
250                 ANY val WHERE val : AbnormalReturnsCONN & val :
                        failureExceptionsCONN & val /= noExceptionCONN & val /:
                        interfaceExceptionsCONN
251                     THEN
252                         IF val /: maskableExceptionsCONN
```

```
253                                   THEN
254                                        unrecoverableCONN := TRUE
255                              END ||
256                              resp := val ||
257                              lastExternalABNreturnCONN := val
258                         END
259           END;
260
261
262 /*40*/
263 ProvReq_req_Conn =
264           PRE
265                     providedCONN = currentComponentCONN  &
266                     prov_req /: activedRequestsCONN
267           THEN
268                     currentComponentCONN := requiredCONN ||
269                     activedRequestsCONN := activedRequestsCONN \/ {prov_req} ||
270                     callListCONN(requiredCONN) := callListCONN(requiredCONN) <-
                            providedCONN ||
271                     sequenceHistoryCONN := sequenceHistoryCONN <- provReq
272           END;
273
274 resp <-- ProvReq_resp_Conn =
275           PRE
276
277                     size(callListCONN(requiredCONN))>0 &
278                     last(callListCONN(requiredCONN)) = providedCONN  &
279                     prov_req : activedRequestsCONN  &
280                     requiredCONN = currentComponentCONN
281
282           THEN
283                     activedRequestsCONN := activedRequestsCONN - {prov_req} ||
284                     currentComponentCONN := providedCONN ||
285                     callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                              ||
286                     resp := norReturnCONN ||
287                     sequenceHistoryCONN := sequenceHistoryCONN <- provReq_nor
288           END;
289
290
291 resp <-- ProvReq_ABNresp_Conn =
292           PRE
293                     size(callListCONN(requiredCONN))>0 &
294                     last(callListCONN(requiredCONN)) = providedCONN  &
295                     prov_req : activedRequestsCONN  &
296                     requiredCONN = currentComponentCONN
297
298           THEN
299                     activedRequestsCONN := activedRequestsCONN - {prov_req} ||
300                     currentComponentCONN := providedCONN ||
301                     callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                              ||
302                     sequenceHistoryCONN := sequenceHistoryCONN <- provReq_abn ||
```

```
303                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                        noExceptionCONN & val /: interfaceExceptionsCONN
304                     THEN
305                         IF val /: maskableExceptionsCONN
306                             THEN
307                                 unrecoverableCONN := TRUE
308                         END ||
309                         resp := val ||
310                         lastABNreturnCONN := val
311                     END
312         END;
313
314
315
316
317
318 /*50*/
319 NorReq_req_Conn =
320         PRE
321                 normalCONN = currentComponentCONN &
322                 nor_req /: activedRequestsCONN
323         THEN
324                 currentComponentCONN := requiredCONN ||
325                 activedRequestsCONN := activedRequestsCONN \/ {nor_req} ||
326                 callListCONN(requiredCONN) := callListCONN(requiredCONN) <-
                        normalCONN ||
327                 sequenceHistoryCONN := sequenceHistoryCONN <- norReq
328         END;
329
330 resp <-- NorReq_resp_Conn =
331         PRE
332                 size(callListCONN(requiredCONN))>0 &
333                 last(callListCONN(requiredCONN)) = normalCONN &
334                 nor_req : activedRequestsCONN &
335                 requiredCONN = currentComponentCONN
336
337         THEN
338                 activedRequestsCONN := activedRequestsCONN - {nor_req} ||
339                 currentComponentCONN := normalCONN ||
340                 callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                        ||
341                 resp := norReturnCONN ||
342                 sequenceHistoryCONN := sequenceHistoryCONN <- norReq_nor
343         END;
344
345
346 resp <-- NorReq_ABNresp_Conn =
347         PRE
348                 size(callListCONN(requiredCONN))>0 &
349                 last(callListCONN(requiredCONN)) = normalCONN &
350                 nor_req : activedRequestsCONN &
351                 requiredCONN = currentComponentCONN
352
353         THEN
```

```
354                 activedRequestsCONN := activedRequestsCONN - {nor_req} ||
355                 currentComponentCONN := normalCONN ||
356                 callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                        ||
357                 sequenceHistoryCONN := sequenceHistoryCONN <- norReq_abn ||
358                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                    noExceptionCONN & val /: interfaceExceptionsCONN
359                     THEN
360                         IF val /: maskableExceptionsCONN
361                             THEN
362                                 unrecoverableCONN := TRUE
363                         END ||
364                         resp := val ||
365                         lastABNreturnCONN := val
366                     END
367         END;
368
369 /*60*/
370 NorProv_req_Conn =
371         PRE
372                 normalCONN = currentComponentCONN &
373                 nor_prov /: activedRequestsCONN
374         THEN
375                 currentComponentCONN := providedCONN ||
376                 activedRequestsCONN := activedRequestsCONN \/ {nor_prov} ||
377                 callListCONN(providedCONN) := callListCONN(providedCONN) <-
                    normalCONN ||
378                 sequenceHistoryCONN := sequenceHistoryCONN <- norProv
379         END;
380
381 resp <-- NorProv_resp_Conn =
382         PRE
383                 size(callListCONN(providedCONN))>0 &
384                 last(callListCONN(providedCONN)) = normalCONN &
385                 nor_prov : activedRequestsCONN &
386                 providedCONN = currentComponentCONN
387
388         THEN
389                 activedRequestsCONN := activedRequestsCONN - {nor_prov} ||
390                 currentComponentCONN := normalCONN ||
391                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                        ||
392                 resp := norReturnCONN ||
393                 sequenceHistoryCONN := sequenceHistoryCONN <- norProv_nor
394         END;
395
396
397 resp <-- NorProv_ABNresp_Conn =
398         PRE
399                 size(callListCONN(providedCONN))>0 &
400                 last(callListCONN(providedCONN)) = normalCONN &
401                 nor_prov : activedRequestsCONN &
402                 providedCONN = currentComponentCONN
403
```

```
404         THEN
405                 activedRequestsCONN := activedRequestsCONN - {nor_prov} ||
406                 currentComponentCONN := normalCONN ||
407                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                        ||
408                 sequenceHistoryCONN := sequenceHistoryCONN <- norProv_abn ||
409                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                    noExceptionCONN
410                     THEN
411                         IF val /: maskableExceptionsCONN
412                             THEN
413                                 unrecoverableCONN := TRUE
414                         END ||
415                         resp := val ||
416                         lastABNreturnCONN := val
417                     END
418         END;
419
420
421 /*70*/
422 NorAbn_req_Conn =
423         PRE
424                 normalCONN = currentComponentCONN &
425                 nor_abn /: activedRequestsCONN
426         THEN
427                 currentComponentCONN := abnormalCONN ||
428                 activedRequestsCONN := activedRequestsCONN \/ {nor_abn} ||
429                 callListCONN(abnormalCONN) := callListCONN(abnormalCONN) <-
                    normalCONN ||
430                 sequenceHistoryCONN := sequenceHistoryCONN <- norAbn
431         END;
432
433 resp <-- NorAbn_resp_Conn =
434         PRE
435                 ((lastABNreturnCONN = noExceptionCONN) or (lastABNreturnCONN :
                    maskableExceptionsCONN)) &
436                 unrecoverableCONN = FALSE &
437                 size(callListCONN(abnormalCONN))>0 &
438                 last(callListCONN(abnormalCONN)) = normalCONN &
439                 nor_abn : activedRequestsCONN &
440                 abnormalCONN = currentComponentCONN
441
442         THEN
443                 activedRequestsCONN := activedRequestsCONN - {nor_abn} ||
444                 currentComponentCONN := normalCONN ||
445                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                        ||
446                 resp := norReturnCONN ||
447                 sequenceHistoryCONN := sequenceHistoryCONN <- norAbn_nor
448         END;
449
450
451 resp <-- NorAbn_ABNresp_Conn =
452         PRE
```

```
453                 size(callListCONN(abnormalCONN))>0 &
454                 last(callListCONN(abnormalCONN)) = normalCONN  &
455                 nor_abn : activedRequestsCONN  &
456                 abnormalCONN = currentComponentCONN
457
458         THEN
459                 activedRequestsCONN := activedRequestsCONN - {nor_abn} ||
460                 currentComponentCONN := normalCONN ||
461                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                        ||
462                 sequenceHistoryCONN := sequenceHistoryCONN <- norAbn_abn ||
463                 ANY val WHERE val : AbnormalReturnsCONN & val :
                        failureExceptionsCONN & val /= noExceptionCONN & val /:
                        interfaceExceptionsCONN
464                     THEN
465                         IF val /: maskableExceptionsCONN
466                             THEN
467                                 unrecoverableCONN := TRUE
468                         END ||
469                         resp := val ||
470                         lastExternalABNreturnCONN := val
471                     END
472         END;
473
474
475
476 /*80*/
477 AbnNor_req_Conn =
478         PRE
479                 abnormalCONN = currentComponentCONN  &
480                 abn_nor /: activedRequestsCONN
481         THEN
482                 currentComponentCONN := normalCONN ||
483                 activedRequestsCONN := activedRequestsCONN \/ {abn_nor} ||
484                 callListCONN(normalCONN) := callListCONN(normalCONN) <-
                        abnormalCONN ||
485                 sequenceHistoryCONN := sequenceHistoryCONN <- abnNor
486         END;
487
488 resp <-- AbnNor_resp_Conn =
489         PRE
490                 size(callListCONN(normalCONN))>0 &
491                 last(callListCONN(normalCONN)) = abnormalCONN  &
492                 abn_nor : activedRequestsCONN  &
493                 normalCONN = currentComponentCONN
494
495         THEN
496                 faultMasked := TRUE ||
497                 activedRequestsCONN := activedRequestsCONN - {abn_nor} ||
498                 currentComponentCONN := abnormalCONN ||
499                 callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
500                 resp :=  norReturnCONN ||
501                 sequenceHistoryCONN := sequenceHistoryCONN <- abnNor_nor
502         END;
```

```
503
504
505
506
507
508 resp <-- AbnNor_ABNresp_Conn =
509         PRE
510                 size(callListCONN(normalCONN))>0 &
511                 last(callListCONN(normalCONN)) = abnormalCONN  &
512                 abn_nor : activedRequestsCONN  &
513                 normalCONN = currentComponentCONN
514
515         THEN
516                 faultMasked := FALSE ||
517                 activedRequestsCONN := activedRequestsCONN - {abn_nor} ||
518                 currentComponentCONN := abnormalCONN ||
519                 callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
520                 sequenceHistoryCONN := sequenceHistoryCONN <- abnNor_abn ||
521                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                        noExceptionCONN & val /: interfaceExceptionsCONN
522                     THEN
523                         IF val /: maskableExceptionsCONN
524                             THEN
525                                 unrecoverableCONN := TRUE
526                         END ||
527                         resp := val ||
528                         lastABNreturnCONN := val
529                     END
530         END;
531
532
533
534
535 /*90*/
536 AbnReq_req_Conn =
537         PRE
538                 abnormalCONN = currentComponentCONN  &
539                 abn_req /: activedRequestsCONN
540         THEN
541                 currentComponentCONN := requiredCONN ||
542                 activedRequestsCONN := activedRequestsCONN \/ {abn_req} ||
543                 callListCONN(requiredCONN) := callListCONN(requiredCONN) <-
                        abnormalCONN ||
544                 sequenceHistoryCONN := sequenceHistoryCONN <- abnReq
545         END;
546
547 resp <-- AbnReq_resp_Conn =
548         PRE
549                 size(callListCONN(requiredCONN))>0 &
550                 last(callListCONN(requiredCONN)) = abnormalCONN  &
551                 abn_req : activedRequestsCONN  &
552                 requiredCONN = currentComponentCONN
553
554         THEN
```

