A Review of Post-Factum Software Integration Methods

Ayaz Isazadeh Glenn H. MacEwen Andrew Malton

October 31, 1995
External Technical Report
ISSN-0836-0227-
95-389

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared October 31, 1995

Abstract

This paper presents a review and discussion of post-factum software systems integration. The problem is defined; the approaches and associated issues are discussed. Integration of redundant software components, developed using diverse software engineering methodologies, into a fault tolerant system is reviewed. Finally, a novel approach to fault tolerant software integration, based on the *critical* properties of software systems and the way in which these properties can be maintained by the integrated system despite a component failure, is presented for further research.

Contents

List of Figures

1 Introduction

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 post-factum integration of software components. Post-factum[1] integration 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 Basic Concepts

Post-factum 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 functions properly (i.e., satisfies 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 given 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.

[1]Some researchers have also used the term "post-facto" for the same concept [12].

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)* [8]: 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".

2.1 Program Understanding

The problem of software systems integration, in any of the above variations, involves integration of some software components. A software component can be as small and simple as to provide a simple feature or as large and complex as a switching system in communication software. A software component can be a new system, possibly developed for the purpose of integration, or an old one, which has been in the field functioning properly for years. In any case, a good understanding of the software components is the first step one must take towards systems integration. There are several approaches to achieve this understanding. Unfortunately, they are all informal; the most systematic ones are *reverse engineering* techniques.

2.1.1 Reverse Engineering Approach

Hausi Müller and others [10] define *reverse engineering* as the process of extracting system abstractions and design information out of existing software systems. Applied to a software component, this process involves identification of software artifacts in the component, exploration of how these artifacts interact with one another, and their aggregation to form more abstract representations that facilitate program understanding. Software artifacts include components such as procedures, modules, interfaces, and dependencies among them.

An example of reverse engineering systems is Rigi [9, 17]. Developed at the University of Victoria for discovering and analyzing the structure of large software systems, Rigi provides the following features:

- *Parsing the source code and extracting the artifacts*: This is a language dependent feature, which currently supports COBOL and C and C++.

- *Storing the artifacts in a database*: Graphical representations of the artifacts, extracted from the source code, are stored in a database called GRAS, which is designed to represent graph structures.
- *Editing the graphical representations of the artifacts*: This is a language independent feature, which permits graphical manipulation of the source code representations.

Another reverse engineering technique, presented recently by Panagiotis Linos and others is CARE (Computer Aided Reengineering) [6], which maintains a repository of control-flow and data-flow dependencies of C programs. Different graphical representations of these dependencies can facilitate program understanding.

2.1.2 Other Approaches

Other approaches to program understanding constitute guidelines and instructions that a system integrator should follow.

Blum and Moore [2], for example, describe their research on integration of some Navy tactical computer systems. Their motivation for this research is the capture of what is known about a software system. To accomplish this they suggest three principal goals:

1. *Collect and organize what is known about a software component.*
2. *Structure what is known about the implementation of the software component.*
3. *Develop techniques to retrieve what is known about the software component.*

The problem with this approach and other informal approaches to program understanding is the uncertainty of these approaches; there is always the possibility of overlooking certain information about a software component, certain assumption about the environment, or something else.

2.2 Software as a Source of Knowledge

A good source of information and, in some cases, the only source of information about an existing software system is the system itself. However, a software system is a response to a need. The need is defined as a problem and a software system is engineered and provided as a solution to the problem. The information collected from a solution does not necessarily reflect the problem. In other words, if one begins from an existing implementation, it is very difficult to distinguish between the result of a design decision and the inherent constraints of the problem. This difficulty can be illustrated by examining the process of arriving at a software solution.

