Architecture-Based Exception Handling

Valérie Issarny, Jean-Pierre Banatre
INRIA-Rocquencourt
Domaine de Voluceau, BP 105, 78 153 Le Chesnay Cédex, France

Abstract

Architecture-based development environments are becoming an effective solution towards the construction of robust distributed systems. Through the abstract description of complex software systems configurations in terms of the interconnection of software elements at the interface level, software reuse and evolution get promoted. In addition, as shown by research results from the software architecture domain, it becomes feasible to provide formal notations for the precise description of configuration behavior, together with associated CASE tools for their automated analyses. However, little attention has been paid to software fault tolerance and in particular exception handling in that context, although this is crucial for achieving software robustness.

This paper investigates the design and implementation of exception handling support for architecture-based development environments. After a survey of the issues raised by exception handling at the level of software architecture description, we introduce an exception handling facility for architecture-based software systems, addressing the resulting extension to architecture description languages and the mapping to implementation of software architectures embedding exception handling.

1. Introduction

The development of large, complex software systems at the architectural level is becoming an effective solution towards ensuring system robustness. Software architectures describe systems at a high-level of abstraction using the three following building blocks [17]: (i) components that represent computation units, (ii) connectors that represent communication protocols, and (iii) configurations that characterize the systems' topology in terms of interconnection of components via connectors. Results in the software architecture field then embrace a number of Architecture Description Languages (ADL), which define notations for the above building blocks so as to enable effectively supporting the software development process. Some existing ADLs introduce notations that are based on formal methods, which allows carrying out useful analyses with the help of CASE tools (e.g. see [18, 2]). Complementary to this work are ADLs that further come along with tools supporting the mapping of architectures to their implementations (e.g. see [22, 15, 16, 23]).

Work in the software architecture domain primarily focuses on the standard (as opposed to exceptional) behavior of the software system. However, it is crucial from the perspective of software robustness to also account for failure occurrences. Failures may be handled through the integration within the system architecture of components and connectors that provide fault tolerance capabilities [21]. Practically, this means that failures are handled by an underlying fault-tolerance mechanism (e.g. transparent replication management). Such fault tolerance means must further be coupled with software fault tolerance support. Software fault tolerance relies at least on an exception handling mechanism, which enables the software developer to specify the actions to be undertaken under the occurrence of application-specific and underlying runtime exceptions.

The focus of this paper is on the introduction of exception handling at the architectural-level, which conveniently complements exception handling implemented within components and connectors. Section 2 discusses issues related to meeting this objective, motivating its need in the light of existing work, and presenting desirable features for the exception handling facility. Section 3 then introduces a base exception handling facility at the architectural level, giving the necessary extension to the ADL. Section 4 addresses mapping of the architectures to their implementations, including needed support from the underlying runtime system for exception handling. Finally, Section 5 assesses our solution with respect to related work, and Section 6 concludes with a summary of our contribution and our research perspectives.

2. Issues in Specifying Exception Handling at the Architectural-level

Simply stated, exception handling enables specifying actions to be undertaken in the presence of exceptional events
during the system execution. In general, exceptions are closely related to the occurrence of failures that prevent a system function to terminate in a state that conforms to the function’s standard specification. An exception handling mechanism then serves implementing the system’s exceptional specification by enabling the definition of: (i) the exceptions to be considered for the given system, and (ii) exception handlers, which prescribe the actions to be executed under the occurrences of relevant exceptions. An exception handling mechanism relies on a model, which specifies: the protocol used for identifying the exception handler to be executed under the occurrence of a given exception, and the action to be executed next to the handler. It is now common practice to implement software components using an exception handling mechanism offered by either the programming language in which case it takes the form of a set of control structures (e.g. Java exception handling [9]), or the underlying operating system in which case it takes the form of a set of system calls (e.g. Windows NT exception handling [8]). However, when developing a software system with an explicit architectural focus, exception handling remains addressed in a way that is internal to the architectural elements. There is no dedicated support to undertake actions at the level of the architecture in the presence of exceptions. If an exception occurrence requires changing the architecture, such a change must in general be handled within the components. This significantly affects the advantages brought by the architectural design since the architecture of the running software system does no longer correspond to the one set at design time.

