Architectural Mismatch Tolerance

Rogério de Lemos[1], Cristina Gacek[2], and Alexander Romanovsky[2]

[1]Computing Laboratory
University of Kent at Canterbury, UK
r.delemos@ukc.ac.uk
[2]School of Computing Science
University of Newcastle upon Tyne, UK
{cristina.gacek, alexander.romanovsky}@ncl.ac.uk

Abstract. The integrity of complex software systems built from existing components is becoming more dependent on the integrity of the mechanisms used to interconnect these components and, in particular, on the ability of these mechanisms to cope with architectural mismatches that might exist between components. There is a need to detect and handle (i.e. to tolerate) architectural mismatches during runtime because in the majority of practical situations it is impossible to localize and correct all such mismatches during development time. When developing complex software systems, the problem is not only to identify the appropriate components, but also to make sure that these components are interconnected in a way that allows mismatches to be tolerated. The resulting architectural solution should be a system based on the existing components, which are independent in their nature, but are able to interact in well-understood ways. To find such a solution we apply general principles of fault tolerance to dealing with architectural mismatches.

1 Introduction

Software architecture can be defined as the structure(s) of a system, which comprise software components, the externally visible properties of those components and the relationships among them [18][20]. A software architecture is usually described in terms of its components, connectors and their configuration [15]: components represent computation units, connectors correspond to the communication protocols, and configurations characterize the topology of the system in terms of the interconnection of components via connectors.

As a result of combining several architectural elements using a specific configuration, architectural mismatches may occur [11]. *Architectural mismatches* are logical inconsistencies between constraints of various architectural elements being composed. An architectural mismatch occurs when the assumptions that a component makes about another component, or the rest of the system, do not match. That is, the assumptions associated with the service provided by a component are different from the assumptions associated with the services required by a component for behaving as specified [15]. These assumptions can be related to the nature of components and connectors (control and data models, and synchronization protocols), the global system structure, or the process of building the system [11][20]. Traditionally, mismatches have been dealt with statically [8][10], by means of analysis and removal. For example, a formal approach has been advocated to uncover architectural mismatches in the behavior of components, in particular, deadlocks [4][12].

There are many reasons to support our claim that it is usually non-practicable to statically localize and correct all possible architectural mismatches, and because of this, we believe that it is vital to be able to build systems that can tolerate such mismatches. This is mainly due to the complexity of modern systems and restricted applicability of the static methods of correcting mismatches (c.f. software design faults). First of all, complex applications have complex software architectures in which components are interconnected in complex ways and have many parameters and characteristics to be taken into account, and they have to meet many functional and non-functional requirements that often have to be expressed at the level of software architecture. Secondly, architects make mistakes while defining software architectures, in general, and while dealing with mismatches, in particular. Thirdly, there is a strong trend in using off-the-shelf elements while building complex applications and because of the very nature of such elements some information about their architectural characteristics may be unavailable. Lastly, modern software systems are to be open, flexible and adaptive, and they may undergo dynamic reconfiguration (often by incorporating new components knowledge about which is not available offline), adding uncertainty about the various architectural elements present at any point in time.

Instead of dealing with architectural mismatches during development time, which is the conventional approach, this paper shows how these mismatches can be tolerated during runtime at the architectural level. The rest of the paper is structured as follows. Section 2 discusses architectural mismatches in the context of features that are associated either with the architectural elements or the application being represented by these elements. In Section 3, we present some basic dependability concepts that provide the basis for the following discussion (in Section 4) on architectural mismatches from the perspective of system dependability. In Section 5, we address the notion of mismatch tolerance by discussing in more detail its basic activities. Section 6 presents several simple examples that demonstrate the proposed approach. Finally, Section 7 concludes with a summary of the contribution and a perspective of future research.

2 Architectural Mismatches and Features

To understand better the ways of tolerating architectural mismatches we will look first into specific characteristics of the individual architectural elements to be composed into a system, as well as into the reoccurring architectural solutions (i.e. architectural styles) applied for building system architectures.

2.1 Architectural Features

The architectural mismatches occur because of inconsistencies among given architectural elements. These inconsistencies can be stated in terms of the features (i.e. characteristics or properties relevant to system composition) exhibited by the architectural elements to be integrated into the system. Such features have proven to be very useful for static mismatch detection [10].

A considerable number of such features have been determined while studying system composition from the viewpoint of detecting architectural mismatches during system development to allow systems to be corrected by removing mismatches [10]. Concurrency, distribution, supported data transfers (e.g. via shared data variables, explicit data connectors, shared repositories, etc.), dynamism (system ability to change dynamically its topology), encapsulation (provision of well-defined interfaces), layering, backtracking, and reentrance are some of the examples of the architectural features relevant to possible mismatches. A very useful source of such features can be found in research on online system upgrading, where, for example, additional (meta-) information describing component behavior is used to deal with component interface upgrades [13].

Another dimension of the analysis proposed in [10] is relevance of such features to the particular architectural styles (such as pipe-and-filter, blackboard, etc.) employed in building system architecture: it is clear that some of these features are not applicable to some particular styles. For example, the pipe-and-filter style assumes multithreaded concurrency, no backtracking or reentrance, while the blackboard style assumes backtracking, imposes no restrictions on types of concurrency and assumes no reentrance. The overall idea here is that by analyzing the characteristics of the architectural elements to be integrated and the styles from which these elements were derived, the system architects are able to localize architectural mismatches earlier in the life cycle.

Some examples of architectural mismatches that can be detected by analyzing the architectural features are [10]:

- *Data transfer from a component that may later backtrack* – this mismatch may cause undesired side effects on the overall composed system state.
- *Call to a non-reentrant component* - this mismatch may happen when system composition is achieved via a bridging (triggered) call, and the callee is not reentrant and is already running at the time of the call.
- *Sharing or transferring data with different underlying representations* - this mismatch happens when sharing or transferring data with different underlying representations, including differences in data formats, units and coordinate systems.

2.2 Style-specific and Application-specific Mismatches

Architectural features of architectural elements and their groupings may be inherent to the architectural style(s) used, or specific to the application at hand. This occurs because architectural styles impose constraints on the kinds of architectural elements that may be present and on their configurations [20], yet they do not prescribe all the features that may be present in an application [10]. During software development, the software architecture is incrementally refined following the refinement of the system definition. Initially, the software architecture is defined in terms of architectural styles, thus binding the style-specific features. Subsequently, as the architecture is further refined towards the life-cycle architecture, application-specific features are bound. This is exemplified on Table 1 (adapted from [9]). In the following we will refer to the architectural features pertinent to particular architectural styles as *style-specific features*. A set of such features is defined in [10]. The features that are defined by the characteristics of the application to be developed but not by the architectural styles employed are called *application-specific features*.

Every time an architectural feature is bound, there is a potential for an architectural mismatch to be introduced. Hence, we refer to architectural mismatches as being:

- *style-specific* - if their presence is brought about by some architectural feature(s) that the style(s) imposes, or
- *application-specific* - if their presence is due to architectural decisions imposed by the application at hand but not the particular style(s) used.

	Early Cycle 1	**End of Cycle 1**	**Cycle 2**	**Cycle 3**
Definition of operational concept and system requirements	Determination of top-level concept of operations	Determination of top-level concept of operations	Determination of detailed concept of operations	Determination of IOC requirements, growth vector
Definition of system and software architecture	System scope/ boundaries/ interfaces	System scope/ boundaries/ interfaces	Top-level HW, SW, human requirements	Choice of life-cycle architecture
Elaboration of software architecture	*No explicit architectural decision*	*Small number of candidate architectures described by architectural styles*	*Provisional choice of top-level information architecture*	*Some components of above TBD (low-risk and/or deferrable)*
Binding of architectural features	*No architectural features explicitly defined*	*Fixed architectural features that are defined by architectural styles, others are unknown*	*Architectural features defined by architectural styles are fixed as are some application specific ones, others are unknown*	*Most architectural features are fixed, the few unknown ones relate to parts of the architecture still to be defined*

Table 1. Refinement of software architecture under a Spiral Model Development.

Identification of the nature of the architectural mismatches, as well as the nature of the architectural features causing these inconsistencies among architectural elements plays a vital role in developing approaches to tolerating such mismatches.

3 Dependability

Dependability is a vital property of any system justifying the reliance that can be placed on the service it delivers [14]. The causal relationship between the dependability impairments, that is, faults, errors and failures, is essential for characterizing the major activities associated with the dependability means (fault tolerance, avoidance, removal and forecasting). A *fault* is the adjudged or hypothesized cause of an error. An *error* is the part of the system state that is liable to lead to the subsequent failure. A *failure* occurs when a system service deviates from the behavior expected by the user.

Fault tolerance is a means for achieving dependability working under assumptions that a system contains faults (e.g. ones made by humans while developing or using systems, and caused by aging hardware) and aiming at providing the required services in spite of them. Fault tolerance is carried by error processing, aiming at removing errors from the system state before failures happen, and fault treatment, aiming at preventing faults from being once again activated [14].

Error processing typically consists of three steps: error detection, error diagnosis and error recovery. *Error detection* identifies an erroneous state in the system. *Error diagnosis* assesses the damage caused by the detected error, or the errors propagated before detection. *Error recovery* transforms a system state that contains errors into an error free state. Recovery typically takes forms of either backward error recovery or forward error recovery. When the former is applied the system is returned to a previous (assumed to be correct) state; the typical techniques used are application-independent and often work transparently for the application (e.g. atomic transactions and checkpoints). Forward error recovery intents to move the system into a correct state using knowledge about the current erroneous state; this recovery is application-specific by its nature. The most general means for achieving it is exception handling [4].

Fault treatment consists of two steps: fault diagnosis and system repair. *Fault diagnosis* determines the causes of the error in terms of both location and nature. *System repair* consists of isolating the fault to avoid its reactivation, reconfiguring the system either by switching on spare components or reassigning tasks among non-failed components, and reinitializing the system by checking, updating and recording the new configuration [1]. The process of repairing the system usually modifies its structure in order for the system to continue to deliver an acceptable service.

Providing system fault tolerance plays an ever-growing role in achieving system dependability as there are many evidences proving that it is not possible to rid the system and system execution from faults. These include the growing complexity of software causing programmers' bugs, operators' mistakes, and failures in the environment in which the system operates.

4 Dependability and Mismatches

In the context of dependability, an architectural mismatch is an undesired, though expected, circumstance, which must be identified as a *design fault* (in the terminology from [14]). When a mismatch is activated, it produces an *error caused by mismatch* (ECM) that can either be latent or detected. Similarly to errors, only a subset of ECMs can be detected as such (see Figure 1). Additional information is needed to allow an error to be associated with a mismatch. Eventually, there is a system failure when the ECM affects the service delivered by the system.

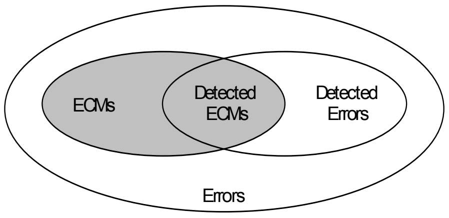


Fig. 1. Detected errors caused by mismatches

For describing the means for dealing with architectural mismatches, we draw an analogy with faults, which can be avoided, removed or tolerated. Faults are tolerated when they cannot be avoided, and their removal is not worthwhile or their existence is not known beforehand. The same kind of issues happens with architectural mismatches. Mismatches can be *prevented* by imposing strict rules on how components should be built and integrated. Mismatches can be *removed* when integrating arbitrary components by using static analysis methods and techniques [10]. However, this does not guarantee the absence of mismatches since risk and cost tradeoffs may hinder their removal, or system integrators may not be aware of their existence (similarly, research has shown that residual faults in software systems are inevitable). Consequently, mismatches should be *tolerated* by processing ECMs and treating mismatches, otherwise the system might fail.

In the following, before presenting mismatch tolerance, we discuss in more detail what is mismatch prevention and mismatch removal.

4.1 Mismatch Prevention

The approaches associated with mismatch prevention attempt to protect a component, or the context of that component, against potential mismatches by adding to the structure of the system architectural solutions. The assumption here is that the integrators are aware of all incompatibilities between system components. For example, if the architectural style of a particular component does not fit the style of the system in which is to be integrated, then a specialized wrapper can be developed as a means of fixing architectural incompatibilities [18].

There are three classes of structuring techniques for dealing with architectural mismatches, all of which are based on inserting code for mediating the interaction between the components [7]:

- *Wrappers* – which are a form of encapsulation whereby some component is enclosed within an alternative abstraction, thus yielding to an alternative interface to the component;

- *Bridges* – which translate some of the assumptions of the components interfaces. Different from a wrapper, a bridge is independent of any particular component, and needs to be explicitly invoked by an external component;
- *Mediators* – which exhibit properties of both wrappers and bridges. Different from a bridge, a mediator incorporates a planning function that results in the runtime determination of the translation. Similar to wrappers, mediators are first class software architecture entities due to their semantic complexity and runtime autonomy.

4.2 Mismatch Removal

The approaches associated with mismatch removal are those that aim at detecting architectural mismatches during the integration of arbitrary components [10]. Existing approaches for identifying architectural mismatches are aimed for the development of software, either during the composition of components while evaluating the architectural options [10], or during architectural modeling and analysis [8]. The Architect's Automated Assistant (AAA) approach uses automatic static analysis for performing early risk assessment for the purpose of detecting mismatches during component composition [10]. It is an approach that supports rapid evaluation of components with respect to potential incompatibilities among them. The software integrator gathers the information for the analysis, known as architectural features, from the system requirements and the specification of the components. On the other hand, the technique for architectural modeling relies on the specification of component invariants and services for analyzing the architectural conformance of its components. For example, the behavioral conformance of the pre- and post-conditions of two components can be analyzed using a model checking tool [8] [15]. The above two techniques, evaluation of architectural options and architectural modeling, are argued to be complementary because the former is able to detect mismatches very early during development, while the latter performs a more detailed and precise analysis of component mismatch.

The techniques being proposed by these approaches are so specific to the context of software development that they cannot be transposed for runtime detection of error caused mismatches (ECMs). For example, how can we detect during runtime whether components have single or multiple threads, and how can we identify inconsistencies between the pre- and post-conditions among operations of interacting components? However, although it is difficult, in general terms, to relate the detection of errors to specific architectural mismatches that have caused them, it is nevertheless feasible to associate some (detectable) errors to architectural mismatches that may occur in the components' behavior, their interfaces, or interaction protocols. For example, a mismatch may occur in the naming of an operation or message, or in the number, ordering, type, and units of parameters [21].

5 Architectural Mismatch Tolerance

The main motivation for specifying mechanisms for tolerating architectural mismatches at the architectural level, instead of the implementation level, for example, is that the nature of mismatches and the context in which they should be fixed would be lost at the later stages of software development. Making an analogy with fault tolerance, it has been shown that the same type of problem exists when exception handling is not considered in the context of the software life cycle [6]. Moreover, we cannot expect that a general runtime mechanism would be able to handle a wide range of architectural mismatches, in the same way as there is no sufficiently general fault tolerance mechanism that can handle all classes of faults. It is envisaged that different classes of architectural mismatches will require different types of detection mechanisms and fixes that have to be specified at the architectural level.

Although the goal is to tolerate architectural mismatches at the architectural level, it is nevertheless necessary to deal with two levels of abstraction: the architectural level, where the mismatches are actually introduced, and the execution level, where ECM processing and mismatch treatment take place. *ECM processing* comprises three steps [14]:

- *Detection of ECMs*, which identifies erroneous states that are caused by mismatches;
- *Diagnosis of ECMs*, which assesses the system damages caused by the detected ECMs;
- *Recovery from ECMs*, which brings the system to an ECM-free state.

However, ECM processing is not sufficient if we would like to avoid the recurrence of the same architectural mismatch, so there is the need to treat mismatches, in the same way as faults are treated [14]. *Mismatch treatment* involves two major steps:

- *Mismatch diagnosis*, which determines the cause (localization and nature) of the ECM;
- *System repair*, which prevents a new activation of the architectural mismatch; it is performed by isolating the mismatch, and reconfiguring and reinitializing the system, in order to continue to provide an adequate, perhaps degraded, service.