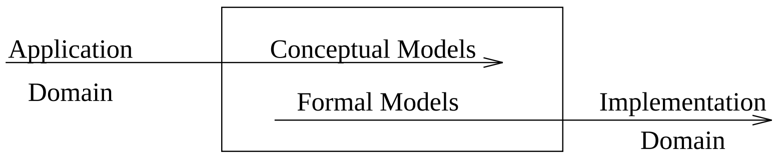

Figure 1: The essential software process (From [2])

2.2.1 Process of Arriving at a Software Solution

Blum and Moore [2] describe the essence of the software process as a transformation from some need in an application domain into a software implementation that responds to that need (Figure 1).

Basically, the process of arriving at a software solution has the following stages:

1. *Identification of a Need in an Application Domain:* In this stage, a need for an application is identified.

2. *Conceptual Models:* Conceptual models describe the proposed response to the need.

3. *Formal Models:* Formal models establish the essential behaviors of the desired software solution.

4. *Implementation:* In this stage, an implementation of the formal model, a software system, is developed as a final solution to the problem.

Notice that there could be many different conceptual models satisfying the need, many different formal models satisfying the conceptual model, and many different implementations for the given formal model. At each stage, therefore, there are many alternative solutions, from which only one is chosen. Thus, the solution space, in each stage, becomes smaller and smaller until only one solution exists. In the process of software engineering, normally, alternative solutions are also discussed and investigated. However, once a software system is developed and delivered, knowledge related to the rejected alternatives is often lost. Furthermore, the software contains additional constraints which have been added to the system, in different stages, based on the design decisions.

3 Integration Architectures

A software integration architecture, as defined by Rossak and Ng [15], is a general pattern that serves as a blueprint to define the basic layout of an integrated system. It has to deal with bottom-up integration of existing components (*post-factum integration* [12]) as well as the development of new components for top-down integration (*pre-facto integration* [15]). An integration architecture describes the general strategy of system decomposition, data storage, and communications. The architecture should not be concerned with the internal structure of the components. A software component within the integration architecture, therefore, should be monolithic, i.e., allow no access to its internal structure. There are two major issues concerning the technical aspects of integration architectures:

- communication in the system, and
- handling data within the system.

These issues and the way in which a large-scale integration architecture may resolve these issues lead to the following basic types of software integration architectures:

- *Message-passing systems*
- *Channel-based systems*
- *Systems with central repository*
- *Generic systems*
- *Object-oriented systems*

In a message-passing paradigm, for example, a communication system is used to send messages between components. In architectures with central repository, the database is the main integrating factor. In a channel-based architecture, the (*active*) components use the channel (i.e., *passive* communication components) to communicate with each other.

The remainder of this section reviews some examples of software integration architectures.

3.1 A Generic Framework

Rossak and Ng [16] propose a Generic Systems Integration Framework (GenSIF). They rely on this framework for systems integration, to tailor a preplaned integration process to the needs of an application domain. Their objective is a *software development process*: a way in which software components can be designed and developed to work together as an integrated system. Obviously, this is not within the scope of this paper, which deals with integration of *existing* software components. The framework, however, to the extent it can serve the interest of this paper, is presented here. GenSIF includes three components (Figure 2):

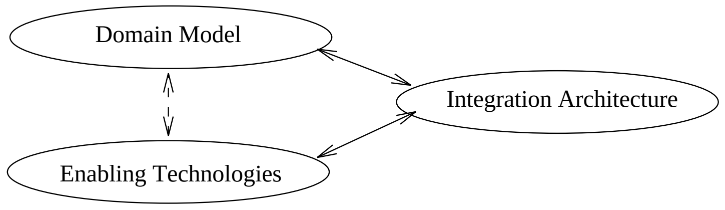

Figure 2: GenSIF framework (From [16])

1. *Global domain integration*: Domain integration involves an analysis of the application domain to define a common model of the environment that the system will serve. This is also called *semantic integration*. Domain analysis provides the conceptual basis for the integration architecture.

2. *Derivation of an integration architecture*: The integration architecture is a conceptual/structural model. It is also an infrastructure, which provides the necessary utilities and components to implement an application system by following the rules of the conceptual model.

3. *Assessment and usage of enabling technologies*: Provides the tools and products that are required by the infrastructure of the integration architecture.

A software integration architectures consists of two parts: the conceptual/structural model and the technical infrastructure. The conceptual architecture model describes the guidelines and standards of the architecture, linking the model of the application domain to the implementation-oriented concepts of development environment and enabling technologies. This model specifies a general strategy of system decomposition, inter-process and user-communications, internal and external interfaces, etc. A good example of this model is OSCA [7] which is a channel-based integration architecture on the conceptual/structural level. OSCA relies on free and implementation-independent communication in a distributed environment. It gives explicit guidelines for decomposition in *building blocks* (See Section 3.3). The technical infrastructure of an integration architecture deals with the development platforms such as RAPID of AT&T and goes beyond the scope of this paper.

This framework can be adapted to include integration of existing software components. The third component of the framework (the *enabling technologies*), then, would include the existing software components. The infrastructure of the integration architecture would use these software components in the integration process. The infrastructure may add a new interface to a given software component if required by the conceptual model of the integration architecture.

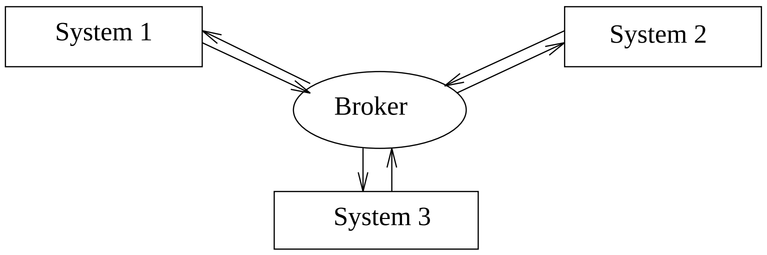

Figure 3: The Broker approach (From [3])

3.2 The Broker Architecture

Clark and Krumm [3] describe the *Broker*, their approach to integration of information across three Air Force legacy systems. Each one of these systems contains a large amount of information that needs to be shared by the other systems. The Broker enables autonomous and heterogeneous systems to share information while maintaining autonomous control over their data (Figure 3). The following items are the key components of the Broker concept:

- Each system interfaces with the Broker and does not exchange any information directly with other systems.
- Each system can interface with the Broker in whatever format the system is designed to support. This is particularly important for legacy systems, which have different interfaces.
- Data received by the Broker are examined, translated into neutral format, and forwarded to the appropriate system(s).
- The Broker does some error checking and may request retransmissions.
- Each legacy system maintains control over its own database. Through data translation and forwarding, the Broker helps ensure data consistency between the systems.
- The Broker uses an internal database for message translation, message storage, and error logging.

As shown in Figure 4, the Broker is composed of a number of modules. *Input* and *output modules* are concerned with receipt and forwarding of messages using various data communication standards. There are multiple instances of these modules because of the multiple data and communication standards employed by the legacy systems. The *event handler* takes care of the events like passing a message to the *translation module*, storing, re-transmitting, etc.

The Broker architecture provides an approach to integration of information across different systems. If the problem is information integration then the

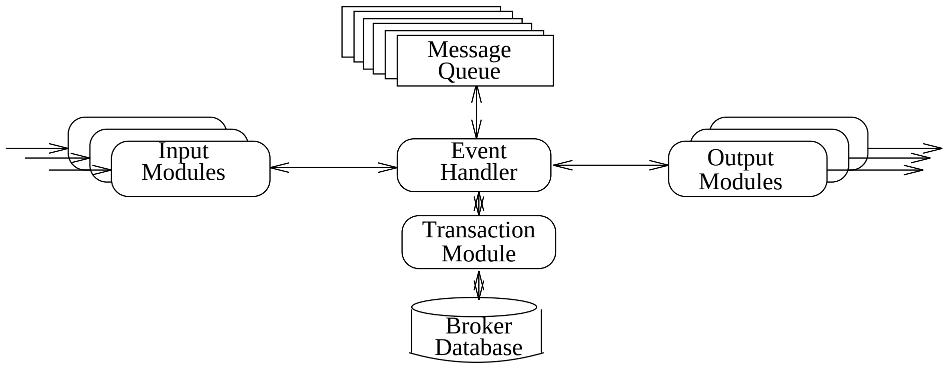

Figure 4: The Broker internals (From [3])

Broker provides a solution. The Broker, however, is not a practical approach to integration of some software components into a larger system, where the components require tighter interaction.

3.3 Interoperability Architectures

Interoperability, as defined by John Mills of Bellcore [7], is the ability to interconnect “business aware software products” to provide access to corporate data and functionality by any authorized user. *Business aware software* is software which provides functions characteristic of a business. The interest of this paper, however, is in the concept of *interoperability* as it may apply to software integration. Interoperability conflicts, in this context, refer to problems that an integrated software system may have as a side effect of the integration, problems that the software components, as independent systems, do not have.

An example of interoperability architecture is Bellcore’s OSCA [7]. OSCA requires that the business aware functions be separated and grouped into three layers: *corporate data layer, processing layer,* and *user layer*. The software that implements the functions in these layers is partitioned into *building blocks*. A building block is a deployable unit of software product. The interfaces between building blocks, termed *contracts*, must meet certain criteria, such as the use of industry and international standards and isolation from building block internals. The building blocks must adhere to specific interoperability principles, some of which are as follows:

- *Release independence*: Building blocks must be upward and downward compatible; that is, upgrading a building block must not affect other building blocks.

- *Resource independence*: Building blocks cannot share any resource if it would cause a violation of any other interoperability principle.

- *No accessibility assumptions among building blocks*: A building block, when it invokes a contract in another building block, must be prepared to detect an unavailable contract.

- *Location independence*: Each building block contract must be addressed logically; that is, the invoking building block does not need to know where the other building block is physically deployed (*location transparency*).

Fundamental to the OSCA architecture is the notion of *separation of concerns*, which means that business aware functions are allocated to the building blocks, that no building block contains business aware functions from more than one layer, and that the interfaces (contracts) offered by the functions of one building block to the functions of other building blocks are well-defined and well-formed, so that the functions of one layer are de-coupled from the functions of other layers.

Building blocks are sets of program modules, functions, data schemas, etc. Building blocks are the units of application interoperability, where a high enough level of de-coupling can be expected to support interoperability. Program modules are not considered as units of interoperability; they require tighter interaction than expected for building blocks.

Building blocks and the interfaces between them, as mentioned above, must adhere to specific interoperability principles. The OSCA architecture, therefore, cannot be used for integration of independently developed existing software components that do not adhere to the interoperability principles of the OSCA architecture.

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, there is a growing number of systems, such as communications software, on line transaction processing, and control systems, for which failure and service unavailability is not acceptable.

Fault tolerance, therefore, is a major requirement in software systems integration, particularly, in integration of real time (and distributed) systems. This section describes approaches to achieving this goal.

4.1 Component Failures

In an integrated system, a component *failure* occurs when the component does not behave in a manner consistent with its specification. Flaviu Cristian [4] describes his view of fault tolerant distributed systems and classifies component failures as follows:

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.

4.2 Fault Tolerant Architectures

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.

The basic idea behind a fault tolerant software architecture is the capability of the architecture to *mask* 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 architectural style, called *group failure masking*, is defined below.

4.2.1 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) [14], *N-Version Software* (NVS) [1], or a combination of both [13].

Based on Brian Randell's work [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.

Introduced by Algirdas Avižienis [1], 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 [4]. The increase in reliability that results from the use of diverse programming techniques has not yet been convincing for some researchers [5].

5 A Novel Approach

A novel approach to software systems integration is presented in this section for further research and experimentation.

The major motivation leading to this approach 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 popular problem of *Feature Interactions*, discussed by Pamela Zave [18].

The principal goal of this approach, therefore, is to ensure that certain properties of a system, called the *critical properties*, are not affected by any failure, caused by changes in the assumptions and environments or by a fault in a component. Ideally, in a fault tolerant system, no property of the system should be affected by the failures. Practically, however, it is easier to deal with a subset of the system properties, the critical properties. Notice that the term *critical properties*, here, refers to a subset of the system properties, which are to be maintained by the system despite the likely failures. The remainder of this section outlines this approach in a 3-step process.

5.1 Step 1: Identify Critical Properties

The system integrator should identify critical properties of the software components. Specific properties that are considered critical depend on the software components and the application, for which the system is being integrated. Generally, however, critical properties can be classified, with respect to the component failure classification discussed in Section 4.1, as follows:

1. *Response Critical:* Properties, the lack of which results in *omission failure* of the corresponding component.

2. *Timing Critical:* Properties, the lack of which results in *timing failure* of the corresponding component. A subset of these properties is called *performance critical.*

3. *Value Critical:* Properties, the lack of which results in *value failure* of the corresponding component.

4. *Progress Critical:* Properties, the lack of which results in *crash failure* of the corresponding component.

5.2 Step 2: Analyze Components

For each component, analyze the component structure concerning any likely failure affecting the critical properties. The objective, in this step, is to ensure whether the component does have a fault tolerant architecture, maintaining its critical properties, or not. To achieve this objective the following methods are suggested:

1. *Reverse Engineering Method*: Reverse engineering and analysis of the component could provide the system integrator with the insight needed to verify whether or not the critical properties can be affected by any likely fault in the component.

2. *Formal Method*: Given the formal specification of the software component, prove that the component satisfy the critical properties despite a likely fault in the component. If this proof is not possible, for a component, then it must be assumed that the critical properties can be affected by a likely fault in the component. Formal methods cannot be used if the software has no formal specification. Unfortunately, most existing software components are developed without a formal specification; in many cases, however, it may not be too late to produce one. Mark Phillips [11] describes how IBM introduced the Z specifications for a CICS new release after 20 years of its original development. The experience proved that Z specifications are appropriate for both in new components and where large changes involve rewrites.

3. *Testing Method*: Put the component under rigorous testing and verify that the component maintains the critical properties in the integrated environment or demonstrate otherwise.

5.3 Step 3: Design System Architecture

Design a fault tolerant architecture, capable of integrating the components and maintaining the critical properties in the integrated environment. Having completed Step 2, software components are categorized as follows:

1. Components with fault tolerant architectures, maintaining their critical properties in the integrated environment despite a likely fault that these components may have.
2. Other components.

Components in the first category can be included in the integrated system architecture as they are. Components in the second category, however, may fail to maintain their critical properties and cannot be included in the system architecture as they are. The problem, now, is the way in which the integration process should deal with these components. The following methods are suggested:

1. Consider alternative components, providing the same services, repeat Step 2, and try to minimize the number of components in the second category.
2. For each component in the second category, develop a program, simulating the critical properties of the component, and use it as the *secondary alternate* in a *recovery block* with the original component as its *primary alternate* (See Section 4.2.1). The system architecture, now, can include this recovery block instead of the component. This method enables the system to make use of the full services of the component as long as the component is functioning properly. In case of the component failure, the secondary alternate, the simulator, will keep the system running, by providing exceptional responses. The system, of course, may not have the functions provided by the faulty component, when the component fails. The component failure, however, does not make the system fail completely; services provided by the other components may still be available to the system and, consequently, to the users. This is, in fact, a *gracefully degrading* system.
3. If the gracefully degrading system is not satisfactory and the application requires a complete fault tolerance, then redundant components, developed using diverse software engineering methodology should be considered. NVS, RB, (See Section 4.2.1), or combination of these techniques can be used to accomplish the task.

6 Conclusion

This paper has presented a discussion of software systems integration with an emphasis on software fault tolerance. The paper has shown that in order to integrate some software components into a larger system, one must have a precise understanding of the components. Reverse engineering techniques (which currently do not go beyond extracting software structural design information) can facilitate program understanding. However, a precise understanding of a software component can only be provided by a formal specification of the software.

Formal methods, therefore, can play an important role in software systems integration. Unfortunately, no significant work can be found in this area; extracting formal specifications from existing code is still an open problem.

This paper has also presented, for further research, a novel approach to fault tolerant software systems integration, based on the *critical* properties of software systems, and the way in which these properties can be maintained by the integrated system despite a component failure.

Acknowledgments

Many thanks to Professors Terry Shepard and David Lamb for their comments.

References

[1] Algirdas Avižienis. Software fault tolerance. In G. X. Ritter, editor, *Information Processing 89*, volume 11 of *IFIP Congress Series*, pages 491–498, San Francisco, Aug 1989.

[2] Bruce I. Blum and Tamra Moore. Representing navy tactical computer system knowledge for reengineering and integration. In *Proceedings of The Second International Conference on Systems Integration*, pages 530–537, Morristown, New Jersey, Jun 1992.

[3] David Clark and John M. Krumm. Broker: A system integration approach. In *Proceedings of The Second International Conference on Systems Integration*, pages 162–170, Morristown, New Jersey, Jun 1992.

[4] Flaviu Cristian. Understanding fault–tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb 1991.

[5] John C. Knight and Paul E. Ammann. Issues influencing the use of n–version programming. In G. X. Ritter, editor, *Information Processing 89*, volume 11 of *IFIP Congress Series*, pages 217–222, San Francisco, Aug 1989.

[6] Panagiotis Linos, Phillippe Aubet, and Laurent Dumas. CARE: An environment for understanding and reengineering C programs. In *Proceedings of Conference on Software Maintenance*, pages 130–139, Montreal, Sep 1993.

[7] John A. Mills. Large scale interoperability and distributed transaction processing. In *Proceedings of The Second International Conference on Systems Integration*, pages 392–400, Morristown, New Jersey, Jun 1992.

[8] Roland T. Mittermeir and Evelin Kofler. Layered specifications to support reusability and integration. In *Proceedings of The Second International Conference on Systems Integration*, pages 699–708, Morristown, New Jersey, Jun 1992.

[9] Hausi A. Müller and Karl Klashinsy. Rigi: A system for programming–in–the–large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, Apr 1988.

[10] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*. In press.

[11] Mark Phillips. CICS/ESA 3.1 experiences. In J. E. Nicholls, editor, *Z User Workshop*, pages 179–185, Oxford, Dec 1989.

[12] Leigh R. Power. Post–facto integration technology: New discipline for an old practice. In *Proceedings of The First International Conference on Systems Integration*, pages 4–13, Morristown, New Jersey, Apr 1990.

[13] James M. Purtilo and Pankaj Jalote. A system for supporting multi–language versions for software fault tolerance. In *Digest of Papers, FTCS–19, The Nineteenth International Symposium on Fault–Tolerance Computing*, pages 268–274, Chicago, Jun 1989.

[14] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE–1(2):220–232, Jun 1975.

[15] Wilhelm Rossak and Peter A. Ng. Some thoughts on systems integration: A conceptual framework. *Journal of Systems Integration*, 1(1):97–114, 1991.

[16] Wilhelm Rossak and Peter A. Ng. System development with integration architecture. In *Proceedings of The Second International Conference on Systems Integration*, pages 96–103, Morristown, New Jersey, Jun 1992.

[17] Scott R. Tilley, Hausi A. Müller, Michael J. Whitney, and Kenney Wong. Domain–retargetable reverse engineering. In *Proceedings of Conference on Software Maintenance*, pages 130–139, Montreal, Sep 1993.

[18] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–30, Aug 1993.