2.1. An Example

In order to illustrate the issues raised by exception handling at the architectural level, we consider a Distributed File System (DFS), which has the advantage of being understood by the vast majority and representative of a number of distributed software systems. The DFS system is composed of clients and servers that are distributed over the network, clients interacting with servers to access files. At a high-level of abstraction, the DFS software architecture may be described in terms of a client and a server component interacting through a connector providing an RPC-based communication protocol (see Figure 1.a). The corresponding running system is then obtained by having as many component and connector instances as there are interacting clients and servers. The issue of identifying actual instances and bindings among them is here abstracted within the connector. A more concrete version of the architecture is depicted in Figure 1.b. It refines the RPC-based connector by introducing the Locator component, which registers clients and server component instances, and locates the right server instance for clients upon each first access to a file (e.g. requests for creating or opening a file) [11]. A running DFS system then maps onto this architecture where there is an open number of client and server component instances, and of RPC connector instances.

Various exceptions may be considered in the above context. To give a few, consider: a client failing to interact with a given file server due to either actual failure at the level of the server machine or network, or simply a timeout whose value has been set too low; a client requesting a file access that cannot be serviced due to reasons such as unknown file or unauthorized access; a server failing to service a request due to the current server load. All the exceptions raised by either a component or a connector instance should be propagated to the initiator of the request that led to the exception occurrence, for internal exception handling. However, while some exceptions are specific to the given request (e.g. wrong file access), others are relevant to the overall DFS system and thus should normally be additionally handled at the configuration level. We qualify these exceptions as configuration exceptions.

As a typical example of configuration exception, consider the failure of a server instance. Such an exception is expected to be handled within component instances interacting with the failed instance, according to their excep-

![Figure 1. The DFS Example](image-url)
tion. Increased reliability of the DFS system may be enforced by replicating files within distinct server instances. Such a behavior may be implemented through replication of server instances or by letting client instances replicate their requests according to their availability need. Any type of solutions requires evolution of the system’s running configuration through the integration of fault-tolerance components (see Figure 1.c for replication of the server instance). In the same way, improved availability of a file server offering poor response time may be obtained by integrating component instances realizing predictive prefetching [14] (see Figure 1.d). The above configurations may be avoided by integrating the support for reliability and availability within the architecture at design time, i.e., by preventing the occurrence of exceptions. However, the added value of such a support is specific to the client and server instances, which cannot be known in advance. In addition, the occurrence of exceptions can not be prevented in general.

We have illustrated the handling of configuration exceptions in terms of structural changes to the embedding configuration. This is not only because this fits with the given exceptional situations. This is also due to the fact that this is the only handling that we consider as pertinent for configuration exceptions. Specifically, our primary design objective for the exception handling facility is to keep highly abstract the description of software architectures. Hence, only the specification of exceptions and of changes to the configuration must be expressed using the ADL, lower-level details must be abstracted away within components and connectors. Notice that configuration exception handling complements but does not substitute to exception handling within component and connector instances; raised exceptions flow among instances according to the embedding architectural style, and are handled within instances according to the instances’ exceptional specifications.

2.2. Background

This subsection investigates existing base solutions to exception handling within ADL-based environments. Consider first exception handling within component instances. Raised exceptions flow among instances according to the system’s architectural style, i.e., the exception raised by an instance will propagate to another instance according to the connector that is used. The only prerequisite for the ADL is then to enable specifying the exceptions that may be raised and handled by components and connectors so as to at least allow for checking that exceptions will actually be handled, which is required for checking system robustness. The specification of exceptions may then take various forms depending on the robustness checks to be enforced. A minimal solution is to have syntactic checks based on the list of exceptions that are handled and raised by the architectural elements. For a thorough robustness assessment, it is advisory to further have post-conditions associated with exceptions and possibly an abstract specification of the exception handling models that are supported by the architectural elements. Although such a feature is not put forward by existing ADLs, base solutions may be found in the literature, e.g., see the Inscape environment that is a precursor of ADL-based development environments [20]. Nonetheless, rigorous specification of exception handling models and of exception propagation at the architectural level remains an issue for future work. In this paper, we concentrate more specifically on the handling of configuration exceptions.

Configuration exception handling requires defining configuration exceptions together with associated changes to the system’s configuration. This concern relates to the work done in the area of dynamic reconfiguration in the software architecture field. Dynamic reconfiguration of a given software architecture may be either determined at runtime or fixed at design time. In the former case, required changes to the configuration are requested to a reconfiguration manager, which may further enforce constraints on valid changes with respect to invariants set for the system’s software architecture (e.g. see [12, 19, 6]). In the latter case, possible configuration changes are specified and thus anticipated within the architecture description (e.g. see [3]). The solution that is the closest to our concern is the one provided by the Durra environment supporting the development of applications in terms of configurations of tasks. Durra enables specifying changes to the current running configuration with respect to boolean conditions [3]. Our work extends this proposal to the general case of software architecture description while restricting it to the specific case of exception handling.