```
555                 activedRequestsCONN := activedRequestsCONN - {abn_req} ||
556                 currentComponentCONN := abnormalCONN ||
557                 callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                        ||
558                 resp := norReturnCONN ||
559                 sequenceHistoryCONN := sequenceHistoryCONN <- abnReq_nor
560         END;
561
562
563 resp <-- AbnReq_ABNresp_Conn =
564         PRE
565                 size(callListCONN(requiredCONN))>0 &
566                 last(callListCONN(requiredCONN)) = abnormalCONN &
567                 abn_req : activedRequestsCONN &
568                 requiredCONN = currentComponentCONN
569
570         THEN
571                 activedRequestsCONN := activedRequestsCONN - {abn_req} ||
572                 currentComponentCONN := abnormalCONN ||
573                 callListCONN(requiredCONN) := front(callListCONN(requiredCONN))
                        ||
574                 sequenceHistoryCONN := sequenceHistoryCONN <- abnReq_abn ||
575                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                    noExceptionCONN & val /: interfaceExceptionsCONN
576                     THEN
577                         IF val /: maskableExceptionsCONN
578                             THEN
579                                 unrecoverableCONN := TRUE
580                         END ||
581                         resp := val ||
582                         lastABNreturnCONN := val
583                     END
584         END;
585
586
587 /*100*/
588 AbnProv_req_Conn =
589         PRE
590                 abnormalCONN = currentComponentCONN &
591                 abn_prov /: activedRequestsCONN
592         THEN
593                 currentComponentCONN := providedCONN ||
594                 activedRequestsCONN := activedRequestsCONN \/ {abn_prov} ||
595                 callListCONN(providedCONN) := callListCONN(providedCONN) <-
                    abnormalCONN ||
596                 sequenceHistoryCONN := sequenceHistoryCONN <- abnProv
597         END;
598
599 resp <-- AbnProv_resp_Conn =
600         PRE
601                 size(callListCONN(providedCONN))>0 &
602                 last(callListCONN(providedCONN)) = abnormalCONN &
603                 abn_prov : activedRequestsCONN &
604                 providedCONN = currentComponentCONN
```

```
605
606         THEN
607                 activedRequestsCONN := activedRequestsCONN - {abn_prov} ||
608                 currentComponentCONN := abnormalCONN ||
609                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                         ||
610                 resp := norReturnCONN ||
611                 sequenceHistoryCONN := sequenceHistoryCONN <- abnProv_nor
612         END;
613
614
615 resp <-- AbnProv_ABNresp_Conn =
616         PRE
617                 size(callListCONN(providedCONN))>0 &
618                 last(callListCONN(providedCONN)) = abnormalCONN  &
619                 abn_prov : activedRequestsCONN  &
620                 providedCONN = currentComponentCONN
621
622         THEN
623                 activedRequestsCONN := activedRequestsCONN - {abn_prov} ||
624                 currentComponentCONN := abnormalCONN ||
625                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                         ||
626                 sequenceHistoryCONN := sequenceHistoryCONN <- abnProv_abn ||
627                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                     noExceptionCONN
628                     THEN
629                         IF val /: maskableExceptionsCONN
630                             THEN
631                                 unrecoverableCONN := TRUE
632                         END ||
633                         resp := val ||
634                         lastABNreturnCONN := val
635                     END
636         END;
637
638
639
640
641 /*110*/
642 ReqAbn_req_Conn =
643         PRE
644                 requiredCONN = currentComponentCONN  &
645                 req_abn /: activedRequestsCONN
646         THEN
647                 currentComponentCONN := abnormalCONN ||
648                 activedRequestsCONN := activedRequestsCONN \/ {req_abn} ||
649                 callListCONN(abnormalCONN) := callListCONN(abnormalCONN) <-
                     requiredCONN ||
650                 sequenceHistoryCONN := sequenceHistoryCONN <- reqAbn
651         END;
652
653 resp <-- ReqAbn_resp_Conn =
654         PRE
```

```
655                 ((lastABNreturnCONN = noExceptionCONN) or (lastABNreturnCONN :
                        maskableExceptionsCONN)) &
656                 unrecoverableCONN = FALSE &
657                 size(callListCONN(abnormalCONN))>0 &
658                 last(callListCONN(abnormalCONN)) = requiredCONN  &
659                 req_abn : activedRequestsCONN  &
660                 abnormalCONN = currentComponentCONN
661
662         THEN
663                 lastABNreturnCONN := noExceptionCONN ||
664                 activedRequestsCONN := activedRequestsCONN - {req_abn} ||
665                 currentComponentCONN := requiredCONN ||
666                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                         ||
667                 resp := norReturnCONN ||
668                 sequenceHistoryCONN := sequenceHistoryCONN <- reqAbn_nor
669         END;
670
671
672 resp <-- ReqAbn_ABNresp_Conn =
673         PRE
674                 size(callListCONN(abnormalCONN))>0 &
675                 last(callListCONN(abnormalCONN)) = requiredCONN  &
676                 req_abn : activedRequestsCONN  &
677                 abnormalCONN = currentComponentCONN
678
679         THEN
680                 activedRequestsCONN := activedRequestsCONN - {req_abn} ||
681                 currentComponentCONN := requiredCONN ||
682                 callListCONN(abnormalCONN) := front(callListCONN(abnormalCONN))
                         ||
683                 sequenceHistoryCONN := sequenceHistoryCONN <- reqAbn_abn ||
684                 ANY val WHERE val : AbnormalReturnsCONN & val :
                        failureExceptionsCONN & val /= noExceptionCONN & val /:
                        interfaceExceptionsCONN
685                         THEN
686                             IF val /: maskableExceptionsCONN
687                                 THEN
688                                     unrecoverableCONN := TRUE
689                             END ||
690                             resp := val ||
691                             lastExternalABNreturnCONN := val
692                         END
693         END;
694
695
696 /*120*/
697 ReqNor_req_Conn =
698         PRE
699                 requiredCONN = currentComponentCONN  &
700                 req_nor /: activedRequestsCONN
701         THEN
702                 currentComponentCONN := normalCONN ||
703                 activedRequestsCONN := activedRequestsCONN \/ {req_nor} ||
```

```
704                     callListCONN(normalCONN) := callListCONN(normalCONN) <-
                            requiredCONN ||
705                     sequenceHistoryCONN := sequenceHistoryCONN <- reqNor
706         END;
707
708 resp <-- ReqNor_resp_Conn =
709         PRE
710                     size(callListCONN(normalCONN))>0 &
711                     last(callListCONN(normalCONN)) = requiredCONN  &
712                     req_nor : activedRequestsCONN  &
713                     normalCONN = currentComponentCONN
714
715         THEN
716                     activedRequestsCONN := activedRequestsCONN - {req_nor} ||
717                     currentComponentCONN := requiredCONN ||
718                     callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
719                     resp := norReturnCONN ||
720                     sequenceHistoryCONN := sequenceHistoryCONN <- reqNor_nor
721         END;
722
723
724
725
726
727 resp <-- ReqNor_ABNresp_Conn =
728         PRE
729                     size(callListCONN(normalCONN))>0 &
730                     last(callListCONN(normalCONN)) = requiredCONN  &
731                     req_nor : activedRequestsCONN  &
732                     normalCONN = currentComponentCONN
733
734         THEN
735                     activedRequestsCONN := activedRequestsCONN - {req_nor} ||
736                     currentComponentCONN := requiredCONN ||
737                     callListCONN(normalCONN) := front(callListCONN(normalCONN)) ||
738                     sequenceHistoryCONN := sequenceHistoryCONN <- reqNor_abn ||
739                     ANY val WHERE val : AbnormalReturnsCONN & val /=
                            noExceptionCONN & val /: interfaceExceptionsCONN
740                         THEN
741                             IF val /: maskableExceptionsCONN
742                                 THEN
743                                     unrecoverableCONN := TRUE
744                             END ||
745                             resp := val ||
746                             lastABNreturnCONN := val
747                         END
748         END;
749
750
751 /*130*/
752 ReqProv_req_Conn =
753         PRE
754                     requiredCONN = currentComponentCONN  &
755                     req_prov /: activedRequestsCONN
```

```
756         THEN
757                 currentComponentCONN := providedCONN ||
758                 activedRequestsCONN := activedRequestsCONN \/ {req_prov} ||
759                 callListCONN(providedCONN) := callListCONN(providedCONN) <-
                        requiredCONN ||
760                 sequenceHistoryCONN := sequenceHistoryCONN <- reqProv
761         END;
762
763 resp <-- ReqProv_resp_Conn =
764         PRE
765                 size(callListCONN(providedCONN))>0 &
766                 last(callListCONN(providedCONN)) = requiredCONN &
767                 req_prov : activedRequestsCONN &
768                 providedCONN = currentComponentCONN
769
770         THEN
771                 activedRequestsCONN := activedRequestsCONN - {req_prov} ||
772                 currentComponentCONN := requiredCONN ||
773                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                         ||
774                 resp := norReturnCONN ||
775                 sequenceHistoryCONN := sequenceHistoryCONN <- reqProv_nor
776         END;
777
778
779 resp <-- ReqProv_ABNresp_Conn =
780         PRE
781                 size(callListCONN(providedCONN))>0 &
782                 last(callListCONN(providedCONN)) = requiredCONN &
783                 req_prov : activedRequestsCONN &
784                 providedCONN = currentComponentCONN
785
786         THEN
787                 activedRequestsCONN := activedRequestsCONN - {req_prov} ||
788                 currentComponentCONN := requiredCONN ||
789                 callListCONN(providedCONN) := front(callListCONN(providedCONN))
                         ||
790                 sequenceHistoryCONN := sequenceHistoryCONN <- reqProv_abn ||
791                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                        noExceptionCONN
792                     THEN
793                         IF val /: maskableExceptionsCONN
794                             THEN
795                                 unrecoverableCONN := TRUE
796                         END ||
797                         resp := val ||
798                         lastABNreturnCONN := val
799                     END
800         END;
801
802
803
804
805 /*140*/
```