The intent of fault tolerant techniques is to structure systems to inhibit the propagation of errors, and to facilitate their detection and the recovery from them. Similarly, when dealing with architectural mismatches, there is the need to structure systems at the architectural level in a way that prevents propagation of ECMs, facilitates ECM detection and recovery, and makes it difficult for the architectural mismatches to be reactivated.

In addition to system structuring, there is also the need for documenting architectural features of the system, as discussed in Section 2. This information is fundamental for distinguishing ECMs from other system errors, architectural mismatches from faults, and for choosing features suitable for tolerating style- and application-specific architectural mismatches. If little or no information is made available at the architectural level, either as interface properties of architectural elements or error codes, then this distinction cannot be characterized. For example, if there is no information about the types of data transferred between two architectural elements but the producer and the consumer assume different types (e.g. measurement units) the following situations are possible:

- An error is detected by the consumer but because there is not enough information it cannot be identified as an ECM, so unsuitable fault tolerance measures are applied (e.g. rollback);
- An error is further propagated outside the consumer and detected by other components. In this case without additional information it is impossible to identify the damage area to be recovered;
- The ECM is not detected and the system fails to deliver the service.

In order to provide the basis for defining an architectural solution for tolerating mismatches, in the rest of this section we present in more detail the activities associated with ECM processing and mismatch treatment. For each of the activities, we take into consideration whether architectural features, both style- and application-specific, are incorporated into the architectural description of a system.

5.1 ECM Processing

As previously discussed, the detection of an ECM implies the presence of an architectural mismatch. The activation of a mismatch causing a system error depends on whether some conditions are satisfied, these conditions are related to inconsistencies in architectural features. In the following, we present in more detail the different activities associated with ECM processing.

5.1.1 ECM Detection

Upon error detection, one must first determine whether that particular error can be identified as an ECM. For an error to be detected as an ECM we need additional information at runtime about the system states and the features of the relevant architectural elements that would enable to identify this particular error as an ECM. This ought to be done based on the detected error and on the presence of the conditions required for activating the architectural mismatch.

The identification of an error as an ECM will facilitate the process of error recovery, in particularly if the error can be differentiated as being either caused by an application- or style-specific mismatch. For both types of ECMs, error codes should be provided as an outcome of a failed operation, and these codes should be related to architectural features of the system (as it will be seen in the examples in Section 6). Provision of an error code to an error caused by a style-specific mismatch could be related to the execution of an operation that violates the properties of an architectural notation, for instance, when in a non-reentrant pipe-and-filter architecture a filter sends data to another filter that is already processing data from other source. On the other hand, provision of an error code to an error caused by an application-specific mismatch could be related, for example, to the semantic discrepancy of data received from other component; this error code should help, for instance, identify that the data received has the wrong type, such as, instead of receiving a value in meters, the value is in feet.

Identification of a system error as an ECM is not essential if provisions are made in the later stages of mismatch tolerance for processing the error and treating the fault accordingly. However, the later an error is identified as an ECM or a fault as a mismatch, the more costly and more uncertain (mainly in its successful outcome) the respective processes of recovery and repair are. One of the techniques that can be used for detecting ECMs is executable assertions.

5.1.2 ECM Diagnosis

The purpose of ECM diagnosis is to assess the damages caused by the detected ECM. During damage assessment it is necessary to identify all the erroneous states of the system before initiating recovery from the ECM, for this purpose there is no need to differentiate system errors from ECMs. If an ECM is not detected close to where it is activated, the propagation can render impossible the error recovery. This is usually the case for errors caused by an application-specific mismatch. The propagation of such ECM to other architectural elements depends on the encapsulation properties of the architectural language used to describe the system. Ideally the error should be contained within the component where the mismatch is activated. On the other hand, an error caused by a style-specific mismatch is more capable of affecting the whole architectural configuration of a system than a single component due to the lack of diversity in the style-specific features of the architecture. For example, in a blackboard architecture where only some of the components are able to backtrack, the impact of a component backtracking has to be assessed in the context of the whole system architecture to identify which components' states might have been affected by the backtracking. For both style- and application-specific mismatches, the process of damage assessment can be performed either by using static or dynamic techniques [1].

5.1.3 ECM Recovery

The purpose of ECM recovery (which can be one of the form: backward, forward or compensation, as well as their combination) is to replace at the architectural level an ECM state by an error-free state. The level of difficulty encountered for recovering from ECMs very much depends on the specific characteristics of the ECM, the application, and the error containment capabilities of the architectural style.

In general terms, the type of ECM, whether style- or application-specific, should dictate the choice of recovery form. For errors caused by style-specific mismatches, backward recovery is more appropriate because they are application independent and require general approaches for recovering. If the architecture provides adequate error containment capabilities, ECM recovery may consist of eliminating existing erroneous states within an architectural element, this can be done by rolling back to an error-free state that the element had prior to the detection of the ECM. For example, if a component semantically checks the information it provides to other components for potential errors then it can be assumed that errors that might occur within the component are not propagated to the rest of the system. On the other hand, if the architecture does not provide adequate error containment capabilities, then the recovery at the architectural element level might not be sufficient, and there is the need to have a coordinated recovery involving several system components and connectors. For example, if a component needs to rollback and there are other components in the system that cannot rollback then some system coordination might be needed to rid the system of the ECMs.