3. Architectural Exception Handling

As suggested in the previous section, our exception handling model lies in:

- Exception handling implemented within components and connectors, leading to let exceptions flow among them according to the embedding architectural style.
- Exception handling at the level of the architecture so as to enable changing the system’s running configuration according to the occurrence of configuration exceptions, which aims at preventing further occurrences of exceptions within component and connector instances.

Taking the DFS example, we get the following general pattern for the description of software architectures embedding exception handling.
COMPONENT Client:
    /* Operations required by client components */
    /REQUIRES open(...) RAISES Open, Failure;
    /REQUIRES write(...) RAISES invalidPtr, nonWritable, Failure;
    /REQUIRES read(...) RAISES invalidPtr, Failure;
    ....
COMPONENT Server:
    /* Operations provided by server components */
    /PROVIDES open(...) RAISES Open, Failure;
    /PROVIDES write(...) RAISES invalidPtr, nonWritable, Failure;
    /PROVIDES read(...) RAISES invalidPtr, Failure;
    ...
COMPONENT Locator:
    /* Interface for locating a file server */
    /REQUIRES Sopen(...) RAISES Open, Failure;
    /PROVIDES open(...) RAISES Open, Failure;
    ...
CONNECTOR Rpc:
    /* Ports provided for RPC interactions */
    PORT Clt RAISES FailureCom; PORT Srv;
CONFIGURATION Dfs:
    COMPONENTS:
    /* The system is composed of a single */
    /PORT 
    /components:
    /instClt[]: Client; instSrv[]: Server;
CONNECTORS:
    /* The system embeds an open number of Rpc */
    /instances:
    /instRpc(): UNSHARED Rpc;
BINDINGS:
    /* Bindings of required operations to the */
    /matching:
    /instances:
    /instLocator().open AS Rpc.Srv USING instRpc();
    /instLocator().Sopen AS Rpc.Clt TO
    /instClt[]().open AS Rpc.Clt TO
    /instSrv[]().open AS Rpc.Srv USING instRpc();
    /instClt[]().write AS Rpc.Clt TO
    /instSrv[]().write AS Rpc.Srv USING instRpc();
    /instClt[]().read AS Rpc.Clt TO
    /instSrv[]().read AS Rpc.Srv USING instRpc();
    ...
EXCEPTION HANDLING:
    /* Exception handling specification */
    /ARCHITECTURAL EXCEPTIONS: Definition
    /HANDLERS: Definition

Except for the EXCEPTION HANDLING part that is detailed in the following, the above declarations are already supported (not considering the specific syntax) by existing ADLs targeting mapping of architectures to their implementations. From an exception handling point of view, the operations and ports, which are respectively declared within the components and connectors, state the list of expected exceptions. The Failure exception is the exception that is ultimately raised and handled in the presence of unexpected exceptions. The important point is that the ADL compiler must check whether exceptions raised by instances are handled within the configuration. Such a check is dependent on the connector type. For instance, in our example, an RPC connector forwards the exception raised by a server operation to the client if we assume synchronous invocations. However, the exception flow differs in the case of asynchronous invocations. From this perspective, the description sample that is provided is too simple because it does not give behavioral information about exception handling. In the same way, robustness of the configuration with respect to exception handling should account for whether an exception is raised according to the termination model (i.e. the action that is executed after the handler termination is the block that follows the handler declaration) or the resumption model (i.e. the action that is executed after the handler termination is the one that follows the point where the exception was raised). We are currently working on extending architectural description for enforcing the above checks. It is our belief that this may be conveniently addressed using the Wright solution to specifying the behavior of connectors [2]. Other relevant approaches include work on the behavioral specification of CORBA objects coping with interaction protocols and exception handling [4, 7]

Consider now the issue of exception handling at the architectural level, this requires precisely setting the corresponding exception handling model, and providing means to specify configuration exceptions and related handlers. We introduce a simple exception handling model. An architecture is associated with a number of configuration exceptions, and each such exception is syntactically bound to a handler that sets changes to be made to the configuration upon the exception’s occurrence. Regarding the progress of the system’s execution under exception handling, some component instances must be blocked during the execution of handlers so as to guarantee that the associated reconfiguration processes leave the system in a consistent state (e.g. see [12]). Once the execution of the exception handler terminates, blocked instances resume their execution where they got blocked. Notice that configuration exceptions are asynchronous and may occur concurrently. We enforce serialization of the handler executions for consistency. Finally, specification of configuration exception handling is enclosed within the EXCEPTION HANDLING clause, which defines the configuration exceptions and associated handlers.

3.1. Specifying Configuration Exceptions

Consider first the specification of configuration exceptions. It must abstractly describe the conditions upon the system state that lead to exceptions occurrences. With an architectural focus, the system state is defined with respect to the embedding configuration, i.e., the interfaces of the architectural elements and the interactions among these elements through their interfaces. An exceptional system state then relates to the behavior of interactions among architectural elements. The definition of a configuration exception thus decomposes into:

- The exception’s name and parameters.
• The **SUBCONFIGURATION** clause that gives the set of component and connector instances of the embedding configuration whose interactions may lead to the exception occurrence.

• The **INTERACTIONS** clause that gives the set of interactions among the component and connector instances of the subconfiguration whose behavior may lead to the exception occurrence.

• The **OCCURS** clause that gives the condition associated with the exception occurrence, with respect to the above set of interactions.

We further detail the specification of configuration exceptions. As exemplified by the D FS configuration, there may be several subconfigurations having the same structure and that differ only with respect to the embedded instances of components and connectors. For example, this is the case of the subconfiguration consisting of two interacting client and server instances. Such subconfigurations are distinguished by parameterizing the exception with the embedded component and connector instances. The definition of the subconfiguration associated with an exception is further simplified as follows. The component and connector instances stated in the parameter list of the exception are implicitly considered as being embedded within the subconfiguration. Also, when connector instances may be deduced from the component instances embedded in the subconfiguration and the overall configuration description, they are not specified.

Any interaction occurring among architectural elements may be decomposed as a sequence of events occurring at the interconnection points of the elements. This is in particular illustrated by the formal specification of architectural connections proposed in [2]. The events of interest for the detection of configuration exceptions are the following:

• The initialization of an interaction by either a component or a connector instance (e.g. a client instance issuing an RPC request). Each such event is denoted by at least the **OUT** keyword followed by the initiating instance. If the interaction relates to specific operations (or ports) of the instance’s interface, these operations are also specified.

• The handling of an interaction by either a component or a connector instance (e.g. receipt of an RPC request by a server instance). Each such event is denoted by the **IN** keyword followed by the handling instance, and possibly a list of operations (or ports).

• The signal of an exception by either a component or a connector instance. Such an event is specified using the **EXCEPTION** keyword followed by at least the relevant exception, in which case it may be any signaling of the given named exception by the various instances embedded in the subconfiguration. The event may be more specific by relating to the signal of the exception by some instances or even some operations (or ports) of some instances, in which case these are specified.

The interactions whose behavior is relevant for the detection of a given configuration exception are then defined as the sets of the events composing the interactions.

Finally, the condition associated with exception occurrence is stated in terms of a boolean condition over the sets of monitored interaction events. One of our ultimate goal in the specification of configuration exception handling is to enable automating its implementation out of base generic underlying services. Regarding the specification of exception occurrence, this is currently dealt with through the provision of base functions defined over sets of interaction events.

Taking the D FS example for illustration, we give the specification of the three following configuration exceptions:

• The exception **UnreliableServer** occurs when a server instance fails “too often”. This exception relates to the subconfigurations of the D FS system that consist of a single server instance. In order to identify the unreliable server instance, the exception is parameterized by the instance. The exception relates to the behavior of the interactions with the server instance, and occurs according to the ratio between the occurrences of **Failure** as raised by the server instance, and calls to the server instance.

• The exception **UnreliableAccess** occurs when the invocation to a server instance is considered as failing “too frequently” from the standpoint of a given client instance. This exception relates to all the subconfigurations made of instances of a client, a server and the Locator; it is parameterized by the relevant client and server instances. The exception occurs depending on the ratio between the invocations issued by the client instance, and the occurrences of the **Failure** and **FailureCom** exceptions, as respectively raised by the server and RPC instances.

• The exception **LowResponseTime** occurs when the response time for requests issued by a given client instance exceeds some threshold. This exception relates to the subconfigurations that consist of a single client instance, and is parameterized by the specific instance. The exception occurs according to the response time of the interactions of the client instance with server instances.

