Tolerating Architectural Mismatches

Rogério de Lemos
University of Kent at Canterbury
UK
+44-1227-823628
r.delemos@ukc.ac.uk

Cristina Gacek
University of Newcastle upon Tyne
UK
+44-191-222-5153
cristina.gacek@ncl.ac.uk

Alexander Romanovsky
University of Newcastle upon Tyne
UK
+44-191-222-8135
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, in particular, on the ability of these mechanisms to cope with architectural mismatches that might exist between components. This paper is based on the assumption that architectural mismatches always exist in such systems, so the need to handle them in run-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 existing components, which are independent in their nature, but are able to interact in well-understood ways. To find a solution to this problem we apply general principles of fault tolerance in the context of dealing with architectural mismatches

1. INTRODUCTION

There are several concepts that are relevant to addressing the tolerance of architectural mismatches in software systems. In this section we introduce the concepts of software architectures, architectural mismatches, and dependability which are pivotal for understanding the need for tolerating architectural mismatches in software systems, as well as the approach we are taking to tackle this problem.

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 [1]. A software architecture is usually described in terms of its components, connectors and their configuration. The way a software architecture is configured defines how various connectors are used to mediate the interactions among components.

As a result of combining several architectural elements using a specific configuration, architectural mismatches may occur [6]. 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 [8]. These assumptions can be related to the nature of components and connectors (control and data models, and synchronisation protocols), the global system structure, or the process of building the system [6, 9]. Traditionally, mismatches have been dealt with statically [5, 3], by means of analysis and removal.

Dependability is a vital property of any system justifying the reliance that can be placed on the service it delivers [7]. A system *failure* occurs when a system service deviates from the behaviour expected by the user. An *error* is the part of the system state that is liable to lead to the subsequent failure. The adjudged or hypothesized cause of an error is a *fault*. Fault tolerance is a method of achieving dependability working under assumptions that a system contains faults (e.g. ones made by humans while developing or using systems, or 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. Error processing typically consists of three steps: error detection, error diagnosis and error recovery. 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, operators' mistakes, and failures in the environment in which the system operates.

There are many reasons to support our claim that it is impossible to detect and correct all possible architectural mismatches statically, 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 while building, they have to meet many functional and non-functional requirements which 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, current software systems may undergo dynamic reconfiguration, adding uncertainty about the various architectural elements present at any point in time. In this paper we show that architectural mismatches can be tolerated.

2. ARCHITECTURAL MISMATCHES

All Architectural mismatches occur because of inconsistencies among the given architectural elements. These inconsistencies can be stated in terms of the features (or characteristics) exhibited by the parts at hand. 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 [9], yet they do not prescribe all the features that may be present in an application [5]. During software development, the software architecture is incrementally refined following the refinement of the system's 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 [4]). Every time an architectural feature is bound there exists the 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) the style(s) imposes; or as *application-specific* if their presence is due to architectural decisions imposed by the application at hand (not the particular style(s) used).

We believe that 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 [7]). 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.

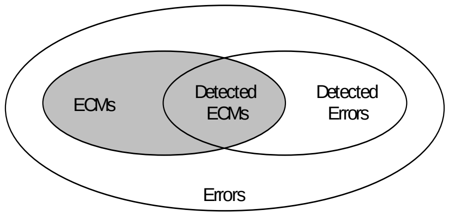

Figure 1. Detected errors caused by mismatches

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 *avoided* by imposing strict rules on how components should be built and integrated, which leads to bespoke products. Mismatches can be *removed* when integrating arbitrary components by using static analysis methods and techniques [5, 3]. 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.

3. MISMATCH TOLERANCE

The tolerance of architectural mismatches during runtime relies on *ECM processing*, which comprises three steps [7]. These are:

- *The detection of ECMs*, which identifies erroneous states that were caused by mismatches.
- *The diagnosis of ECMs*, which assesses the system damages caused by the detected ECMs.
- *The recovery from ECMs*, which brings the system to an error-free state.

However, error processing is not sufficient if we want to avoid the recurrence of the same architectural mismatch, also there is the need to treat mismatches, in the same way that faults are treated [7]. *Mismatch treatment* involves diagnosis, which determines the cause (localization and nature) of the ECM, isolation that prevents a new activation of the mismatch, and reconfiguration, which modifies the structure of the system for the mismatch free components 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 to avoid the propagation of ECMs, to facilitate ECMs detection and recovery, and to make difficult the reactivation of architectural mismatches.

The particular problem associated with mismatch tolerance is that we are dealing with two levels of abstraction: the architectural level where the mismatches are introduced, and the execution level where the ECMs detection and recovery takes place. Hence, the needs for identifying what are the potential consequences upon the state of the system when an architectural mismatch is activated. This additional information is fundamental for distinguishing ECMs from other system errors, thus providing the basis for defining an architectural solution for tolerating mismatches.

A motivation for specifying mechanisms for handling architectural mismatches at the architectural level, instead, for example, during implementation, 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 once again 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 lifecycle [2]. Moreover, we cannot expect that a general runtime mechanism would be able to handle a wide range of architectural mismatches, in the same way that there is no sufficiently general fault tolerant mechanism that can handle all classes of faults. It is envisaged that different classes of mismatches will require different types of detection mechanisms and fixes that have to be specified at the architectural level.

3.1 ECMs' Processing

As previously discussed, the detection of an ECM implies the presence of a mismatch. For a mismatch to be activated some preconditions must be satisfied, which can be defined in terms of systems states and features' inconsistencies [5].

Upon error detection, one must first determine whether that particular instance is an ECM or not. For an error to be detected as an ECM we need additional information at run time, about system's states and features of the relevant architectural elements, that would enable to associate that error with a mismatch. This ought to be done based on the particular error observed and on the presence of the preconditions required to activate it. The level of difficulty encountered on recovering from ECMs will very much depend on the individual error's characteristics and the architecture at hand. The treatment of faults that do cause ECMs will depend on whether the relevant features are style or application specific. It is our current belief that ECMs caused by style-specific features would require more fundamental changes to the system at hand, but this conjecture requires further study to be properly supported or contradicted.

3.2 Examples

In the rest of this section we briefly outline several simple examples demonstrating how our approach can be used.

First of all, let us consider a simple application-specific mismatch. Mismatch "sharing or transferring data with differing underlying representations" (mismatch 42 in [5] occurs when, for example, a component provides a value in feet and another component requires it in meters. In this case, "data transfer" is the conceptual feature used for ECM detection and recovery at the level of the application. The meta information that is required for detection and recovery concerns types of data to be transferred. Fault treatment may consist of replacing a connector with a new one that transforms the data.

The second example is that of application and style-specific mismatches. It happens, for example, when several components are connected in a system but only some of them can be backtracked (mismatch 28 from [5]). If any of such interconnected components backtracks, it has to inform all the components with which it interacts. To detect such a mismatch during run-time it is necessary to have additional information on the ability of each component to backtrack (conceptual feature "backtracking"), on the fact that backtracking is initiated by a component and on a set of interconnected components to be involved in backtracking. The detection of the ECM can be at the style level, but the recovery should be at the application level because, generally speaking, the application decides how to proceed with inconsistent data.

A style-specific mismatch happens, for example, when a non-reentrant component is called without waiting for the previous call to be completed (mismatch 24 from [5]). In this case we have a style-specific mismatch that can be detected and recovered if additional mechanisms are incorporated into the basic style (which can be based on the conceptual feature "re-entrance"). For example, an instantiation of a style should be able to detect any attempts to re-enter a process being executed. In the context of the pipe and filter architecture, it can happen that two filters try to access a single process in a third filter (see Figure 2). An extended style should provide means for detecting the ECM and for local recovery by either ignoring the second request or introducing additional concurrency control into the style (the simplest of which would be just delaying the second request).

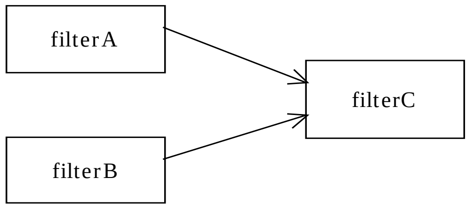

Figure 2. A non-reentrant component in a pipe-and-filter architecture

4. CONCLUSIONS

The problem of tolerating architectural mismatches during runtime can be summarised 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 run time. 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. 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.

ACKNOWLEDGMENTS

Alexander Romanovsky is supported by European IST DSoS project, and Cristina Gacek by the EPSRC (UK) DIRC project.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*: Addison-Wesley. 1998.

[2] R. de Lemos, A. Romanovsky. “Exception Handling in the Software Lifecycle”. *International Journal of Computer Systems Science & Engineering 16(2)*. March 2001. pp. 167-181.

[3] A. Egyed, N. Medvidovic, C. Gacek. “Component-Based Perspective on Software Mismatch Detection and Resolution”. *IEE Proceedings on Software 147(6)*. December 2000. pp. 225-236.

[4] 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, 24-25 April 1995. pp. 85-95.

[5] C. Gacek. *Detecting Architectural Mismatches during System Composition*. PhD Dissertation. Center for Software Engineering. University of Southern California. Los Angeles, CA, USA. 1998.

[6] D. Garlan, R. Allen, J. Ockerbloom, “Architectural Mismatch: Why Reuse is so Hard”. *IEEE Software 12(6)*. November 1995. pp. 17-26.

[7] 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

[8] P. Oberndorf, K. Wallnau, A. M. Zaremski. “Product Lines: Reusing Architectural Assets within an Organisation. *Software Architecture in Practice*. Eds. L. Bass, P. Clements, R. Kazman. Addison-Wesley. 1998. pp. 331-344.

[9] M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall. 1996.

	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