```
806 ReqServer_req_Conn =
807         PRE
808                 requiredCONN = currentComponentCONN  &
809                 req_srv  /: activedRequestsCONN
810         THEN
811                 currentComponentCONN := serverCONN ||
812                 activedRequestsCONN := activedRequestsCONN \/ {req_srv} ||
813                 callListCONN(serverCONN) := callListCONN(serverCONN) <-
                        requiredCONN ||
814                 sequenceHistoryCONN := sequenceHistoryCONN <- reqSrv
815         END;
816
817 resp <-- ReqServer_resp_Conn =
818         PRE
819                 size(callListCONN(serverCONN))>0 &
820                 last(callListCONN(serverCONN)) = requiredCONN  &
821                 req_srv : activedRequestsCONN  &
822                 serverCONN = currentComponentCONN
823
824         THEN
825                 activedRequestsCONN := activedRequestsCONN - {req_srv} ||
826                 currentComponentCONN := requiredCONN ||
827                 callListCONN(serverCONN) := front(callListCONN(serverCONN)) ||
828                 resp := norReturnCONN ||
829                 sequenceHistoryCONN := sequenceHistoryCONN <- reqSrv_nor
830         END;
831
832
833 resp <-- ReqServer_resp_Conn =
834         PRE
835                 size(callListCONN(serverCONN))>0 &
836                 last(callListCONN(serverCONN)) = requiredCONN  &
837                 req_srv : activedRequestsCONN  &
838                 serverCONN = currentComponentCONN
839
840         THEN
841                 activedRequestsCONN := activedRequestsCONN - {req_srv} ||
842                 currentComponentCONN := requiredCONN ||
843                 callListCONN(serverCONN) := front(callListCONN(serverCONN)) ||
844                 sequenceHistoryCONN := sequenceHistoryCONN <- reqSrv_abn ||
845                 ANY val WHERE val : AbnormalReturnsCONN & val /=
                        noExceptionCONN & val /: interfaceExceptionsCONN
846                     THEN
847                         IF val /: maskableExceptionsCONN
848                             THEN
849                                 unrecoverableCONN := TRUE
850                         END ||
851                         resp := val ||
852                         lastABNreturnCONN := val
853                     END
854         END
855
856 END
```

D.3 B-Method Machine of the `Server` iFTE

```
1  MACHINE serverSERVER
2
3  /* ================================= */
4  SETS
5        BOperationsSERVER = {cliProv2 , cliProv2_nor , cliProv2_abn , provNor2 ,
             provNor2_nor , provNor2_abn , provAbn2, provAbn2_nor , provAbn2_abn ,
             provReq2, provReq2_nor , provReq2_abn , norReq2, norReq2_nor ,
             norReq2_abn , norProv2 , norProv2_nor , norProv2_abn , norAbn2,
             norAbn2_nor , norAbn2_abn , abnNor2, abnNor2_nor , abnNor2_abn ,
             abnReq2, abnReq2_nor , abnReq2_abn , abnProv2, abnProv2_nor ,
             abnProv2_abn , reqAbn2, reqAbn2_nor , reqAbn2_abn , reqNor2 ,
             reqNor2_nor , reqNor2_abn , reqProv2 , reqProv2_nor , reqProv2_abn ,
             reqSrv2 , reqSrv2_nor , reqSrv2_abn }; /*Each defined operation and
             their possible returns*/
6        ComponentsSERVER = {noneSERVER, clientSERVER, providedSERVER,
             normalSERVER, abnormalSERVER, requiredSERVER, serverSERVER};
7        CallsSERVER = {begin2 , cli_prov2 , prov_nor2 , prov_abn2 , prov_req2 ,
             nor_req2 , nor_prov2 , nor_abn2 , abn_nor2 , abn_req2 , abn_prov2 ,
             req_abn2 , req_nor2 , req_prov2 , req_srv2 };
8        NormalReturnsSERVER = {norReturnSERVER};
9        AbnormalReturnsSERVER = {noExceptionSERVER, internalExceptionSERVER,
             failureExceptionSERVER , interfaceExceptionSERVER}
10
11
12
13 /* ================================= */
14 VARIABLES
15       callListSERVER ,              /*control variable*/
16       currentComponentSERVER,/*control variable*/
17       activedRequestsSERVER ,     /*control variable*/
18       sequenceHistorySERVER, /*control variable*/
19       lastABNreturnSERVER,          /*control variable*/
20       lastExternalABNreturnSERVER, /*control variable*/
21       unrecoverableSERVER ,         /*control variable*/
22       maskableExceptionsSERVER ,
23       failureExceptionsSERVER ,
24       interfaceExceptionsSERVER
25
26
27
28 /* ================================= */
29 /*DEFINITIONS
30    GOAL == (...)*/
31
32
33
34 /* ================================= */
35 INVARIANT
36             callListSERVER : ComponentsSERVER --> seq(ComponentsSERVER) &
                   /*for each component, this variable stores the list of its
                   callers (in chronological order)*/
```

```
37                  currentComponentSERVER : ComponentsSERVER & /*the component
                        that is actually working*/
38                  activedRequestsSERVER : POW(CallsSERVER) & /*a set of
                        CallsSERVER that were not finished*/
39                  sequenceHistorySERVER : seq(BOperationsSERVER) &/*store the
                        sequence of operations*/
40                  maskableExceptionsSERVER : POW(AbnormalReturnsSERVER) & /*
                        exceptions which can be handled*/
41                  failureExceptionsSERVER : POW(AbnormalReturnsSERVER) & /*
                        failure exceptions*/
42                  interfaceExceptionsSERVER : POW(AbnormalReturnsSERVER) & /*
                        interface exceptions (can be signalled by the Provided)*/
43                  lastABNreturnSERVER : AbnormalReturnsSERVER & /*indicate which
                        exception had been threw by some internal element */
44                  lastExternalABNreturnSERVER : AbnormalReturnsSERVER & /*
                        indicate which exception had been threw by the Abnormal*/
45                  unrecoverableSERVER : BOOL /*indicates if a component can
                        recover itself of a failure. In other words, if it have
                        received an exception which cannot be masked*/
46
47
48
49
50
51 /* ================================= */
52 INITIALISATION
53          callListSERVER := {noneSERVER |-> <>, clientSERVER |-> <>,
                providedSERVER |-> <>, normalSERVER |-> <>, abnormalSERVER |-> <>,
                requiredSERVER |-> <>, serverSERVER |-> <>} ||
54          activedRequestsSERVER := {} ||
55          currentComponentSERVER := noneSERVER ||
56          sequenceHistorySERVER := <> ||
57          maskableExceptionsSERVER := {internalExceptionSERVER} ||
58          failureExceptionsSERVER := {failureExceptionSERVER} ||
59          interfaceExceptionsSERVER := {interfaceExceptionSERVER} ||
60          lastABNreturnSERVER := noExceptionSERVER ||
61          lastExternalABNreturnSERVER := noExceptionSERVER ||
62          unrecoverableSERVER := FALSE
63
64
65
66
67 /* ================================= */
68 OPERATIONS
69
70 /* */
71 Start_Server =
72          PRE
73                  noneSERVER = currentComponentSERVER & card(
                        activedRequestsSERVER)=0
74          THEN
75                  currentComponentSERVER := clientSERVER ||
76                  activedRequestsSERVER := activedRequestsSERVER \/ {begin2} ||
77                  sequenceHistorySERVER := <> ||
```

```
                lastABNreturnSERVER := noExceptionSERVER ||
                lastExternalABNreturnSERVER := noExceptionSERVER ||
                unrecoverableSERVER := FALSE
        END;



/* */
Stop_Server =
        PRE
                clientSERVER = currentComponentSERVER & begin2 :
                    activedRequestsSERVER & card(activedRequestsSERVER)=1
        THEN
                activedRequestsSERVER := activedRequestsSERVER - {begin2} ||
                currentComponentSERVER := noneSERVER ||
                sequenceHistorySERVER := <> ||
                skip
        END;



/*10*/
ClientProv_req_Server =
        PRE
                clientSERVER = currentComponentSERVER &
                cli_prov2 /: activedRequestsSERVER
        THEN
                currentComponentSERVER := providedSERVER ||
                activedRequestsSERVER := activedRequestsSERVER \/ {cli_prov2}
                    ||
                callListSERVER(providedSERVER) := callListSERVER(providedSERVER
                    ) <- clientSERVER ||
                sequenceHistorySERVER := [cliProv2]
        END;

resp2 <-- ClientProv_resp_Server =
        PRE
                size(callListSERVER(providedSERVER))>0 &
                last(callListSERVER(providedSERVER)) = clientSERVER &
                cli_prov2 : activedRequestsSERVER &
                providedSERVER = currentComponentSERVER

        THEN
                activedRequestsSERVER := activedRequestsSERVER - {cli_prov2} ||
                currentComponentSERVER := clientSERVER ||
                callListSERVER(providedSERVER) := front(callListSERVER(
                    providedSERVER)) ||
                resp2 := norReturnSERVER ||
                sequenceHistorySERVER := sequenceHistorySERVER <- cliProv2_nor
        END;


resp2 <-- ClientProv_ABNresp_Server =
        PRE
```

```
128                 size(callListSERVER(providedSERVER))>0 &
129                 last(callListSERVER(providedSERVER)) = clientSERVER  &
130                 cli_prov2 : activedRequestsSERVER  &
131                 providedSERVER = currentComponentSERVER
132
133         THEN
134                 activedRequestsSERVER := activedRequestsSERVER - {cli_prov2} ||
135                 currentComponentSERVER := clientSERVER ||
136                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
137                 sequenceHistorySERVER := sequenceHistorySERVER <- cliProv2_abn
                        ||
138                 ANY val WHERE val : AbnormalReturnsSERVER & ((val :
                        interfaceExceptionsSERVER) or (val =
                        lastExternalABNreturnSERVER)) & val /= noExceptionSERVER
139                     THEN
140                         IF val /: maskableExceptionsSERVER
141                             THEN
142                                 unrecoverableSERVER := TRUE
143                         END ||
144                         resp2 := val ||
145                         lastABNreturnSERVER := val
146                     END
147         END;
148
149
150
151
152 /*20*/
153 ProvNor_req_Server =
154         PRE
155                 providedSERVER = currentComponentSERVER  &
156                 prov_nor2 /: activedRequestsSERVER
157         THEN
158                 currentComponentSERVER := normalSERVER ||
159                 activedRequestsSERVER := activedRequestsSERVER \/ {prov_nor2}
                        ||
160                 callListSERVER(normalSERVER) := callListSERVER(normalSERVER) <-
                         providedSERVER ||
161                 sequenceHistorySERVER := sequenceHistorySERVER <- provNor2
162         END;
163
164 resp2 <-- ProvNor_resp_Server =
165         PRE
166                 size(callListSERVER(normalSERVER))>0 &
167                 last(callListSERVER(normalSERVER)) = providedSERVER  &
168                 prov_nor2 : activedRequestsSERVER  &
169                 normalSERVER = currentComponentSERVER
170
171         THEN
172                 activedRequestsSERVER := activedRequestsSERVER - {prov_nor2} ||
173                 currentComponentSERVER := providedSERVER ||
174                 callListSERVER(normalSERVER) := front(callListSERVER(
                        normalSERVER)) ||
```

```
175                 resp2 := norReturnSERVER ||
176                 sequenceHistorySERVER := sequenceHistorySERVER <- provNor2_nor
177         END;
178
179
180 resp2 <-- ProvNor_intABNresp_Server =
181         PRE
182                 size(callListSERVER(normalSERVER))>0 &
183                 last(callListSERVER(normalSERVER)) = providedSERVER  &
184                 prov_nor2 : activedRequestsSERVER  &
185                 normalSERVER = currentComponentSERVER
186
187         THEN
188                 activedRequestsSERVER := activedRequestsSERVER - {prov_nor2} ||
189                 currentComponentSERVER := providedSERVER ||
190                 callListSERVER(normalSERVER) := front(callListSERVER(
                        normalSERVER)) ||
191                 sequenceHistorySERVER := sequenceHistorySERVER <-
                        provNor2_intAbn ||
192                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER & val /: interfaceExceptionsSERVER
193                     THEN
194                         IF val /: maskableExceptionsSERVER
195                             THEN
196                                 unrecoverableSERVER := TRUE
197                         END ||
198                         resp2 := val ||
199                         lastABNreturnSERVER := val
200                     END
201         END;
202
203
204
205
206
207 /*30*/
208 ProvAbn_req_Server =
209         PRE
210                 providedSERVER = currentComponentSERVER  &
211                 prov_abn2 /: activedRequestsSERVER
212         THEN
213                 currentComponentSERVER := abnormalSERVER ||
214                 activedRequestsSERVER := activedRequestsSERVER \/ {prov_abn2}
                        ||
215                 callListSERVER(abnormalSERVER) := callListSERVER(abnormalSERVER
                        ) <- providedSERVER ||
216                 sequenceHistorySERVER := sequenceHistorySERVER <- provAbn2
217         END;
218
219 resp2 <-- ProvAbn_resp_Server =
220         PRE
221                 ((lastABNreturnSERVER = noExceptionSERVER) or (
                        lastABNreturnSERVER : maskableExceptionsSERVER)) &
222                 unrecoverableSERVER = FALSE &
```

```
223                     size(callListSERVER(abnormalSERVER))>0 &
224                     last(callListSERVER(abnormalSERVER)) = providedSERVER  &
225                     prov_abn2 : activedRequestsSERVER  &
226                     abnormalSERVER = currentComponentSERVER
227
228          THEN
229                     lastABNreturnSERVER := noExceptionSERVER ||
230                     activedRequestsSERVER := activedRequestsSERVER - {prov_abn2} ||
231                     currentComponentSERVER := providedSERVER ||
232                     callListSERVER(abnormalSERVER) := front(callListSERVER(
                            abnormalSERVER)) ||
233                     resp2 := norReturnSERVER ||
234                     sequenceHistorySERVER := sequenceHistorySERVER <- provAbn2_nor
235          END;
236
237
238 resp2 <-- ProvAbn_ABNresp_Server =
239          PRE
240                     size(callListSERVER(abnormalSERVER))>0 &
241                     last(callListSERVER(abnormalSERVER)) = providedSERVER  &
242                     prov_abn2 : activedRequestsSERVER  &
243                     abnormalSERVER = currentComponentSERVER
244
245          THEN
246                     activedRequestsSERVER := activedRequestsSERVER - {prov_abn2} ||
247                     currentComponentSERVER := providedSERVER ||
248                     callListSERVER(abnormalSERVER) := front(callListSERVER(
                            abnormalSERVER)) ||
249                     sequenceHistorySERVER := sequenceHistorySERVER <- provAbn2_abn
                            ||
250                    ANY val WHERE val : AbnormalReturnsSERVER & val :
                            failureExceptionsSERVER & val /= noExceptionSERVER & val /:
                             interfaceExceptionsSERVER
251                          THEN
252                              IF val /: maskableExceptionsSERVER
253                                  THEN
254                                      unrecoverableSERVER := TRUE
255                              END ||
256                              resp2 := val ||
257                              lastExternalABNreturnSERVER := val
258                          END
259          END;
260
261
262 /*40*/
263 ProvReq_req_Server =
264          PRE
265                     providedSERVER = currentComponentSERVER  &
266                     prov_req2 /: activedRequestsSERVER
267          THEN
268                     currentComponentSERVER := requiredSERVER ||
269                     activedRequestsSERVER := activedRequestsSERVER \/ {prov_req2}
                            ||
```

```
270                 callListSERVER(requiredSERVER) := callListSERVER(requiredSERVER
                        ) <- providedSERVER ||
271                 sequenceHistorySERVER := sequenceHistorySERVER <- provReq2
272         END;
273 
274 resp2 <-- ProvReq_resp_Server =
275         PRE
276 
277                 size(callListSERVER(requiredSERVER))>0 &
278                 last(callListSERVER(requiredSERVER)) = providedSERVER  &
279                 prov_req2 : activedRequestsSERVER  &
280                 requiredSERVER = currentComponentSERVER
281 
282         THEN
283                 activedRequestsSERVER := activedRequestsSERVER - {prov_req2} ||
284                 currentComponentSERVER := providedSERVER ||
285                 callListSERVER(requiredSERVER) := front(callListSERVER(
                        requiredSERVER)) ||
286                 resp2 := norReturnSERVER ||
287                 sequenceHistorySERVER := sequenceHistorySERVER <- provReq2_nor
288         END;
289 
290 
291 resp2 <-- ProvReq_ABNresp_Server =
292         PRE
293                 size(callListSERVER(requiredSERVER))>0 &
294                 last(callListSERVER(requiredSERVER)) = providedSERVER  &
295                 prov_req2 : activedRequestsSERVER  &
296                 requiredSERVER = currentComponentSERVER
297 
298         THEN
299                 activedRequestsSERVER := activedRequestsSERVER - {prov_req2} ||
300                 currentComponentSERVER := providedSERVER ||
301                 callListSERVER(requiredSERVER) := front(callListSERVER(
                        requiredSERVER)) ||
302                 sequenceHistorySERVER := sequenceHistorySERVER <- provReq2_abn
                        ||
303                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER & val /: interfaceExceptionsSERVER
304                     THEN
305                         IF val /: maskableExceptionsSERVER
306                             THEN
307                                 unrecoverableSERVER := TRUE
308                         END ||
309                         resp2 := val ||
310                         lastABNreturnSERVER := val
311                     END
312         END;
313 
314 
315 
316 
317 
318 /*50*/
```

```
319 NorReq_req_Server =
320         PRE
321                 normalSERVER = currentComponentSERVER  &
322                 nor_req2 /: activedRequestsSERVER
323         THEN
324                 currentComponentSERVER := requiredSERVER ||
325                 activedRequestsSERVER := activedRequestsSERVER \/ {nor_req2} ||
326                 callListSERVER(requiredSERVER) := callListSERVER(requiredSERVER
                        ) <- normalSERVER ||
327                 sequenceHistorySERVER := sequenceHistorySERVER <- norReq2
328         END;
329
330 resp2 <-- NorReq_resp_Server =
331         PRE
332                 size(callListSERVER(requiredSERVER))>0 &
333                 last(callListSERVER(requiredSERVER)) = normalSERVER  &
334                 nor_req2 : activedRequestsSERVER  &
335                 requiredSERVER = currentComponentSERVER
336
337         THEN
338                 activedRequestsSERVER := activedRequestsSERVER - {nor_req2} ||
339                 currentComponentSERVER := normalSERVER ||
340                 callListSERVER(requiredSERVER) := front(callListSERVER(
                        requiredSERVER)) ||
341                 resp2 := norReturnSERVER ||
342                 sequenceHistorySERVER := sequenceHistorySERVER <- norReq2_nor
343         END;
344
345
346 resp2 <-- NorReq_ABNresp_Server =
347         PRE
348                 size(callListSERVER(requiredSERVER))>0 &
349                 last(callListSERVER(requiredSERVER)) = normalSERVER  &
350                 nor_req2 : activedRequestsSERVER  &
351                 requiredSERVER = currentComponentSERVER
352
353         THEN
354                 activedRequestsSERVER := activedRequestsSERVER - {nor_req2} ||
355                 currentComponentSERVER := normalSERVER ||
356                 callListSERVER(requiredSERVER) := front(callListSERVER(
                        requiredSERVER)) ||
357                 sequenceHistorySERVER := sequenceHistorySERVER <- norReq2_abn
                        ||
358                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER & val /: interfaceExceptionsSERVER
359                     THEN
360                         IF val /: maskableExceptionsSERVER
361                             THEN
362                                 unrecoverableSERVER := TRUE
363                         END ||
364                         resp2 := val ||
365                         lastABNreturnSERVER := val
366                     END
367         END;
```

```
368
369 /*60*/
370 NorProv_req_Server =
371         PRE
372                 normalSERVER = currentComponentSERVER  &
373                 nor_prov2 /: activedRequestsSERVER
374         THEN
375                 currentComponentSERVER := providedSERVER ||
376                 activedRequestsSERVER := activedRequestsSERVER \/ {nor_prov2}
                        ||
377                 callListSERVER(providedSERVER) := callListSERVER(providedSERVER
                        ) <- normalSERVER ||
378                 sequenceHistorySERVER := sequenceHistorySERVER <- norProv2
379         END;
380
381 resp2 <-- NorProv_resp_Server =
382         PRE
383                 size(callListSERVER(providedSERVER))>0 &
384                 last(callListSERVER(providedSERVER)) = normalSERVER  &
385                 nor_prov2 : activedRequestsSERVER  &
386                 providedSERVER = currentComponentSERVER
387
388         THEN
389                 activedRequestsSERVER := activedRequestsSERVER - {nor_prov2} ||
390                 currentComponentSERVER := normalSERVER ||
391                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
392                 resp2 := norReturnSERVER ||
393                 sequenceHistorySERVER := sequenceHistorySERVER <- norProv2_nor
394         END;
395
396
397 resp2 <-- NorProv_ABNresp_Server =
398         PRE
399                 size(callListSERVER(providedSERVER))>0 &
400                 last(callListSERVER(providedSERVER)) = normalSERVER  &
401                 nor_prov2 : activedRequestsSERVER  &
402                 providedSERVER = currentComponentSERVER
403
404         THEN
405                 activedRequestsSERVER := activedRequestsSERVER - {nor_prov2} ||
406                 currentComponentSERVER := normalSERVER ||
407                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
408                 sequenceHistorySERVER := sequenceHistorySERVER <- norProv2_abn
                        ||
409                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER
410                     THEN
411                         IF val /: maskableExceptionsSERVER
412                             THEN
413                                 unrecoverableSERVER := TRUE
414                         END ||
415                         resp2 := val ||
```

```
416                         lastABNreturnSERVER := val
417                     END
418         END;
419
420
421 /*70*/
422 NorAbn_req_Server =
423         PRE
424                 normalSERVER = currentComponentSERVER  &
425                 nor_abn2 /: activedRequestsSERVER
426         THEN
427                 currentComponentSERVER := abnormalSERVER ||
428                 activedRequestsSERVER := activedRequestsSERVER \/ {nor_abn2} ||
429                 callListSERVER(abnormalSERVER) := callListSERVER(abnormalSERVER
                        ) <- normalSERVER ||
430                 sequenceHistorySERVER := sequenceHistorySERVER <- norAbn2
431         END;
432
433 resp2 <-- NorAbn_resp_Server =
434         PRE
435                 ((lastABNreturnSERVER = noExceptionSERVER) or (
                        lastABNreturnSERVER : maskableExceptionsSERVER)) &
436                 unrecoverableSERVER = FALSE &
437                 size(callListSERVER(abnormalSERVER))>0 &
438                 last(callListSERVER(abnormalSERVER)) = normalSERVER  &
439                 nor_abn2 : activedRequestsSERVER  &
440                 abnormalSERVER = currentComponentSERVER
441
442         THEN
443                 activedRequestsSERVER := activedRequestsSERVER - {nor_abn2} ||
444                 currentComponentSERVER := normalSERVER ||
445                 callListSERVER(abnormalSERVER) := front(callListSERVER(
                        abnormalSERVER)) ||
446                 resp2 := norReturnSERVER ||
447                 sequenceHistorySERVER := sequenceHistorySERVER <- norAbn2_nor
448         END;
449
450
451 resp2 <-- NorAbn_ABNresp_Server =
452         PRE
453                 size(callListSERVER(abnormalSERVER))>0 &
454                 last(callListSERVER(abnormalSERVER)) = normalSERVER  &
455                 nor_abn2 : activedRequestsSERVER  &
456                 abnormalSERVER = currentComponentSERVER
457
458         THEN
459                 activedRequestsSERVER := activedRequestsSERVER - {nor_abn2} ||
460                 currentComponentSERVER := normalSERVER ||
461                 callListSERVER(abnormalSERVER) := front(callListSERVER(
                        abnormalSERVER)) ||
462                 sequenceHistorySERVER := sequenceHistorySERVER <- norAbn2_abn
                        ||
463                 ANY val WHERE val : AbnormalReturnsSERVER & val :
                        failureExceptionsSERVER & val /= noExceptionSERVER & val /:
```

```
                        interfaceExceptionsSERVER
464                     THEN
465                         IF val /: maskableExceptionsSERVER
466                             THEN
467                                 unrecoverableSERVER := TRUE
468                         END ||
469                         resp2 := val ||
470                         lastExternalABNreturnSERVER := val
471                     END
472         END;
473
474
475
476 /*80*/
477 AbnNor_req_Server =
478         PRE
479                 abnormalSERVER = currentComponentSERVER  &
480                 abn_nor2 /: activedRequestsSERVER
481         THEN
482                 currentComponentSERVER := normalSERVER ||
483                 activedRequestsSERVER := activedRequestsSERVER \/ {abn_nor2} ||
484                 callListSERVER(normalSERVER) := callListSERVER(normalSERVER) <-
                        abnormalSERVER ||
485                 sequenceHistorySERVER := sequenceHistorySERVER <- abnNor2
486         END;
487
488 resp2 <-- AbnNor_resp_Server =
489         PRE
490                 size(callListSERVER(normalSERVER))>0 &
491                 last(callListSERVER(normalSERVER)) = abnormalSERVER  &
492                 abn_nor2 : activedRequestsSERVER  &
493                 normalSERVER = currentComponentSERVER
494
495         THEN
496                 faultMasked2 := TRUE ||
497                 activedRequestsSERVER := activedRequestsSERVER - {abn_nor2} ||
498                 currentComponentSERVER := abnormalSERVER ||
499                 callListSERVER(normalSERVER) := front(callListSERVER(
                    normalSERVER)) ||
500                 resp2 :=  norReturnSERVER ||
501                 sequenceHistorySERVER := sequenceHistorySERVER <- abnNor2_nor
502         END;
503
504
505 resp2 <-- AbnNor_intABNresp_Server =
506         PRE
507                 size(callListSERVER(normalSERVER))>0 &
508                 last(callListSERVER(normalSERVER)) = abnormalSERVER  &
509                 abn_nor2 : activedRequestsSERVER  &
510                 normalSERVER = currentComponentSERVER
511
512         THEN
513                 faultMasked2 := FALSE ||
514                 activedRequestsSERVER := activedRequestsSERVER - {abn_nor2} ||
```

```
515                 currentComponentSERVER := abnormalSERVER ||
516                 callListSERVER(normalSERVER) := front(callListSERVER(
                        normalSERVER)) ||
517                 sequenceHistorySERVER := sequenceHistorySERVER <-
                        abnNor2_intAbn ||
518                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER & val /: interfaceExceptionsSERVER
519                         THEN
520                             IF val /: maskableExceptionsSERVER
521                                 THEN
522                                     unrecoverableSERVER := TRUE
523                             END ||
524                             resp2 := val ||
525                             lastABNreturnSERVER := val
526                         END
527         END;
528
529
530
531
532
533
534 /*90*/
535 AbnReq_req_Server =
536         PRE
537                 abnormalSERVER = currentComponentSERVER  &
538                 abn_req2 /: activedRequestsSERVER
539         THEN
540                 currentComponentSERVER := requiredSERVER ||
541                 activedRequestsSERVER := activedRequestsSERVER \/ {abn_req2} ||
542                 callListSERVER(requiredSERVER) := callListSERVER(requiredSERVER
                        ) <- abnormalSERVER ||
543                 sequenceHistorySERVER := sequenceHistorySERVER <- abnReq2
544         END;
545
546 resp2 <-- AbnReq_resp_Server =
547         PRE
548                 size(callListSERVER(requiredSERVER))>0 &
549                 last(callListSERVER(requiredSERVER)) = abnormalSERVER  &
550                 abn_req2 : activedRequestsSERVER  &
551                 requiredSERVER = currentComponentSERVER
552
553         THEN
554                 activedRequestsSERVER := activedRequestsSERVER - {abn_req2} ||
555                 currentComponentSERVER := abnormalSERVER ||
556                 callListSERVER(requiredSERVER) := front(callListSERVER(
                        requiredSERVER)) ||
557                 resp2 := norReturnSERVER ||
558                 sequenceHistorySERVER := sequenceHistorySERVER <- abnReq2_nor
559         END;
560
561
562 resp2 <-- AbnReq_ABNresp2_Server =
563         PRE
```

```
564                  size(callListSERVER(requiredSERVER))>0 &
565                  last(callListSERVER(requiredSERVER)) = abnormalSERVER  &
566                  abn_req2 : activedRequestsSERVER  &
567                  requiredSERVER = currentComponentSERVER
568
569          THEN
570                  activedRequestsSERVER := activedRequestsSERVER - {abn_req2} ||
571                  currentComponentSERVER := abnormalSERVER ||
572                  callListSERVER(requiredSERVER) := front(callListSERVER(
                         requiredSERVER)) ||
573                  sequenceHistorySERVER := sequenceHistorySERVER <- abnReq2_abn
                         ||
574                  ANY val WHERE val : AbnormalReturnsSERVER & val /=
                         noExceptionSERVER & val /: interfaceExceptionsSERVER
575                      THEN
576                          IF val /: maskableExceptionsSERVER
577                              THEN
578                                  unrecoverableSERVER := TRUE
579                          END ||
580                          resp2 := val ||
581                          lastABNreturnSERVER := val
582                      END
583          END;
584
585
586 /*100*/
587 AbnProv_req_Server =
588          PRE
589                  abnormalSERVER = currentComponentSERVER  &
590                  abn_prov2 /: activedRequestsSERVER
591          THEN
592                  currentComponentSERVER := providedSERVER ||
593                  activedRequestsSERVER := activedRequestsSERVER \/ {abn_prov2}
                         ||
594                  callListSERVER(providedSERVER) := callListSERVER(providedSERVER
                         ) <- abnormalSERVER ||
595                  sequenceHistorySERVER := sequenceHistorySERVER <- abnProv2
596          END;
597
598 resp2 <-- AbnProv_resp_Server =
599          PRE
600                  size(callListSERVER(providedSERVER))>0 &
601                  last(callListSERVER(providedSERVER)) = abnormalSERVER  &
602                  abn_prov2 : activedRequestsSERVER  &
603                  providedSERVER = currentComponentSERVER
604
605          THEN
606                  activedRequestsSERVER := activedRequestsSERVER - {abn_prov2} ||
607                  currentComponentSERVER := abnormalSERVER ||
608                  callListSERVER(providedSERVER) := front(callListSERVER(
                         providedSERVER)) ||
609                  resp2 := norReturnSERVER ||
610                  sequenceHistorySERVER := sequenceHistorySERVER <- abnProv2_nor
611          END;
```

```
612
613
614 resp2 <-- AbnProv_ABNresp_Server =
615         PRE
616                 size(callListSERVER(providedSERVER))>0 &
617                 last(callListSERVER(providedSERVER)) = abnormalSERVER  &
618                 abn_prov2 : activedRequestsSERVER  &
619                 providedSERVER = currentComponentSERVER
620
621         THEN
622                 activedRequestsSERVER := activedRequestsSERVER - {abn_prov2} ||
623                 currentComponentSERVER := abnormalSERVER ||
624                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
625                 sequenceHistorySERVER := sequenceHistorySERVER <- abnProv2_abn
                        ||
626                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER
627                     THEN
628                         IF val /: maskableExceptionsSERVER
629                             THEN
630                                 unrecoverableSERVER := TRUE
631                         END ||
632                         resp2 := val ||
633                         lastABNreturnSERVER := val
634                     END
635         END;
636
637
638
639
640 /*110*/
641 ReqAbn_req_Server =
642         PRE
643                 requiredSERVER = currentComponentSERVER  &
644                 req_abn2 /: activedRequestsSERVER
645         THEN
646                 currentComponentSERVER := abnormalSERVER ||
647                 activedRequestsSERVER := activedRequestsSERVER \/ {req_abn2} ||
648                 callListSERVER(abnormalSERVER) := callListSERVER(abnormalSERVER
                        ) <- requiredSERVER ||
649                 sequenceHistorySERVER := sequenceHistorySERVER <- reqAbn2
650         END;
651
652 resp2 <-- ReqAbn_resp_Server =
653         PRE
654                 ((lastABNreturnSERVER = noExceptionSERVER) or (
                        lastABNreturnSERVER : maskableExceptionsSERVER)) &
655                 unrecoverableSERVER = FALSE &
656                 size(callListSERVER(abnormalSERVER))>0 &
657                 last(callListSERVER(abnormalSERVER)) = requiredSERVER  &
658                 req_abn2 : activedRequestsSERVER  &
659                 abnormalSERVER = currentComponentSERVER
660
```

```
661         THEN
662                 lastABNreturnSERVER := noExceptionSERVER ||
663                 activedRequestsSERVER := activedRequestsSERVER - {req_abn2} ||
664                 currentComponentSERVER := requiredSERVER ||
665                 callListSERVER(abnormalSERVER) := front(callListSERVER(
                        abnormalSERVER)) ||
666                 resp2 := norReturnSERVER ||
667                 sequenceHistorySERVER := sequenceHistorySERVER <- reqAbn2_nor
668         END;
669
670
671 resp2 <-- ReqAbn_ABNresp_Server =
672         PRE
673                 size(callListSERVER(abnormalSERVER))>0 &
674                 last(callListSERVER(abnormalSERVER)) = requiredSERVER &
675                 req_abn2 : activedRequestsSERVER &
676                 abnormalSERVER = currentComponentSERVER
677
678         THEN
679                 activedRequestsSERVER := activedRequestsSERVER - {req_abn2} ||
680                 currentComponentSERVER := requiredSERVER ||
681                 callListSERVER(abnormalSERVER) := front(callListSERVER(
                        abnormalSERVER)) ||
682                 sequenceHistorySERVER := sequenceHistorySERVER <- reqAbn2_abn
                        ||
683                 ANY val WHERE val : AbnormalReturnsSERVER & val :
                        failureExceptionsSERVER & val /= noExceptionSERVER & val /:
                         interfaceExceptionsSERVER
684                       THEN
685                           IF val /: maskableExceptionsSERVER
686                               THEN
687                                   unrecoverableSERVER := TRUE
688                           END ||
689                           resp2 := val ||
690                           lastExternalABNreturnSERVER := val
691                       END
692         END;
693
694
695 /*120*/
696 ReqNor_req_Server =
697         PRE
698                 requiredSERVER = currentComponentSERVER &
699                 req_nor2 /: activedRequestsSERVER
700         THEN
701                 currentComponentSERVER := normalSERVER ||
702                 activedRequestsSERVER := activedRequestsSERVER \/ {req_nor2} ||
703                 callListSERVER(normalSERVER) := callListSERVER(normalSERVER) <-
                         requiredSERVER ||
704                 sequenceHistorySERVER := sequenceHistorySERVER <- reqNor2
705         END;
706
707 resp2 <-- ReqNor_resp_Server =
708         PRE
```

```
709                 size(callListSERVER(normalSERVER))>0 &
710                 last(callListSERVER(normalSERVER)) = requiredSERVER  &
711                 req_nor2 : activedRequestsSERVER  &
712                 normalSERVER = currentComponentSERVER
713
714         THEN
715                 activedRequestsSERVER := activedRequestsSERVER - {req_nor2} ||
716                 currentComponentSERVER := requiredSERVER ||
717                 callListSERVER(normalSERVER) := front(callListSERVER(
                        normalSERVER)) ||
718                 resp2 := norReturnSERVER ||
719                 sequenceHistorySERVER := sequenceHistorySERVER <- reqNor2_nor
720         END;
721
722
723 resp2 <-- ReqNor_intABNresp_Server =
724         PRE
725                 size(callListSERVER(normalSERVER))>0 &
726                 last(callListSERVER(normalSERVER)) = requiredSERVER  &
727                 req_nor2 : activedRequestsSERVER  &
728                 normalSERVER = currentComponentSERVER
729
730         THEN
731                 activedRequestsSERVER := activedRequestsSERVER - {req_nor2} ||
732                 currentComponentSERVER := requiredSERVER ||
733                 callListSERVER(normalSERVER) := front(callListSERVER(
                        normalSERVER)) ||
734                 sequenceHistorySERVER := sequenceHistorySERVER <-
                        reqNor2_intAbn ||
735                 ANY val WHERE val : AbnormalReturnsSERVER & val /=
                        noExceptionSERVER & val /: interfaceExceptionsSERVER
736                         THEN
737                             IF val /: maskableExceptionsSERVER
738                                 THEN
739                                     unrecoverableSERVER := TRUE
740                             END ||
741                             resp2 := val ||
742                             lastABNreturnSERVER := val
743                         END
744         END;
745
746
747
748
749
750 /*130*/
751 ReqProv_req_Server =
752         PRE
753                 requiredSERVER = currentComponentSERVER  &
754                 req_prov2 /: activedRequestsSERVER
755         THEN
756                 currentComponentSERVER := providedSERVER ||
757                 activedRequestsSERVER := activedRequestsSERVER \/ {req_prov2}
                        ||
```

```
758                 callListSERVER(providedSERVER) := callListSERVER(providedSERVER
                        ) <- requiredSERVER ||
759                 sequenceHistorySERVER := sequenceHistorySERVER <- reqProv2
760         END;
761
762 resp2 <-- ReqProv_resp_Server =
763         PRE
764                 size(callListSERVER(providedSERVER))>0 &
765                 last(callListSERVER(providedSERVER)) = requiredSERVER  &
766                 req_prov2 : activedRequestsSERVER  &
767                 providedSERVER = currentComponentSERVER
768
769         THEN
770                 activedRequestsSERVER := activedRequestsSERVER - {req_prov2} ||
771                 currentComponentSERVER := requiredSERVER ||
772                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
773                 resp2 := norReturnSERVER ||
774                 sequenceHistorySERVER := sequenceHistorySERVER <- reqProv2_nor
775         END;
776
777
778 resp2 <-- ReqProv_ABNresp_Server =
779         PRE
780                 size(callListSERVER(providedSERVER))>0 &
781                 last(callListSERVER(providedSERVER)) = requiredSERVER  &
782                 req_prov2 : activedRequestsSERVER  &
783                 providedSERVER = currentComponentSERVER
784
785         THEN
786                 activedRequestsSERVER := activedRequestsSERVER - {req_prov2} ||
787                 currentComponentSERVER := requiredSERVER ||
788                 callListSERVER(providedSERVER) := front(callListSERVER(
                        providedSERVER)) ||
789                 sequenceHistorySERVER := sequenceHistorySERVER <- reqProv2_abn
                        ||
790                 ANY val WHERE val : AbnormalReturnsSERVER  & val /=
                        noExceptionSERVER
791                     THEN
792                         IF val /: maskableExceptionsSERVER
793                             THEN
794                                 unrecoverableSERVER := TRUE
795                         END ||
796                         resp2 := val ||
797                         lastABNreturnSERVER := val
798                     END
799         END;
800
801
802
803
804 /*140*/
805 ReqServer_req_Server =
806         PRE
```

```
807                  requiredSERVER = currentComponentSERVER  &
808                  req_srv2 /: activedRequestsSERVER
809         THEN
810                  currentComponentSERVER := serverSERVER ||
811                  activedRequestsSERVER := activedRequestsSERVER \/ {req_srv2} ||
812                  callListSERVER(serverSERVER) := callListSERVER(serverSERVER) <-
                          requiredSERVER ||
813                  sequenceHistorySERVER := sequenceHistorySERVER <- reqSrv2
814         END;
815
816 resp2 <-- ReqServer_resp_Server =
817         PRE
818                  size(callListSERVER(serverSERVER))>0 &
819                  last(callListSERVER(serverSERVER)) = requiredSERVER  &
820                  req_srv2 : activedRequestsSERVER  &
821                  serverSERVER = currentComponentSERVER
822
823         THEN
824                  activedRequestsSERVER := activedRequestsSERVER - {req_srv2} ||
825                  currentComponentSERVER := requiredSERVER ||
826                  callListSERVER(serverSERVER) := front(callListSERVER(
                         serverSERVER)) ||
827                  resp2 := norReturnSERVER ||
828                  sequenceHistorySERVER := sequenceHistorySERVER <- reqSrv2_nor
829         END;
830
831
832 resp2 <-- ReqServer_ABNresp_Server =
833         PRE
834                  size(callListSERVER(serverSERVER))>0 &
835                  last(callListSERVER(serverSERVER)) = requiredSERVER  &
836                  req_srv2 : activedRequestsSERVER  &
837                  serverSERVER = currentComponentSERVER
838
839         THEN
840                  activedRequestsSERVER := activedRequestsSERVER - {req_srv2} ||
841                  currentComponentSERVER := requiredSERVER ||
842                  callListSERVER(serverSERVER) := front(callListSERVER(
                         serverSERVER)) ||
843                  sequenceHistorySERVER := sequenceHistorySERVER <- reqSrv2_abn
                         ||
844                  ANY val WHERE val : AbnormalReturnsSERVER & val /=
                         noExceptionSERVER & val /: interfaceExceptionsSERVER
845                        THEN
846                            IF val /: maskableExceptionsSERVER
847                                THEN
848                                    unrecoverableSERVER := TRUE
849                            END ||
850                            resp2 := val ||
851                            lastABNreturnSERVER := val
852                        END
853         END
854
855 END
```

D.4 CSP Process Algebra

```
1  MAIN = Start!clientComponent -> REQUEST1;;
2  REQUEST1 = ((Service_req!clientComponent!to!interface -> CONN) [] Stop -> MAIN
       );;
3
4
5  -- a -> a_b
6  RETURN1_NOR_CONN = (Service_resp!from!to!interface?N -> REQUEST1);;
7  RETURN1_ABN_CONN = (Service_ABNresp!from!to!interface?E -> REQUEST1);;
8
9
10 -- ******************************************************
11 -- ******************************************************
12 -- CONN ARCHITECTURAL ELEMENT
13 -- ******************************************************
14 --SPECIFICATION:
15 CONN = Start_Conn -> (CLIENT [] Stop_Conn -> CONN);;
16 CLIENT = ClientProv_req_Conn -> PROVIDED;;
17
18
19 -- *********************
20 -- ***NORMAL EXECUTION***
21 -- *********************
22
23 -- Client -> Provided
24 PROVIDED = ((ProvNor_req_Conn -> NORMAL) [] INTERFACE_EXCEPTION);;
25 INTERFACE_EXCEPTION = ClientProv_ABNresp_Conn?E -> Stop_Conn ->
       RETURN1_ABN_CONN;;
26
27 -- Provided -> Normal
28 NORMAL = (NORMAL_NORMAL_REPONSE [] NORMAL_INTERNAL_EXCEPTION [] (
       NorReq_req_Conn -> REQUIRED));;
29 NORMAL_NORMAL_REPONSE = ProvNor_resp_Conn?R1 -> ClientProv_resp_Conn.R1->
       Stop_Conn -> RETURN1_NOR_CONN;;
30 NORMAL_INTERNAL_EXCEPTION = ProvNor_ABNresp_Conn?IE -> HANDLING(IE);; --
       internal exception
31
32
33 -- PROPAGATION
34 -- Provided -> Normal -> Required
35 REQUIRED = ReqServer_req_Conn -> COMP2_PROV_NOR_REQ;; --external service
36 EXTERNAL_NOR = ReqServer_resp_Conn?R2 -> (REQUIRED [] (NorReq_resp_Conn.R3 ->
       NORMAL));;
37 EXTERNAL_ABN = ReqServer_resp_Conn?EE -> NorReq_ABNresp_Conn.EE ->
       ProvNor_ABNresp_Conn.EE -> HANDLING(EE);;
38
39
40
41
42
```

```
43 -- **************
44 -- ***HANDLING***
45 -- **************
46
47 -- Provided -> Abnormal
48 HANDLING(E) = ProvAbn_req_Conn -> (SUCCESS_OR_UNSUCCESS1);;
49 SUCCESS1 = (ABN_NOR [] ABN_REQ);; --handling
50 SUCCESS2 = (ABN_NOR [] ABN_REQ [] (ProvAbn_resp_Conn.R2-> ClientProv_resp_Conn
     .R3 -> Stop_Conn -> RETURN1_NOR_CONN));; --handling
51 ABN_NOR = AbnNor_req_Conn -> (ABN_NOR_NORMAL [] ABN_NOR_INTERNAL_ESCEPTION []
     ABN_NOR_REQUIRED);; --roll back / retry
52 ABN_REQ = AbnReq_req_Conn -> ABN_REQ_REQUIRED;;--external services
53 UNSUCCESS = ProvAbn_ABNresp_Conn?E1 -> ClientProv_ABNresp_Conn.E1 -> Stop_Conn
     -> RETURN1_ABN_CONN;;
54 SUCCESS_OR_UNSUCCESS1 = SUCCESS1 [] UNSUCCESS;;
55 SUCCESS_OR_UNSUCCESS2 = SUCCESS2 [] UNSUCCESS;;
56
57 -- Abnormal -> Normal
58 ABN_NOR_NORMAL = (ABN_NOR_NORMAL_NORMAL_REPONSE []
     ABN_NOR_NORMAL_INTERNAL_EXCEPTION [] (NorReq_req_Conn -> ABN_NOR_REQUIRED)
     );;
59 ABN_NOR_NORMAL_NORMAL_REPONSE = AbnNor_resp_Conn?R1 -> SUCCESS_OR_UNSUCCESS2;;
     -- can be an abort
60 ABN_NOR_NORMAL_INTERNAL_EXCEPTION = AbnNor_ABNresp_Conn?IE ->
     SUCCESS_OR_UNSUCCESS1;; --internal exception - it is possible to try again
61
62 --PROPAGATION
63 -- Abnormal -> Normal -> Req
64 ABN_NOR_REQUIRED = ReqServer_req_Conn -> COMP2_ABN_NOR_REQ;; --external
     service
65 ABN_NOR_EXTERNAL_NOR = ReqServer_resp_Conn?R2 -> (ABN_NOR_REQUIRED [] (
     NorReq_resp_Conn.R3 -> ABN_NOR_NORMAL));;
66 ABN_NOR_EXTERNAL_ABN = ReqServer_resp_Conn?EE -> NorReq_ABNresp_Conn.EE ->
     AbnNor_ABNresp_Conn.EE -> SUCCESS_OR_UNSUCCESS1;;
67
68 --PROPAGATION
69 -- Abnormal -> Required
70 ABN_REQ_REQUIRED = ReqServer_req_Conn -> COMP2_ABN_REQ;; --external service
71 ABN_REQ_EXTERNAL_NOR = ReqServer_resp_Conn?R2 -> (ABN_REQ_REQUIRED [] (
     AbnReq_resp_Conn.R3 -> SUCCESS_OR_UNSUCCESS2));; -- it is possible to
     execute an external service (e.g. logging) despite the fault has not been
     masked.
72 ABN_REQ_EXTERNAL_ABN = ReqServer_resp_Conn?EE -> AbnReq_ABNresp_Conn.EE ->
     SUCCESS_OR_UNSUCCESS2;; -- it is possible an exception decurrent of an
     external service (e.g. logging), but the fault has been masked before it.
73
74 -- **********************************************************
75 -- **********************************************************
76
77
78
79
80
81
```

```
82
83
84 -- *************************
85 -- SERVER ARCHITECTURAL ELEMENT
86 -- *************************
87
88
89
90 -- ****************************************************
91 -- ****************************************************
92 -- SERVER - COMP2_ABN_NOR_REQ
93 -- ****************************************************
94 --SPECIFICATION:
95 COMP2_ABN_NOR_REQ = Start_Server -> (CLIENT2 [] Stop_Server ->
      COMP2_ABN_NOR_REQ);;
96 CLIENT2 = ClientProv_req_Server -> PROVIDED2;;
97
98
99 -- *********************
100 -- ***NORMAL2 EXECUTION***
101 -- *********************
102
103 -- Client -> Provided
104 PROVIDED2 = ((ProvNor_req_Server -> NORMAL2) [] INTERFACE_EXCEPTION2);;
105 INTERFACE_EXCEPTION2 = ClientProv_ABNresp_Server?E -> Stop_Server ->
      ABN_NOR_EXTERNAL_ABN;;
106
107 -- Provided -> Normal
108 NORMAL2 = (NORMAL2_NORMAL2_REPONSE [] NORMAL2_INTERNAL_EXCEPTION [] (
      NorReq_req_Server -> REQUIRED2));;
109 NORMAL2_NORMAL2_REPONSE = ProvNor_resp_Server?R1 -> ClientProv_resp_Server.R1
      -> Stop_Server -> ABN_NOR_EXTERNAL_NOR;;
110 NORMAL2_INTERNAL_EXCEPTION = ProvNor_intABNresp_Server?IE -> HANDLING2(IE);;
      --internal exception
111
112 -- Provided -> Normal -> Required
113 REQUIRED2 = ReqServer_req_Server -> (EXTERNAL_NOR2 [] EXTERNAL_ABN2);; --
      external service
114 EXTERNAL_NOR2 = ReqServer_resp_Server?R2 -> (REQUIRED2 [] (NorReq_resp_Server.
      R3 -> NORMAL2));;
115 EXTERNAL_ABN2 = ReqServer_ABNresp_Server?EE -> NorReq_ABNresp_Server.EE ->
      ProvNor_extABNresp_Server.EE -> HANDLING2(EE);;
116
117
118
119 -- **************
120 -- ***HANDLING2***
121 -- **************
122
123 -- Provided -> Abnormal
124 HANDLING2(E) = ProvAbn_req_Server -> (SUCCESS_OR_UNSUCCESS21_2);;
125 SUCCESS1_2 = (ABN_NOR2 [] ABN_REQ2);; --handling
126 SUCCESS2_2 = (ABN_NOR2 [] ABN_REQ2 [] (ProvAbn_resp_Server.R2->
      ClientProv_resp_Server.R3 -> Stop_Server -> ABN_NOR_EXTERNAL_NOR));; --
```

```
    handling
127 ABN_NOR2 = AbnNor_req_Server -> (ABN_NOR2_NORMAL2 []
    ABN_NOR2_INTERNAL_ESCEPTION [] ABN_NOR2_REQUIRED2);; --roll back / retry
128 ABN_REQ2 = AbnReq_req_Server -> ABN_REQ2_REQUIRED2;;--external services
129 UNSUCCESS2 = ProvAbn_ABNresp_Server?E1 -> ClientProv_ABNresp_Server.E1 ->
    Stop_Server -> ABN_NOR_EXTERNAL_ABN;;
130 SUCCESS_OR_UNSUCCESS21_2 = SUCCESS1_2 [] UNSUCCESS2;;
131 SUCCESS_OR_UNSUCCESS22 = SUCCESS2_2 [] UNSUCCESS2;;
132
133 -- Abnormal -> Normal
134 ABN_NOR2_NORMAL2 = (ABN_NOR2_NORMAL2_NORMAL2_REPONSE []
    ABN_NOR2_NORMAL2_INTERNAL_EXCEPTION [] (NorReq_req_Server ->
    ABN_NOR2_REQUIRED2));;
135 ABN_NOR2_NORMAL2_NORMAL2_REPONSE = AbnNor_resp_Server?R1 ->
    SUCCESS_OR_UNSUCCESS22;; -- can be an abort
136 ABN_NOR2_NORMAL2_INTERNAL_EXCEPTION = AbnNor_intABNresp_Server?IE ->
    SUCCESS_OR_UNSUCCESS21_2;; --internal exception - it is possible to try
    again
137
138 -- Abnormal -> Normal -> Req
139 ABN_NOR2_REQUIRED2 = ReqServer_req_Server -> (ABN_NOR2_EXTERNAL_NOR2 []
    ABN_NOR2_EXTERNAL_ABN2);; --external service
140 ABN_NOR2_EXTERNAL_NOR2 = ReqServer_resp_Server?R2 -> (ABN_NOR2_REQUIRED2 [] (
    NorReq_resp_Server.R3 -> ABN_NOR2_NORMAL2));;
141 ABN_NOR2_EXTERNAL_ABN2 = ReqServer_ABNresp_Server?EE -> NorReq_ABNresp_Server.
    EE -> AbnNor_extABNresp_Server.EE -> SUCCESS_OR_UNSUCCESS21_2;;
142
143 -- Abnormal -> Required
144 ABN_REQ2_REQUIRED2 = ReqServer_req_Server -> (ABN_REQ2_EXTERNAL_NOR2 []
    ABN_REQ2_EXTERNAL_ABN2);; --external service
145 ABN_REQ2_EXTERNAL_NOR2 = ReqServer_resp_Server?R2 -> (ABN_REQ2_REQUIRED2 [] (
    AbnReq_resp_Server.R3 -> SUCCESS_OR_UNSUCCESS22));; -- it is possible to
    execute an external service (e.g. logging) despite the fault has not been
    masked.
146 ABN_REQ2_EXTERNAL_ABN2 = ReqServer_ABNresp_Server?EE -> AbnReq_ABNresp2_Server
    .EE -> SUCCESS_OR_UNSUCCESS22;; -- it is possible an exception decurrent
    of an external service (e.g. logging), but the fault has been masked
    before it.
147 -- ********************************************************
148 -- ********************************************************
149
150
151 -- ********************************************************
152 -- ********************************************************
153 -- ********************************************************
154 -- SERVER - COMP2_PROV_NOR_REQ
155 -- ********************************************************
156 --SPECIFICATION:
157 COMP2_PROV_NOR_REQ = Start_Server -> (CLIENT2_NOR_REQ [] Stop_Server ->
    COMP2_PROV_NOR_REQ);;
158 CLIENT2_NOR_REQ = ClientProv_req_Server -> PROVIDED2_NOR_REQ;;
159
160
161 -- *********************
```

```
162 -- ***NORMAL2_NOR_REQ EXECUTION***
163 -- *********************
164
165 -- Client -> Provided
166 PROVIDED2_NOR_REQ = ((ProvNor_req_Server -> NORMAL2_NOR_REQ) []
      INTERFACE_EXCEPTION2_NOR_REQ);;
167 INTERFACE_EXCEPTION2_NOR_REQ = ClientProv_ABNresp_Server?E -> Stop_Server ->
      EXTERNAL_ABN;;
168
169 -- Provided -> Normal
170 NORMAL2_NOR_REQ = (NORMAL2_NOR_REQ_NORMAL2_NOR_REQ_REPONSE []
      NORMAL2_NOR_REQ_INTERNAL_EXCEPTION [] (NorReq_req_Server ->
      REQUIRED_NOR_REQ2));;
171 NORMAL2_NOR_REQ_NORMAL2_NOR_REQ_REPONSE = ProvNor_resp_Server?R1 ->
      ClientProv_resp_Server.R1-> Stop_Server -> EXTERNAL_NOR;;
172 NORMAL2_NOR_REQ_INTERNAL_EXCEPTION = ProvNor_intABNresp_Server?IE ->
      HANDLING2_NOR_REQ(IE);; --internal exception
173
174 -- Provided -> Normal -> Required
175 REQUIRED_NOR_REQ2 = ReqServer_req_Server -> (EXTERNAL_NOR2_NOR_REQ []
      EXTERNAL_ABN2_NOR_REQ);; --external service
176 EXTERNAL_NOR2_NOR_REQ = ReqServer_resp_Server?R2 -> (REQUIRED_NOR_REQ2 [] (
      NorReq_resp_Server.R3 -> NORMAL2_NOR_REQ));;
177 EXTERNAL_ABN2_NOR_REQ = ReqServer_ABNresp_Server?EE -> NorReq_ABNresp_Server.
      EE -> ProvNor_extABNresp_Server.EE -> HANDLING2_NOR_REQ(EE);;
178
179
180
181 -- **************
182 -- ***HANDLING2_NOR_REQ***
183 -- **************
184
185 -- Provided -> Abnormal
186 HANDLING2_NOR_REQ(E) = ProvAbn_req_Server -> (
      SUCCESS_OR_UNSUCCESS2_NOR_REQ1_2_NOR_REQ);;
187 SUCCESS1_2_NOR_REQ = (ABN_NOR2_NOR_REQ [] ABN_REQ2_NOR_REQ);; --handling
188 SUCCESS2_2_NOR_REQ = (ABN_NOR2_NOR_REQ [] ABN_REQ2_NOR_REQ [] (
      ProvAbn_resp_Server.R2-> ClientProv_resp_Server.R3 -> Stop_Server ->
      EXTERNAL_NOR));; --handling
189 ABN_NOR2_NOR_REQ = AbnNor_req_Server -> (ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ []
      ABN_NOR2_NOR_REQ_INTERNAL_ESCEPTION [] ABN_NOR2_NOR_REQ_REQUIRED_NOR_REQ2)
      ;; --roll back / retry
190 ABN_REQ2_NOR_REQ = AbnReq_req_Server -> ABN_REQ2_NOR_REQ_REQUIRED_NOR_REQ2;;--
      external services
191 UNSUCCESS2_NOR_REQ = ProvAbn_ABNresp_Server?E1 -> ClientProv_ABNresp_Server.E1
       -> Stop_Server -> EXTERNAL_ABN;;
192 SUCCESS_OR_UNSUCCESS2_NOR_REQ1_2_NOR_REQ = SUCCESS1_2_NOR_REQ []
      UNSUCCESS2_NOR_REQ;;
193 SUCCESS_OR_UNSUCCESS2_NOR_REQ2 = SUCCESS2_2_NOR_REQ [] UNSUCCESS2_NOR_REQ;;
194
195 -- Abnormal -> Normal
196 ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ = (
      ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ_NORMAL2_NOR_REQ_REPONSE []
      ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ_INTERNAL_EXCEPTION [] (NorReq_req_Server
```