We get the declaration given hereafter for the **CONFIGURATION EXCEPTIONS** part of the D FS definition. The detection of the occurrences of **UnreliableServer** and **UnreliableAccess** is quite trivial. It consists of detecting that the
number of times the monitored exceptions were raised divided by the total number of monitored invocations, exceeds a given threshold. Assuming that the interactions with the Locator instance is not a performance bottleneck, the occurrence of the LowResponseTime exception is detected by monitoring the response times of all the invocations to the read and write operations, whose average must not exceed a given threshold. The response times are here monitored according to the behavior of the interactions of the client with the server instance when the client issues a request (e.g. OUT C.read) and gets back its result (e.g. IN C.read).

**CONFIGURATION EXCEPTIONS:**

- **EXCEPTION UnreliableServer(S: Server):**
  - INTERACTIONS: failed: EXCEPTION Failure; called: IN S;
  - OCCURS: \( \text{size}(\text{failed}) \geq t \)

- **EXCEPTION UnreliableAccess(S: Server, C: Client):**
  - SUBCONFIGURATION: instantLocator;
  - INTERACTIONS: failedSrv: EXCEPTION S.Failure; failedCom: FailureCom; call: OUT C;
  - OCCURS: \( \text{size}(\text{failedSrv}) + \text{size}(\text{failedCom}) \geq t' \)

- **EXCEPTION LowResponseTime(C: Client):**
  - INTERACTIONS: call: OUT C.read, OUT C.write, IN C.read, IN C.write;
  - OCCURS: \( \text{average} \left( \text{response time} \right) \geq t'' \)

### 3.2. Specifying Handlers

Given the specification of configuration exceptions, the associated handlers specify required changes to the running configuration. The main issue that arises here relates to managing the various reconfigurations occurring over the system’s lifetime. For instance, consider the running DFS configuration after the handling of the UnreliableServer exception for some server instance. The DFS configuration is then composed of a number of subconfigurations corresponding to the one depicted in Figure 1.b, and of a subconfiguration corresponding to the one depicted in Figure 1.c. Subsequent exception occurrences should thus account for these various running configurations of the DFS system.

One solution consists of undertaking a solution similar to the one of the Durra environment [3]. This proposal supports nested reconfigurations, i.e., every reconfiguration defines a new configuration, which may include nested specifications of reconfigurations and hence other configurations. From our point of view, this solution alters the ease of reasoning about the software system’s behavior as brought by architectural description. The system’s software architecture gets specified in a number of places, possibly in a redundant way (e.g. consider the reconfiguration depicted in Figure 1.d following the occurrence of LowResponseTime that may apply to both configurations of Figure 1.b and Figure 1.c).

We further claim that the system’s architecture should remain compliant with the initial architecture configuration. A configuration \( C_2 \) is said to comply with another configuration \( C_1 \) if the architectural elements of \( C_2 \) may be composed into more abstract elements so that every architectural element of \( C_2 \) maps onto an element of \( C_1 \). An architectural element maps onto another one if it is a refinement of it in the sense that it enforces a stronger behavior. Hence, a refined architectural element must at least provide the same interface as the element it refines. This leads us to specify every exception handler as a set of reconfiguration actions so that the (possibly composed) elements of the resulting configuration map onto the elements of the initial one. In that way, compliance with the initial architecture is ensured and further configuration exception handling may always be achieved with respect to the initial configuration. However, we have to consider the case where the same exception occurs several times within the same subconfiguration. The resulting reconfigurations will be valid but may not be effective. Such a case is handled by disabling further handling of the configuration exception whose later occurrence is then notified to the system administrator.

Focusing now on exception handler specification, the required reconfiguration is stated in terms of refinements of the architectural elements embedded in the subconfiguration associated with the handled exception. Specifically, the declaration of a handler decomposes into the definition of: additional component and connector instances (usual COMPONENTS and CONNECTORS clauses), refinements of elements of the subconfiguration (REFINES clause that specifies the refined instance and its refinement), and possibly the DISABLE keyword to prevent further handling of the configuration exception.