For errors caused by application-specific mismatches, forward recovery is more appropriate since knowledge about the application allows bringing the system into a new (correct) state from which the processing can resume. In particularly forward recovery in the form of exception handling can be used for dealing with those errors that are anticipated. For example, if a component detects a semantic discrepancy in the value of a variable that is transferred by other component with a different underlying representation, then the component can calculate a new value (assuming it knows the correct underlying context, which, again, can be documented in a form of corresponding architectural features), and resume normal processing.

In those cases where we cannot distinguish whether the ECM is either style- or application specific, or even an error cannot be identified as an ECM, error recovery should follow a general approach based on backward error recovery. In these situations, as in all those in which not enough information is provided for supporting process of tolerating a mismatch, error recovery often becomes intrinsically complex.

5.2 Mismatch Treatment

The treatment of mismatches aims to avoid mismatches from being further activated once their nature and location have been identified. As an activity following ECM processing, mismatch treatment attempts to avoid the re-activation of mismatches. If enough information regarding architectural features is made available as the interface properties of architectural elements, the process of tolerating mismatches might be reduced to mismatch repair. This can be achieved if, before any operation, architectural elements check for potential mismatches by requesting information about the architectural features of the other elements. After a potential mismatch is localized it should be repaired. For example, in a pipe-and-filter architecture, if a filter before sending its data checks for the status of the other filter and detects that the other filter is already receiving data from other source, then an alternative filter that is able to provide the same kind of services could be sought in the system.

As we have already seen in the descriptions of previous activities, the treatment of mismatches depends on whether the relevant architectural features are style- or application-specific. For example, as we will show later, mismatches caused by incompatibilities in the style-specific features of an architecture often require more fundamental changes to the system architecture at hand. In the following, we present in more detail the different activities associated with mismatch treatment, considering again style- and application-specific mismatches.

5.2.1 Mismatch Diagnosis

The purpose of mismatch diagnosis is to determine the cause of ECMs, in terms of both location and nature, which, in particular, means identification of the architectural elements that failed and the way they failed. This activity is fundamental for the process of mismatch repair since a clear identification of the mismatch is needed before any changes are made on the system architecture. The activity of diagnosis is complicated by the fact that it often requires a lot of information from the system and elaborate means to process this information. The types of information that are necessary: the detected erroneous state (which presumably is cause by a mismatch), the overall state of the system when the ECM is detected, the configuration of the architecture, together with the available information on architectural features. The latter, in particular, provides the means for identifying the nature of the architectural mismatches. Although it is important to known whether a mismatch is style- or application-specific, the identification of the type of mismatch among a list of potential mismatches [10] is equally necessary for selecting the appropriate repair for the architecture.

5.2.2 Mismatch Repair

The purpose of mismatch repair is to prevent mismatches from being activated again. Since each mismatch is caused by incompatibilities between features of architectural elements (mainly components), the repair of this mismatch can be performed by modifying the system structure. This architectural reconfiguration is performed in runtime, and it is not a simple task as in most cases it requires redundant architectural elements that are intrinsically diverse[1] in the way they provide architectural features (both application- and style-specific), since mismatches are design faults. The reconfiguration can be performed in various forms: removal or/and addition of a single component, removal of all the components involved in a mismatch (e.g., a particular architectural style-specific mismatch), replacement of the connector linking the problematic components with a new connector with additional functionalities aiming at avoiding mismatches. For example, in the case of a component that is not able to rollback, this component can be replaced by other component that allows rollback, or an alternative connector can be provided that allows information to be buffered.

The dichotomy between style- and application-specific mismatches for system repair is difficult to observe since for repairing some style-specific mismatches it is necessary to rely on application level mechanisms and techniques. In these cases, simple replacing an architectural element is not a viable option due to the lack of diversity in the features of the architectural style, which creates inherent difficulties in repairing some style-specific mismatches. In terms of application-specific mismatches, the repair mechanisms and techniques are essentially application related and as such should exploit available redundancies at the application level.

Although the general aim of mismatch repair is to find and employ mechanisms and techniques that are sufficiently general to allow dealing with a wide range of mismatches, in real systems this is difficult to achieve because of three main reasons: mismatches of different types require different ways of reconfiguration and different types of redundant elements, very often not enough system redundancies can be made available, and the most effective way of performing repair is application dependent.

[1] By diverse elements we mean here architectural elements that provide the same functionality but have different designs and implementations.

Summarizing, in order for the system to tolerate architectural mismatches, it is crucial that the information associated with the architectural features (either style- or application-specific) is documented and encoded in the system in different forms, either as error codes for performing activities associated with ECM processing, or as interface properties for the activities associated with mismatch treatment. If enough information is made available, then the process of tolerating architectural mismatches becomes less complex and less prone to faults. In the following, we demonstrate through examples the different activities associated with mismatch tolerance, including the cases in which the whole process can be improved, sometimes by suppressing some of the activities, when suitable architectural features are exploited during runtime.

6 Examples

This section demonstrates how mismatches can be tolerated following the framework discussed above. From the whole set of potential architectural mismatches discussed in [10], we have selected three mismatches, which are representative of the different types of mismatches and allow us to show different ways of tolerating them. In order to analyze the particularities associated with style-specific mismatches, the examples will be presented in the context of three architectural styles [20]: pipe-and-filter, blackboard, and client-server.

Our assumption here is that some of the non-functional properties/attributes of components (in particular, ones related to the architectural features) are published at their interfaces. Depending on the information available and on the way it is processed, we can distinguish three general scenarios in which architectural mismatches can be tolerated.

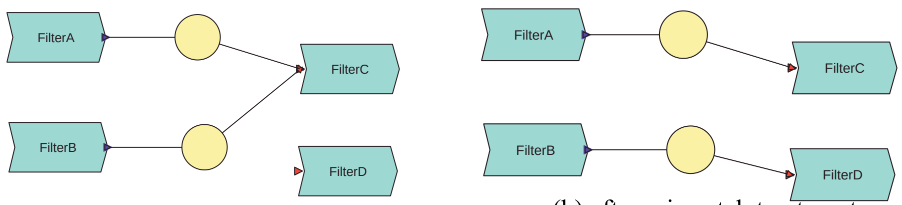
1. The first scenario falls into the category in which mismatch tolerance is restricted to mismatch repair. The basis for this scenario is the above assumption that features of the architectural elements are provided at their interfaces. The availability of this information allows for a component before engaging into an interaction, trying to identify potential mismatches. If an architectural mismatch is localized, then the activities associated with error processing and mismatch diagnosis can be ignored.
2. The second scenario falls into the category in which mismatch tolerance starts with ECM detection. During the interaction between two components when one of them returns an error diagnosed as an ECM, there is no need for the system to perform error diagnosis. Once again, for this to be possible it is necessary that additional information, about its architectural features, be made available at the interface of the architectural components.
3. The third scenario falls into the category in which mismatch tolerance starts with detection of a system error. In this scenario, there is no additional information available about the architectural features. Hence, there is a need to perform error diagnosis to identify the nature of the error as being an ECM, for that, additional information is needed about the state of the system.

In the following, we discuss how three different mismatches can be tolerated in the context of the above three scenarios. For each example of architectural mismatch, we present the type of mismatch, the characteristics of the architectural styles being used, and provide a small architectural configuration capturing the mismatch being discussed. Within this context, we proceed to explain, in detail, the different steps associated with the process of tolerating architectural mismatches.

6.1 Style-specific Mismatch

As a style-specific mismatch, we consider the mismatch "*call to a non-reentrant component*". This mismatch happens when a component calls another component, and the latter may already be performing the requested execution and this execution is not reentrant [10]. This mismatch will be analyzed in the context of the pipe-and-filter style in which data is transformed by successive components. The components are the filters, which have almost no contextual information and retain no state between executions, and the connectors are the pipes, which are stateless [20].

An example of the pipe-and-filter architectural configuration in which the above mismatch can occur is shown in Figure 2(a), when FilterA calls FilterC without waiting until the currently executed request from FilterB is completed. In this example we assume that a Unix environment is used, which can provide a runtime documentation related to the appropriate style-specific features. For implementing the first scenario we have to ensure that the resources/ports become exclusive before executing any interaction between the filters. Implementation of the second scenario assumes that FilterB always executes additional (application) code before connecting to FilterC. The third scenario is not applicable here because FilterB always receives information based on the Unix error while accessing a busy filter, so we will not consider it any further.



(a) before mismatch treatment (b) after mismatch treatment

Fig. 2 A non-reentrant component in a pipe-and-filter architecture.

This is how ECM processing and mismatch treatment (see Section 5) look like for the two remaining scenarios:

a) ECM Detection:

- for scenario 1, no detection needed;
- for scenario 2, such an ECM is detected using the error code generated by Unix.

b) ECM Diagnosis:

- for scenario 1, no diagnosis needed;

- for scenario 2, during damage assessment there is no need to identify the nature, since the error code provides enough information.

c) ECM Recovery:

- for scenario 1, no recovery required;
- for scenario 2, different ways of recovering are possible depending on the application (e.g., backward or forward recovery, compensation, and recovery employing time redundancy).

d) Mismatch Diagnosis:

- for both scenarios, the error code should provide enough information to identify the location and the nature of mismatch;

e) Mismatch Repair:

- for both scenarios, system reconfiguration involves the usage of alternative component or connector. The way of reconfiguring the system depends on the application characteristics. Some possible scenarios are as follows: FilterB can switch to using an alternate FilterD (as shown in Figure 2(b)); it can create another instance of FilterC and switch to using it; it can use a different type of pipe (e.g. a timed pipe); in the situation when FilterB has a higher priority than FilterA, FilterC could be killed. In Unix environments, a script can execute this repair.

6.2 Application-specific Mismatch

As an application-specific mismatch, we consider the mismatch "*sharing or transferring data with differing underlying representations*". This mismatch occurs when communication between two components concerning a specific data cannot happen because the data being shared or transferred has different underlying representations, which might include, different data formats, units and coordinate systems [10]. This mismatch will be analyzed in the context of the client-server style, which is representative of data abstract systems in which a component – the server, provides services to other components – the clients.

An example of the client-server architectural configuration in which the above mismatch can occur is shown in Figure 3(a), when Client1 requests a service from ServerA but provides a value in feet while the server requires it to be in meters. For implementing the two first scenarios, we assume that the application-specific features of both components contain information about the units being used. In the first scenario, this information is part of the interface of the server, while in the second scenario this information is part of the services provided by the server application. The implementation of the third scenario does not assume any additional information is provided.

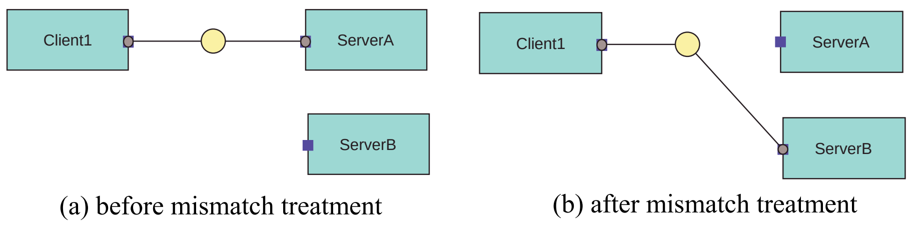


Fig. 3 Components with different underlying representations in a client-server architecture.

This is how ECM processing and mismatch treatment look like for the three scenarios:

a) ECM Detection:

- for scenario 1, no detection needed;
- for scenario 2, an ECM is detected from the application-specific features of the component;
- for scenario 3, a system error is detected.

b) ECM Diagnosis:

- for scenario 1, no diagnosis needed;
- for scenario 2, during damage assessment there is no need to identify the nature, since the error provides enough information.
- for scenario 3, a full damage assessment is necessary.

c) ECM Recovery:

- for scenario 1, no recovery required;
- for scenarios 2 and 3, different ways of recovering are possible depending on the application (backward or forward recovery, or compensation).

d) Mismatch Diagnosis:

- for scenarios 1 & 2, the error should provide enough information to identify the location and the nature of mismatch;
- for scenario 3, a full mismatch diagnosis is necessary.

e) Mismatch Repair:

- for all the scenarios, system reconfiguration involves the usage of alternative component or connector. One of the ways to repair is for Client1 to request services of another ServerB (as shown in Figure 3(b)), which allows processing the request in feet. Another way to repair is to introduce a bridge that performs unit transformation.

6.3 Style- and Application-specific Mismatch

As an application-specific mismatch, we consider the mismatch "*call or spawn from a subsystem that may later backtrack*". This mismatch occurs when after a component transfers data to other components, it backtracks, which might cause some undesired side effects. This mismatch will be analyzed in the context of the blackboard style, which allows building an active repository for sharing and transferring data between clients that run of independent threads [20]. Blackboard systems support backtracking, but they are neither reentrant nor preemptive.

An example of a blackboard architectural configuration in which the above mismatch can occur is shown in Figure 4, when Client1 attempts to backtrack, which is permitted by the BlackboardA, BlackboardB, and Client3, but not by Client2. For implementing the first and the second scenarios, it is necessary to have additional information on the ability of each component to backtrack, on the fact that backtracking is initiated by a component, and on a set of interconnected components to be involved in backtracking. In the first scenario, this information is part of the interfaces of the components, while in the second scenario this information is part of the services provided by the application. The implementation of the third scenario does not assume that any additional information is provided.

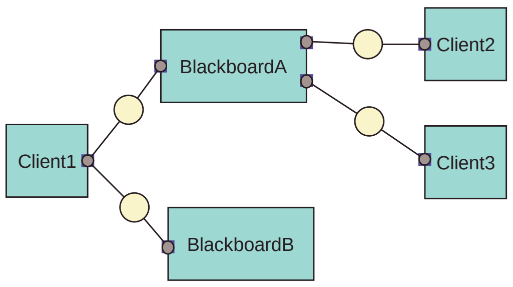


Fig. 4 Components that cannot backtrack in the blackboard architecture.

This is how ECM processing and mismatch treatment look like for the three scenarios:

a) ECM Detection:

- for scenario 1, no detection needed;
- for scenario 2, an ECM is detected from the application-specific features of the components;
- for scenario 3, a system error is detected.

b) ECM Diagnosis:

- for scenario 1, no diagnosis needed;
- for scenario 2, during damage assessment there is no need to identify the nature, since the error provides enough information;
- for scenario 3, a full damage assessment is necessary, which might have affected all the system components.

c) ECM Recovery:

- for scenario 1, no recovery required;
- for scenarios 2 and 3, only forward recovery is possible because recovery is application dependent (backward recovery does not apply because one of the components is not able to backtrack).

d) Mismatch Diagnosis:

- for scenarios 1 and 2, the error should provide enough information to identify the location and the nature of mismatch;

- for scenario 3, a full mismatch diagnosis is necessary.

e) Mismatch Repair:

- for all the scenarios, system reconfiguration involves the usage of alternative component or connector; for example: an alternative component that allows backtracking, or a buffered connector to store data until there is no more risk of backtracking.

From the above exercise, we can draw several conclusions. If the appropriate information for dealing with potential mismatches (for example, documentation of the architectural features and availability of redundant architectural elements) is embedded into the system, right from its architectural conception, then the actual process of tolerating mismatches becomes much simpler. This includes both developing measures for mismatch tolerance and tolerating mismatches at runtime. The reasons for this are that on the one hand some of its activities, like diagnosis, that are complex, time consuming and prone to errors, cease to be necessary, but on the other hand important architectural information related to mismatches and their tolerance is lost during the following phases of the life cycle. Another conclusion is that, although the dichotomy for identifying the nature of mismatches (as style- or application-specific) is clear, the same cannot be said about the process of repairing them, since its techniques might require handling aspects that are particular to style and application. The assumption that architectural features should be part of the interfaces of the components can be weakened by employing other means for retrieving all information related to such properties, for example, using a reflective capability or a specialized registry for storing it.

