Software Engineering: Integration Requirements

AYAZ ISAZADEH
Department of Computer Science
Tabriz University
Tabriz, IRAN

Abstract: This paper presents a discussion of software integration requirements, with an emphasis on the requirements of post-factum software systems integration. The problem is defined; the requirements and associated issues are discussed. Formality as a major requirement of software integration is emphasized and a formal approach, called Compositional integration, is introduced. The requirements for integration of redundant software components, developed using diverse software engineering methodologies, into a fault tolerant system is described. Finally, the paper concludes with a brief discussion of dynamic integration requirement and some final remarks.

Key-Words: Software Integration, Requirements Specification, Software Architecture.

1 Introduction

The major motivation leading to this work comes from some unpleasant industrial experiences in software maintenance, enhancement, and integration. There have been cases where side effects of simple modifications or small enhancements have caused failures of large-scale properly functioning software systems in the field. A software system, functioning as specified in the field for years, may not behave according to the specifications when integrated with another system. Earlier features of communication systems, for example, were specified and developed based on assumptions which may no longer be true. Later features, then, added to an environment with changing assumptions, result in the problem of Feature Interactions, discussed by Pamela Zave [1].

Software technology is growing and changing so rapidly that almost any software system must be modified or reconfigured to provide enhanced integrated solutions to the changing world. In many cases, the requirements specification of an existing software system, is either no longer available or does not correspond to the actual system. Modification and maintenance of such a system, as well as its integration with a new system, is a serious problem.

The focus of this paper is on the requirements of post-factum integration of software components. Post-factum integration, also called “post-facto integration” by some researchers [2], refers to techniques for combining existing software components to form complete systems. In post-factum integration, a software component can be a library subroutine, a program developed specifically for the purpose of integration, or a complete and possibly complex system. Post-factum integration, however, includes at least one existing software system, developed in the past with no plan for its systematic integration with any other component.

2 Definitions

Software integration problem includes one or more of the following variations:

1. Systems Integration: Given two or more software systems, subsystems, or components,
each of which functioning properly (i.e., satisfying their requirements within their environments), the problem is to integrate them into one larger system, satisfying the combined requirements within the newly formed environment.

2. New Function or New Technology Integration: Given a software system, which may have been functioning properly in the field for a significant period, the problem is to integrate a new function or a new technology within the system. The integrated system should provide the new functionality or use the new technology, while preserving the original system’s functionality.

3. Incremental Engineering: A software system can be developed and delivered using available technologies and with less functionality than it is intended to eventually provide. New technologies and/or more functions, then, can be integrated within the system. The problem is to design the system with such future integration in mind.

4. Modification: Sometimes an existing and properly functioning software system must be decomposed and reintegrated to carry out a modification. The reintegration may include a new component, providing a new functionality and/or employing a new technology, and/or it may exclude an old component.

5. Building a System from Prefabricated Parts (Reuse) [3]: Software engineering using reusable software components is another form of the system integration problem. System designers, here, are constrained by the software components and are not free, for example, to define the system breakdown in a top-down manner. This can be better described as “component integration engineering” or Component-Based Software Development (CBSD). In this approach, software systems are developed by selecting appropriate Commercial Off-The-Shelf (COTS) components and assembling them with a well-defined software architecture [4, 5]. CBSD can reduce development costs, increase reliability, and improve quality of software systems [6].

3 Formality

Formality in software integration process is a major requirement, for ambiguity and inconsistency in the requirements specification of a product can do a great deal of damage to the production process, making the final product unlikely to satisfy its requirements. And, in the words of David Parnas [7], unless you have a complete and precise description of your product’s requirements, it is very unlikely that you will satisfy those requirements. Precision in the requirements specification, in turn, demands using formal methods. Despite the controversial issues concerning the practicality of formal methods, formality and mathematics are, generally, becoming more and more useful in all phases of software engineering. There are arguments on how mathematics can provide a scientific foundation for the modeling aspects, description techniques, and development methods of software engineering, leading to a deeper understanding of the development process [8].

A natural approach to the integration of some existing software components (post-factum integration) is the way in which the components can be integrated without using any knowledge of their internal structures. This leads to the notion of compositionality. In a compositional development method, which has been around for quite a while [9], a design is verified in terms of specification of its sub-programs. Our problem is compositional integration of existing software components.

In the compositional integration, similar to the compositional development [10], the specification of a system should be verified on the basis of the specifications of its constituent components without knowledge of the internal structure of these components. Formally, assume that a system $S$ is composed of an integration of the components $S_1, S_2, ..., S_n$; that is, $S$ is of the form $C(S_1, S_2, ..., S_n)$, where $C$ is some syntactic constructor of finite and fixed arity (i.e., combines a fixed number of components). Then a specification $spec$ for $S$ should be proven from specifications $spec_1, spec_2, ..., spec_n$ without referring to the in-
4 Fault Tolerance

Failure, and a consequent temporary unavailability of service, is not an unusual phenomenon for many software systems. Users of such systems have actually accepted this fact and learned to live with it. However, an integrated system may require a high level of fault tolerance; there is in fact a growing number of systems, such as communication on line transaction processing, and control systems, for which failure and service unavailability is not acceptable. Furthermore, a software system, functioning as specified in the field for years, may not behave according to the specifications when integrated with another system (because of the changes in environment and assumptions). Consequently, systems integration can be viewed as the integration of potentially faulty components. Fault tolerance, therefore, is a major requirement in software systems integration, particularly, in integration of real time (and distributed) systems.