Considering the DFS example, and the reconfigurations depicted in Figures 1.c and 1.d for the respective handling of UnreliableServer and LowResponseTime, we get:

**HANDLERS:**

- **EXCEPTION UnreliableServer(S: Server):**
  - COMPONENTS:
    - FJ: ForkJoin; S2: Server FROM instServ - S;
    - CONNECTORS: Rpcs1, Rpcs2: Rpc;
    - REFINES S:
      - SUBSTITUTES FJ TO S;
      - BINDS FJ.REQUIRED AS Rpc.clt TO S.PROVIDED AS Rpc.Srv USING Rpcs1;
      - BINDS FJ.REQUIRED AS Rpc.clt TO S2.PROVIDED AS Rpc.Srv USING Rpcs2;
    - DISABLE EXCEPT LowResponseTime(C: Client);
  - COMPONENTS: P: Prefetch;
  - CONNECTORS:RpcP: Rpc;
  - REFINES C:
    - SUBSTITUTES P.read TO C.read;
    - BINDS C.read AS Rpc.clt TO P.read AS Rpc.Srv USING RpcP;
    - DISABLE

The handler of UnreliableServer refines the instance \( S \) of the server component that is passed as parameter of the exception. The refinement lies in composing \( S \) with a distinct
server instance and an instance of the ForkJoin component, which duplicates the requests issued to S. The ForkJoin instance then substitutes to S in the bindings of the initial configuration, being thus bound to the client and locator instances that were bound to S (see SUBSTITUTE S1 TO S3). This additional component instance gets further bound to S and the additional server instance for all the operations provided by the Server component (see declarations following BINDS). This latter server instance is taken from the set of existing server instances but S (see S: Server FROM instSrv - S). The handling of LowResponseTime is quite direct from the above, it consists of refining the client instance so that any request to the read operation relies on a prefetching mechanism for improved performance. Due to the lack of space, we do not provide the handler of ReliableAccess. It consists of refining the two connector instances binding the client instance with the Locator instance, and with the file server instance that is considered as being not reliable enough from the perspective of the client. The refined connectors ensure that any access to this file server gets replicated on another server instance.

The above solution to the specification of handlers does not account for configuration exception handling within refined elements although these are defined in terms of configurations. For instance, one may consider introducing additional replicas for a file server instance that has previously been refined following the occurrence of UnreliableServer. Such a feature is easy to introduce by exploiting subtyping and hierarchical description of software architectures:

- The refinement of architectural elements naturally leads to the definition of subtypes. Based on the substitutability principle of subtyping, an instance may be of any type that is a subtype of the declared component type.
- The hierarchical description of software architectures enables defining component and connector types as configurations. Considering the refinement of an architectural element, this consists of introducing a configuration that is abstracted by the element and for which exception handling may be specified.

For instance, the handling of UnreliableServer can be written as:

```plaintext
HANDLERS
    EXCEPTION UnreliableServer(S: Server);
    COMPONENTS:
        Replication: ConfigRepl;
        S2: Server FROM instSrv - S2;
        SUBSTITUTE S1 SUBSTITUTES ConfigRepl[S, S2] TO S;
        DISABLE
```

1This specification assumes that the bound operations of S syntactically match with operations of FJ. In general, the operations that substitute in the bindings must be specified.

where the configuration ConfigRepl is defined as:

```plaintext
CONFIGURATION ConfigRepl(S1, S2: Server)
    REFINES Server:
       /* The configuration refines the Server component */
       /* and takes embedded server instances as parameters */
       COMPONENTS: FJ: ForkJoin;
       CONNECTORS: RpcS1, RpcS2: Rpc;
       BINDINGS:
           BINDS FJ.REQUIRED AS Rpc.clt TO
                   S1.PROVIDED AS Rpc.Srv USING RpcS1;
           BINDS FJ.REQUIRED AS Rpc.clt TO
                   S2.PROVIDED AS Rpc.Srv USING RpcS2;
       EXCEPTION HANDLING:
           Exception handling for the configuration
           PROVIDES FJ PROVIDED;
           /* The configuration can be further composed */
           /* through the interface of the FJ component */
```

4. Mapping to Implementation

The proposed exception handling facility has been integrated within the Aster environment2 that we are developing at INRIA. The overall Aster environment aims at providing methods and tools for easing the design, analysis and implementation of distributed systems from the systems' architectural descriptions. One feature of the current Aster prototype lies in the support for the systematic mapping of architectures to their implementations above middleware architectures [23]. We thus have extended this support so as to integrate the proposed architectural exception handling facility. The current version of our prototype is still preliminary in that we have been concentrating on the extension of the Aster ADL and on the provision of the base exception handling support required from the underlying runtime system. Extension to the Aster ADL is direct given our presentation of the previous section. The only difference lies in the fact that there is no explicit definition of connectors in Aster.

![Figure 2. The runtime support for exception handling](image-url)

The main constituents of the runtime support are depicted in Figure 2. These are discussed below in the context of their implementation aimed at configurations running over a CORBA-compliant middleware.

The reconfiguration service ensures that the reconfigurations performed by handlers leave the system in a consistent state.

2see http://www-rocq.inria.fr/solidor/work/aster.html.
In our prototype, we use the reconfiguration service that we built for CORBA-compliant middleware platforms [5]. This service offers a set of primitives for updating a CORBA configuration in terms of its component instances (or objects using CORBA terminology) and bindings among them, while preserving the configuration consistency.

The instances embedded in the configuration and more precisely component instances, are customized for configuration exception handling. Component instances offer the interface required by the reconfiguration service (e.g. handling requests for blocking the instance so as to enable safe reconfiguration). This is achieved through the inheritance of the class provided by the reconfiguration service. Component instances are further customized so as to notify the configuration manager about the occurrences of the interaction events that must be monitored for the detection of exception occurrences. We use here the interceptor facility of CORBA (precisely, we use the filter facility of ORBIX that is an implementation of it). Instances run interceptors that at least embed code for the notification of the interaction events stated in the definition of exceptions. Such a notification is achieved through an asynchronous RPC invocation, which carries the following detail about the interaction event: the configuration exception(s) to which it relates, destination and source, time of occurrence, message carried by the interaction event. The instances that run interceptors for the notification of interaction events are determined as follows from the definition of exceptions. Any exception event (i.e. exception events) is notified by either the (CORBA) server if the server is the exception signaler, or the (CORBA) client if the exception got raised by the broker (e.g. communication failure). For the other interaction events (i.e. IN and OUT events), the relevant instances are stated in the declaration of configuration exceptions, and hence direct to identify.

Finally, the configuration manager is the core part of the exception handling support. The manager offers an operation that processes the notifications of interaction events issued by components instances. The manager further implements an operation for each configuration exception. Such an operation is invoked upon the notification of a relevant interaction event by a component instance; it checks for the occurrence of the exception according to the specification given in the exception’s definition. Upon the occurrence of the exception, the manager interacts with the reconfiguration service for requesting the configuration changes specified in the exception handler.

Notice that the CORBA implementation of the proposed architectural exception handling is quite straightforward to automate given the reconfiguration service. First, the needed customization of the configuration components is direct from the specification of configuration exceptions. In the same way, the implementation of the operations embedded within the configuration manager for exception detection and handling can be inferred from the definitions of exceptions and handlers.

5. Related Work

The proposed exception handling facility lies in enabling the specification of exceptions at the architectural level and of the changes that need to be applied to the architecture in order to prevent further exception occurrences. To the best of our knowledge, the definition of exceptions at the architectural level has not been investigated in previous work. On the other hand, there is a number of proposals on supporting architectural evolution at runtime, which are not specifically related to exception handling. These solutions may be seen as enabling exception handling. However, we claim that exception handling must be explicitly distinguished in the specification of software system architectures, as it is already done in software implementations. In the following, we further compare our solution to the specification of architectural changes with work on the specification of architecture evolution at runtime.

As mentioned earlier in the paper, the Durra environment with its language support for specifying changes to a running configuration according to some conditions is close to our work [3]. Except the issue of targeting specifically exception handling, our proposal differs from the standpoint of how changes to the configuration are specified. Durra allows any type of changes to the running configuration, and changes over the system’s lifetime are specified through nested reconfigurations. In our solution, any change is a refinement of an architectural element, which enables maintaining compliance with the original architecture. This thus ensures that the results of the analyses performed over the architecture holds for any of its running instances. It further eases the specification of configuration changes since they can always be specified with respect to the initial reference architecture, independently of exception handling that may have been performed earlier.

Other related work that was also mentioned earlier in this paper relates to runtime support for ensuring that the system remains in a consistent state after a reconfiguration. Solutions in this area are complementary to our work in that they introduce reconfiguration services, which aim at making the reconfiguration process more efficient (e.g. [10]). In general, these proposals offer alternative implementations for the reconfiguration service that we used in our prototype.

The current trend in the design of distributed systems is to support self-adaptiveness so as to account for the evolution of the environment. An architecture-based approach is in particular introduced in [19], which addresses the evolution of the system’s software architecture. The aforementioned reference gives a general overview of the solution.
and thus does not address the expression of architectural changes. In addition, it focuses on a specific architectural style. Adaptiveness of software architectures has also been examined in [6] for middleware architectures. This work that has been partly realized in the context of the Aster project, complements the proposed architectural exception handling facility by addressing constrained changes to the underlying runtime system according to environmental parameters. While exception handling is concerned with the treatment of failures with respect to the specifics of the software system, additional support for fault tolerance may be integrated at the level of the underlying runtime system, which may be realized dynamically using the above solution.

The last piece of work that relates to ours is the effort on specifying dynamic software architectures. The Darwin ADL allows specifying architectures whose elements may only be known at runtime [15]. The specification of the software architecture then gives the most general structure of the system where some of its components may be attributed with the dynamic keyword, meaning that those components are dynamically integrated within the architecture. Our solution differs in that the evolution of the architecture is coupled with the specification of handlers for separation of concerns. Notice that the initial DFS configuration is also a dynamic architecture with respect to the embedded instances of client and server components. However, the system’s base structure is invariant and corresponds to the one depicted in Figure 1.b. Another approach to specifying dynamic software architectures has been proposed in [13]. In this solution, a software architecture is specified using a graph grammar, and the architecture evolution is specified within a coordinator in terms of conditional graph rewrite rules. This work focuses on formal specification of dynamic architectures so as to enable checking consistency of the modified architecture with respect to the architectural style. Our solution is more practical in that we are concerned with a solution enabling mapping the architecture to an implementation. However, we also address consistency with the initial architectural style by constraining architecture evolution through the refinement of the elements of the initial architecture. The reference [1] proposes a way to specify dynamism in software architectures in the Wright ADL [2]. This solution lies in specifying the behavior of a reconfiguration program, which depends on the events generated by the architectural elements. As for the previous work, this one concentrates on the analysis of the architecture behavior rather than on its implementation. Our solution provides a more pragmatic approach to the specification of architecture evolution. It also allows for the analysis of the architecture behavior given the precise descriptions of exceptions and handlers, although translation in a convenient formal framework remains to be done.

6. Conclusion

Results in the software architecture field contribute to easing the design and implementation of robust software systems. By focusing on the software system at a high level of abstraction, formal methods may practically be exploited for reasoning about the properties of the system even if it is a complex one. In addition, tools are provided for mechanizing the mapping of the architecture to an implementation. Robustness of a software system further requires to account for possible failures occurring at runtime. In general, the handling of failures is achieved through the use of fault tolerance mechanisms within both the system’s software implementation (i.e., exception handling and possibly versions programming) and the underlying runtime system. With an architectural focus for software development, the former issue must be addressed through at least exception handling at the architectural level. However, little attention has been paid to this issue in the software architecture community, which is treated by relying on the exception handling mechanisms implemented within the components. Existing support for dynamic reconfiguration may further be exploited when available but this support is independent of exception handling. This paper has introduced a base solution towards enabling exception handling at the architectural level. The proposed solution consists in the combined specification of the exceptions requiring changes to the current running configuration and of their handlers. The solution complements but does not substitute to the exception handling implemented within architectural elements since they serve distinct purposes: architectural exception handling consists of changing the system’s configuration for preventing further occurrence of raised exceptions; exception handling within architectural elements implements the exceptional specifications of the elements.

The proposed exception handling support has been designed so as to maintain the ease of reasoning about the system’s behavior that is enabled by software architecture description. However, the proposed specification remains at the level of structural description and does not include precise behavioral information, hence preventing direct behavioral analysis. Our objective is to extend the proposed specification of exception handling so as to enable behavioral analyses, possibly with the aid of CASE tools. We intend to exploit here the various results from the software architecture community in the area of architecture specification based on formal methods. Behavioral specification needs also investigation regarding exception handling implemented within architectural elements so as to guarantee consistent handling among interacting elements (e.g., with respect to the underlying exception handling model). Another area of future work relates to the implementation of architectural exception handling. We have implemented a
first prototype so as to gain confidence in the practicality of our solution. However, our prototype is preliminary and the implementation of exception handling has been done by hand for a specific software system. Only the management of architecture reconconfigurations relies on a generic service, which is application-independent but aimed at CORBA middleware. We are working on the automation of the exception handling implementation from its specification. A significant part of it can already be automated quite straightforwardly. The open issue that remains relates to the specification of the conditions of configuration exception occurrences, which requires further investigation regarding its expressiveness for various applications.

Acknowledgments: This work has been supported by the European IST DSoS project. The authors would like to thank Apostolos Zarras for helpful discussions about the work presented in this paper.

References