7 Conclusions

The problem of tolerating architectural mismatches during runtime can be summarized as follows. When an error caused by mismatch (ECM) is detected in the system, mechanisms and techniques have to recover the state of the system to an error free state, otherwise the erroneous state of the system can propagate, eventually leading to a system failure. However, the detection and recovery of an error is not enough for maintaining the integrity of the system services because if the mismatch, which has caused the detected error, is not treated, it can yet again be activated and be the cause of other errors. Similarly to fault tolerance in which one cannot develop techniques that can tolerate any possible faults, it is difficult to develop techniques that are able to deal with all types of architectural mismatches, hence assumptions have to be made about the types of mismatches that caused the errors to be detected and handled during runtime.

In this paper, we have mainly stated the problems and outlined a general approach to handling architectural mismatches during runtime. Our preliminary analysis shows that a number of particular mismatch tolerance techniques can be developed depending on the application, architectural styles used, types of mismatches, redundancies available, etc. It is clear for us that there will always be situations when mismatches should be avoided or removed rather than tolerated. Beyond the working examples discussed in the paper, the applicability of the proposed approach to real systems still remains an open issue. However, since the paper advocates application of general fault tolerant mechanisms and techniques for handling architectural mismatches, the potential limitations of our approach are the same as those associated with traditional fault tolerance when applied to the systems of the same scale and complexity. This, in particular, concerns scalability of the techniques and the ways the systems are structured.

In our future work we will be addressing these issues, trying to define in a more rigorous way the applicability of the approach and to develop a set of general mismatch tolerance techniques. Some of the possible approaches are to modify how existing architectural styles are applied, to design a set of connectors capable of tolerating typical mismatches, to extend existing components and connectors with an ability to execute exception handling, and to develop a number of handlers that are specific for mismatches of different types.

Acknowledgements. Alexander Romanovsky is supported by IST DSoS project (IST-1999-11585).

References

1. T. Anderson, and P. Lee. *Fault-Tolerance: Principles and Practice*. Prentice-Hall Int. Englewood Cliffs, NJ. 1981.
2. A. Avizienis, J.-C. Laprie, and B. Randell. *Fundamental Concepts of Dependability*. Technical Report 739. Department of Computing Science. University of Newcastle upon Tyne. 2001.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley. 1998.
4. D. Compare, P. Inverardi, and A. L. Wolf. "Uncovering Architectural Mismatch in Component Behavior". *Science of Computer Programming (33)2*. 1999. pp. 101-131.
5. F. Cristian. "Exception Handling". *Dependability of Resilient Computers*. T. Anderson (Ed.). Blackwell Scientific Publications. 1989. pp. 68-97.
6. R. de Lemos, and A. Romanovsky. "Exception Handling in the Software Lifecycle". *International Journal of Computer Systems Science & Engineering 16(2)*. March 2001. pp. 167-181.
7. R. DeLine. "A Catalog of Techniques for Resolving Packaging Mismatch". *Proceedings of the 5th Symposium on Software Reusability (SSR'99)*. Los Angeles, CA. May 1999. pp. 44-53.
8. A. Egyed, N. Medvidovic, and C. Gacek. "Component-Based Perspective on Software Mismatch Detection and Resolution". *IEE Proceedings on Software 147(6)*. December 2000. pp. 225-236.
9. C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm, "On the Definition of Software Architecture". *Proceedings of the First International Workshop on Architectures for Software Systems – In Cooperation with the 17th International Conference on Software Engineering*. D. Garlan (Ed.). Seattle, WA, USA. April 1995. pp. 85-95.
10. C. Gacek. *Detecting Architectural Mismatches during System Composition*. PhD Dissertation. Center for Software Engineering. University of Southern California. Los Angeles, CA, USA. 1998.

11. D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch: Why Reuse is so Hard”. *IEEE Software 12(6)*. November 1995. pp. 17-26.
12. P. Inverardi, A.L. Wolf, and D. Yankelevich. “Checking Assumptions in Component Dynamics at the Architectural Level”. *Proceedings of the 2nd International Conference on Coordination Models and Languages*. Lecture Notes in Computer Science 1282. Springer, Berlin. September 1997. pp. 46-63.
13. C. Jones, A. Romanovsky, I. Welch. A Structured Approach to Handling On-Line Interface Upgrades. *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002)*. Oxford, UK. August 2002. IEEE CS Press. pp. 1000-1005.
14. J.-C. Laprie. “Dependable Computing: Concepts, Limits, Challenges”. *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*. IEEE Computer Society Press. Pasadena, CA. June 1995. pp. 42-54
15. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. “A Language and Environment for Architecture-Based Software Development and Evolution”. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA. May 1999. pp. 44-53.
16. N. Medvidovic, and R. N. Taylor. “A Classification and Comparison Framework for Software Architecture Description Languages”. *IEEE Transactions on Software Engineering 26(1)*. 2000. pp. 70-93.
17. P. Oberndorf, K. Wallnau, and A. M. Zaremski. “Product Lines: Reusing Architectural Assets within an Organization”. *Software Architecture in Practice*. Eds. L. Bass, P. Clements, R. Kazman. Addison-Wesley. 1998. pp. 331-344.
18. D. E. Perry, and A. L. Wolf. “Foundations for the Study of Software Architecture”. *SIGSOFT Software Engineering Notes 17(4)*. 1992. pp. 40-52.
19. D. S. Roseblum, and R. Natarajan. “Supporting Architectural Concerns in Component Interoperability Standards”. *IEE Proceedings on Software 147(6)*. December 2000. pp. 215-223.
20. M. Shaw, and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall. 1996.
21. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow “A Component- and Message-Based Architectural Style for GUI Software”. *IEEE Transactions on Software Engineering 22(6)*. June 1996. pp. 390-406.