```
    -> ABN_NOR2_NOR_REQ_REQUIRED_NOR_REQ2) ) ;;
197 ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ_NORMAL2_NOR_REQ_REPONSE = AbnNor_resp_Server?
    R1 -> SUCCESS_OR_UNSUCCESS2_NOR_REQ2;; -- can be an abort
198 ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ_INTERNAL_EXCEPTION = AbnNor_intABNresp_Server
    ?IE -> SUCCESS_OR_UNSUCCESS2_NOR_REQ1_2_NOR_REQ;; --internal exception -
    it is possible to try again
199
200 -- Abnormal -> Normal -> Req
201 ABN_NOR2_NOR_REQ_REQUIRED_NOR_REQ2 = ReqServer_req_Server -> (
    ABN_NOR2_NOR_REQ_EXTERNAL_NOR2_NOR_REQ []
    ABN_NOR2_NOR_REQ_EXTERNAL_ABN2_NOR_REQ) ;; --external service
202 ABN_NOR2_NOR_REQ_EXTERNAL_NOR2_NOR_REQ = ReqServer_resp_Server?R2 -> (
    ABN_NOR2_NOR_REQ_REQUIRED_NOR_REQ2 [] (NorReq_resp_Server.R3 ->
    ABN_NOR2_NOR_REQ_NORMAL2_NOR_REQ) ) ;;
203 ABN_NOR2_NOR_REQ_EXTERNAL_ABN2_NOR_REQ = ReqServer_ABNresp_Server?EE ->
    NorReq_ABNresp_Server.EE -> AbnNor_extABNresp_Server.EE ->
    SUCCESS_OR_UNSUCCESS2_NOR_REQ1_2_NOR_REQ;;
204
205 -- Abnormal -> Required
206 ABN_REQ2_NOR_REQ_REQUIRED_NOR_REQ2 = ReqServer_req_Server -> (
    ABN_REQ2_NOR_REQ_EXTERNAL_NOR2_NOR_REQ []
    ABN_REQ2_NOR_REQ_EXTERNAL_ABN2_NOR_REQ) ;; --external service
207 ABN_REQ2_NOR_REQ_EXTERNAL_NOR2_NOR_REQ = ReqServer_resp_Server?R2 -> (
    ABN_REQ2_NOR_REQ_REQUIRED_NOR_REQ2 [] (AbnReq_resp_Server.R3 ->
    SUCCESS_OR_UNSUCCESS2_NOR_REQ2) ) ;; -- it is possible to execute an
    external service (e.g. logging) despite the fault has not been masked.
208 ABN_REQ2_NOR_REQ_EXTERNAL_ABN2_NOR_REQ = ReqServer_ABNresp_Server?EE ->
    AbnReq_ABNresp2_Server.EE -> SUCCESS_OR_UNSUCCESS2_NOR_REQ2;; -- it is
    possible an exception decurrent of an external service (e.g. logging), but
     the fault has been masked before it.
209 -- ***********************************************************
210 -- ***********************************************************
211
212
213 -- ********************************************************
214 -- ********************************************************
215 -- SERVER - COMP2_ABN_REQ
216 -- ********************************************************
217 --SPECIFICATION:
218 COMP2_ABN_REQ = Start_Server -> (CLIENT2_ABN_REQ [] Stop_Server ->
    COMP2_ABN_REQ) ;;
219 CLIENT2_ABN_REQ = ClientProv_req_Server -> PROVIDED2_ABN_REQ;;
220
221
222 -- *********************
223 -- ***NORMAL2_ABN_REQ EXECUTION***
224 -- *********************
225
226 -- Client -> Provided
227 PROVIDED2_ABN_REQ = ((ProvNor_req_Server -> NORMAL2_ABN_REQ) []
    INTERFACE_EXCEPTION2_ABN_REQ) ;;
228 INTERFACE_EXCEPTION2_ABN_REQ = ClientProv_ABNresp_Server?E -> Stop_Server ->
    ABN_REQ_EXTERNAL_ABN;;
229
```

```
230 -- Provided -> Normal
231 NORMAL2_ABN_REQ = (NORMAL2_ABN_REQ_NORMAL2_ABN_REQ_REPONSE []
      NORMAL2_ABN_REQ_INTERNAL_EXCEPTION [] (NorReq_req_Server ->
      REQUIRED2_ABN_REQ));;
232 NORMAL2_ABN_REQ_NORMAL2_ABN_REQ_REPONSE = ProvNor_resp_Server?R1 ->
      ClientProv_resp_Server.R1-> Stop_Server -> ABN_REQ_EXTERNAL_NOR;;
233 NORMAL2_ABN_REQ_INTERNAL_EXCEPTION = ProvNor_intABNresp_Server?IE ->
      HANDLING2_ABN_REQ(IE);; --internal exception
234
235 -- Provided -> Normal -> Required
236 REQUIRED2_ABN_REQ = ReqServer_req_Server -> (EXTERNAL_NOR2_ABN_REQ []
      EXTERNAL_ABN2_ABN_REQ);; --external service
237 EXTERNAL_NOR2_ABN_REQ = ReqServer_resp_Server?R2 -> (REQUIRED2_ABN_REQ [] (
      NorReq_resp_Server.R3 -> NORMAL2_ABN_REQ));;
238 EXTERNAL_ABN2_ABN_REQ = ReqServer_ABNresp_Server?EE -> NorReq_ABNresp_Server.
      EE -> ProvNor_extABNresp_Server.EE -> HANDLING2_ABN_REQ(EE);;
239
240
241
242 -- **************
243 -- ***HANDLING2_ABN_REQ***
244 -- **************
245
246 -- Provided -> Abnormal
247 HANDLING2_ABN_REQ(E) = ProvAbn_req_Server -> (
      SUCCESS_OR_UNSUCCESS2_ABN_REQ1_2_ABN_REQ);;
248 SUCCESS1_2_ABN_REQ = (ABN_NOR2_ABN_REQ [] ABN_REQ2_ABN_REQ);; --handling
249 SUCCESS2_2_ABN_REQ = (ABN_NOR2_ABN_REQ [] ABN_REQ2_ABN_REQ [] (
      ProvAbn_resp_Server.R2-> ClientProv_resp_Server.R3 -> Stop_Server ->
      ABN_REQ_EXTERNAL_NOR));; --handling
250 ABN_NOR2_ABN_REQ = AbnNor_req_Server -> (ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ []
      ABN_NOR2_ABN_REQ_INTERNAL_ESCEPTION [] ABN_NOR2_ABN_REQ_REQUIRED2_ABN_REQ)
      ;; --roll back / retry
251 ABN_REQ2_ABN_REQ = AbnReq_req_Server -> ABN_REQ2_ABN_REQ_REQUIRED2_ABN_REQ;;--
      external services
252 UNSUCCESS2_ABN_REQ = ProvAbn_ABNresp_Server?E1 -> ClientProv_ABNresp_Server.E1
       -> Stop_Server -> ABN_REQ_EXTERNAL_ABN;;
253 SUCCESS_OR_UNSUCCESS2_ABN_REQ1_2_ABN_REQ = SUCCESS1_2_ABN_REQ []
      UNSUCCESS2_ABN_REQ;;
254 SUCCESS_OR_UNSUCCESS2_ABN_REQ2 = SUCCESS2_2_ABN_REQ [] UNSUCCESS2_ABN_REQ;;
255
256 -- Abnormal -> Normal
257 ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ = (
      ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ_NORMAL2_ABN_REQ_REPONSE []
      ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ_INTERNAL_EXCEPTION [] (NorReq_req_Server
      -> ABN_NOR2_ABN_REQ_REQUIRED2_ABN_REQ));;
258 ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ_NORMAL2_ABN_REQ_REPONSE = AbnNor_resp_Server?
      R1 -> SUCCESS_OR_UNSUCCESS2_ABN_REQ2;; -- can be an abort
259 ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ_INTERNAL_EXCEPTION = AbnNor_intABNresp_Server
      ?IE -> SUCCESS_OR_UNSUCCESS2_ABN_REQ1_2_ABN_REQ;; --internal exception -
      it is possible to try again
260
261 -- Abnormal -> Normal -> Req
```

```
262 ABN_NOR2_ABN_REQ_REQUIRED2_ABN_REQ = ReqServer_req_Server -> (
      ABN_NOR2_ABN_REQ_EXTERNAL_NOR2_ABN_REQ []
      ABN_NOR2_ABN_REQ_EXTERNAL_ABN2_ABN_REQ);; --external service
263 ABN_NOR2_ABN_REQ_EXTERNAL_NOR2_ABN_REQ = ReqServer_resp_Server?R2 -> (
      ABN_NOR2_ABN_REQ_REQUIRED2_ABN_REQ [] (NorReq_resp_Server.R3 ->
      ABN_NOR2_ABN_REQ_NORMAL2_ABN_REQ));;
264 ABN_NOR2_ABN_REQ_EXTERNAL_ABN2_ABN_REQ = ReqServer_ABNresp_Server?EE ->
      NorReq_ABNresp_Server.EE -> AbnNor_extABNresp_Server.EE ->
      SUCCESS_OR_UNSUCCESS2_ABN_REQ1_2_ABN_REQ;;
265
266 -- Abnormal -> Required
267 ABN_REQ2_ABN_REQ_REQUIRED2_ABN_REQ = ReqServer_req_Server -> (
      ABN_REQ2_ABN_REQ_EXTERNAL_NOR2_ABN_REQ []
      ABN_REQ2_ABN_REQ_EXTERNAL_ABN2_ABN_REQ);; --external service
268 ABN_REQ2_ABN_REQ_EXTERNAL_NOR2_ABN_REQ = ReqServer_resp_Server?R2 -> (
      ABN_REQ2_ABN_REQ_REQUIRED2_ABN_REQ [] (AbnReq_resp_Server.R3 ->
      SUCCESS_OR_UNSUCCESS2_ABN_REQ2));; -- it is possible to execute an
      external service (e.g. logging) despite the fault has not been masked.
269 ABN_REQ2_ABN_REQ_EXTERNAL_ABN2_ABN_REQ = ReqServer_ABNresp_Server?EE ->
      AbnReq_ABNresp2_Server.EE -> SUCCESS_OR_UNSUCCESS2_ABN_REQ2;; -- it is
      possible an exception decurrent of an external service (e.g. logging), but
       the fault has been masked before it.
270 -- ***********************************************************************
271 -- ***********************************************************************
```