In an integrated system, a component failure occurs when the component does not behave in a manner consistent with its specification. Component failures can be classified as follows [11]:

1. **Omission Failure**: Occurs when a component fails to respond to an input. Example: a communication link, which may occasionally lose messages.

2. **Timing Failure**: Occurs when a component’s response is functionally correct but untimely. It could be early timing failure or late timing failure. Late timing failure is also called performance failure. An excessive delay in message transmission is an example of communication performance failure.

3. **Response Failure**: Occurs when a component responds incorrectly. It could be value failure, if the value of its output is incorrect, or state transition failure, if the state transition that takes place is incorrect. A communication link, which may deliver corrupted messages, for example, suffers value failure.

4. **Crash Failure**: Occurs when a component, after a first omission failure, fails to respond to the subsequent inputs until its restart.

A fault tolerant system requires recovery action upon detection of a component failure. For each component, therefore, the likely failure behaviors must be identified. The term most widely used to describe failure behaviors is failure semantics. If the likely failure behaviors of a software component are in class \( F \), then the component is said to have \( F \) failure semantics. For example, a communication link, which may occasionally lose messages—with no message corruption or transmission delay—has omission failure semantics. A component may have \( F/G \) failure semantics if its likely failure behaviors are in \( F \cup G \). \( F/G \) is said to be a weaker failure semantics than \( F \). When any failure behavior is allowed for a component, then it is said to have arbitrary failure semantics, which is the weakest of all failure semantics and includes all the failure classes defined above.

4.1 Component Failure Masking

If a component \( P \) of an integrated system \( S \) fails and \( S \) continues functioning according to its specifications despite the failure of \( P \), then \( S \) is said to mask \( P \)’s failure.

Component failure masking requires an integration architecture, capable of masking component failures. In an integrated system, if a component \( P \) has arbitrary failure semantics then any other component \( Q \) which depends on the output provided by \( P \) may have arbitrary failure semantics, unless \( Q \) has some way of checking the correctness of output provided by \( P \). Designers of fault tolerant systems, therefore, prefer to use components with stronger failure semantics such as omission or performance. However, the stronger the failure semantics, the more complex and expensive is the component that implements it.

An architecture, therefore, should provide a way in which the component behaviors and outputs are checked for correctness, recovery and corrective actions are taken as necessary and, consequently, the system’s services may be provided without interruption during a component failure. One such architec-
tural style is called *group failure masking*.

A group of redundant software components, in an integrated software system, *masks* the failure of a group member, if the group functions as specified despite the member failure. The group *output* is a function of the outputs generated by the group members; it can be, for example, the output of a specific member, or the result of a majority vote on member outputs. Normally, each group is managed by a *group management mechanism*. The stronger the failure semantics of the group members the simpler is the group management mechanism. On the other hand, as mentioned above, the stronger the failure semantics, the more complex and expensive is the component that implements it. So, a balance should be achieved here.

Group failure masking can be accomplished using diverse programming techniques like *Recovery Block* (RB) [12], *N-Version Software* (NVS) [13], or a combination of both [14].

A *recovery block* in an integrated software system consists of two or more *alternate* software components and an *acceptance test* component. The alternate components in a recovery block are different versions of the same software. One version, the *primary alternate*, is executed and the result is checked using the acceptance test. If the output of the primary alternate fails the acceptance test, then the next version is executed, hoping that the output of this version will not fail the test. Otherwise, the third version is executed and so on. The integrated software, therefore, can mask a component failure by using another component in the same recovery block.

An *N-Version Software* (NVS) system consists of two or more functionally equivalent, yet independently developed software components called *member versions*. The NVS versions are executed concurrently under a supervisory system called *N-Version Executive* (NVX), which uses a decision algorithm based on consensus to determine final outputs. An integrated system, therefore, can achieve fault tolerance by using NVS systems (i.e., by masking a component failure using other member versions). NVS systems reliability, in turn, depend deeply on the diversity in design and development of member versions.

Diverse software design methodology, ideally, pursues the goal of ensuring that the group members, running diverse programs, do not fail at the same time despite the possibility of design faults in these programs. However, diverse programming methodology is still a controversial issue [11]. The increase in reliability that results from the use of diverse programming techniques has not yet been convincing for some researchers [15].

### 5 Dynamic integration

The term *dynamic integration* (or reconfiguration) refers to changes in a system’s configuration while it is running. Dynamic integration includes adding, removing, or replacing a component, changing the connections between components, and moving a component from one machine to another one. Dynamic integration is either ad-hoc performed interactively, or pre-planned, triggered by the system and performed automatically.

For many large software systems, it is simply not acceptable to shut down the entire system for a modification or reconfiguration. These systems must be modified, reconfigured, and extended while they are running without imposing significant performance degradation. It is, therefore, an important requirement for the integration process to be capable of *dynamic reconfiguration*.

Applications of dynamic reconfiguration include adding a new functionality or changing/removing an existing one, employing a new technology, implementing fault tolerance and service availability methodologies, and using load sharing/balancing techniques.

### 6 Conclusion

This paper has presented a discussion of software systems integration requirements with an emphasis on post-factum integration and fault tolerance. In order to integrate some software components into a larger system, one must have a precise understanding of the components. A precise understanding of a software component can only be provided by a formal specification of the software [16]. Formal methods, therefore, can play an important role in software systems integration. The paper has emphasized the importance of having formal bases for
software integration requirements specification and introduced a formal approach, the Compositional integration, for a sound integration process.

The paper has also presented the requirements for integration of redundant software components, developed using diverse software engineering methodologies, into a system with a high level of fault tolerance.

References:


