A Model and Methodology for Distributed Integration

C.V. Ramamoorthy,
Claudia Chandra, Han Kim
Y.C. Shim, Vikram Vij

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

Abstract

In this paper, we will first identify some of the problems in integration. Then, we will give a survey of the various integration strategies that have been proposed in the literature. The success of these integration strategies depend heavily on the characteristics of the application and choosing the appropriate strategy has been for the most part done subjectively. What is needed is a general model which allows us to quantitatively explore the tradeoffs and select a strategy or combination of strategies. We propose such a model for the integration process based on graph theory. Our model provides a quantitative method for comparing the different integration strategies to be used for integrating a particular system. Another issue in systems integration is maintaining the consistency of assumptions made by different software engineers. We suggest a technique to keep track of consistent assumptions during the development process.

1. Introduction

The typical software development environment involves a very large and complex system, which is developed under a distributed multi-disciplinary effort. The typical size of application programs commonly ranges from 10^{16} to 10^{26} bytes of codes. Examples of such large systems are real time process control systems which has an average of 0.5 million lines of code (MLOC) and space systems which are in the order of 1.26 MLOC. Military avionics systems are another application area which are large and multidisciplinary in nature, requiring expertise in guidance systems, flight control, weapon systems, life support, and communications.

The difficulty of integrating such complex systems is that the software developers must have a complete knowledge of the whole system in order to design, test, and integrate the system correctly. In practice, however, the developers may have different assumptions about the functionality of the modules and the interactions among them. An added difficulty lies in the inability to observe module interactions completely. As a result, when the modules are integrated, inconsistent and conflicting assumptions may cause the system to fail.

Another issue in systems integration is how to partition the system into collections of modules to be tested independently and come up with an optimal schedule for integrating and testing those partitions. The points to be considered here are the cost of testing each partition, the degree of parallelism that can be achieved during the testing process, and the total effort that has to be spent to integrate the partitions. Many different integration strategies, such as the Big-bang, top-down, and bottom-up integration have been proposed, but none of them consider the properties of the application program. As a result, these strategies may be effective only for a specific class of applications.

There are several levels of integration. Total system integration involves the integration of separate system modules into a complete working system. This is usually done when the software product is first developed. The other types of integration involves the introduction of new functions or new technology. This is usually performed during the maintenance stage of the software development. New functions integration may be required as a result of changing user requirements or changing conditions in the environment; while new technology integration is required to keep up with the development of new techniques in the field.

In developing a methodology for systems integration, we unfortunately have no theoretical foundations on which we can build our solutions. Neither has much relevant experience in the field been documented. Therefore, we are forced to rely on our intuition and borrow from the experiences and techniques in other fields, such as the hardware industry, where components and integration processes have been largely standardized. In software, we see the same need for standardization. If software systems are built using standard interfaces and standard interchangeable parts consisting of well understood functions, then the integration process would be more tractable.

In this paper, our goal is first to identify some of the problems in integration. Then, we focus on the second issue outlined above and suggest an approach which attempts to lay a theoretical basis for an integration methodology. We will also briefly suggest an approach to the first issue above, but will leave the details as a topic for a future paper. The rest of the paper is organized as follows. Section 2 gives an overview of the various integration strategies that have been proposed.

0-8186-2755-7/92 $03.00 © 1992 IEEE

Section 3 presents a graph theoretical technique for modeling the integration process. The graph model provides a basis to quantitatively compare different integration strategies. Section 4 presents a brief overview of the technique to maintain consistent assumptions using the truth maintenance system. Finally, in section 5 we conclude with some suggestions on what we think are important areas for future research in systems integration.

2. Strategies for Integration

The various integration testing strategies can be classified into three categories. In the first category, we can classify the horizontal and vertical integration methods. Horizontal integration combines modules which perform related functions first, while vertical integration combines modules which are implemented using similar technologies first. An illustration of horizontal and vertical integration is shown in Figure 1 and Figure 2 respectively.

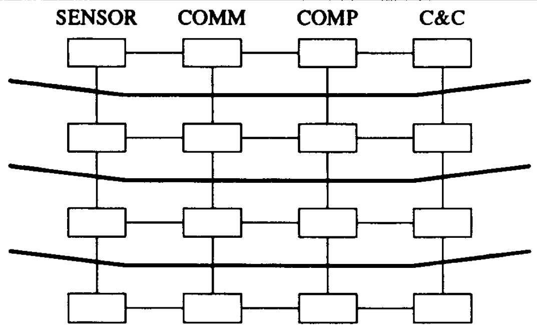


Figure 1. Horizontal - Functionality-Based

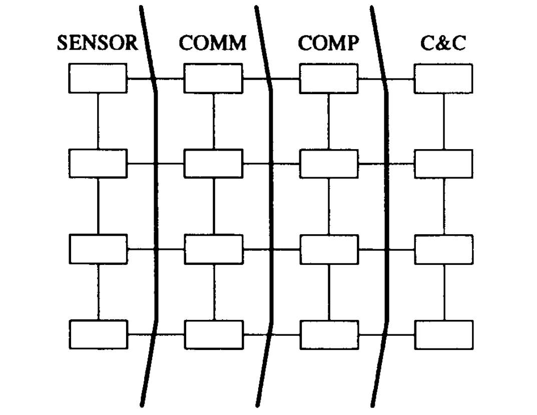


Figure 2. Vertical - Technology Based

The second category is the flow oriented integration method, which are mainly variations of the incremental, sequential, or parallel integration techniques. This class includes such well known integration methods as the Big-bang, bottom-up, and top-down approaches.

In the Big-bang integration and testing, each of the modules is tested individually and in isolation. The order of testing of these modules is dependent upon the software development environment, and on the number of people involved in the development. As a final step, all the modules are combined and a single suite of integration tests is performed. This method is perhaps attractive for very small software systems. The integration phase is short and therefore results in cost as well as schedule savings. However, Big-Bang integration testing is very inefficient for medium to large systems.

The drawbacks of non-incremental integration testing such as the Big-Bang method, can be dealt with by taking an incremental approach to integration testing. In incremental integration testing, the next module to be tested is first combined with the set of modules that have already been tested. There are several incremental testing strategies: top-down, bottom-up, and critical module.

In top-down integration testing, the top module of the program, also called the *control module*, is tested first. For this module to be tested, lower level stub modules must be written but a driver (platform on which to test them) is not required. The major focus is on interface testing which provides early identification of interface errors among modules. The disadvantages of using this method are: errors in the low-level critical modules are not found until later, the writing of stubs is a non-trivial task, and there are no formal rules for integrating the lower level modules once the top level modules have been tested.

In bottom-up integration testing, the modules at the lowest levels are tested first. The key aspect here is the focus on individual module functionality and performance. Therefore, errors at the lower level are spotted first. However, as in the top-down testing, there are no formal rules for finding the next module to be integrated and tested, once the lowest modules have been tested. Another significant disadvantage of bottom-up integration testing is that a working program is obtained only when the final modules (the top level modules) are integrated. Although stubs are not needed in this approach, drivers must be written in order to test the low level modules in the absence of their higher level counterparts. The advantages of top down testing become the disadvantages of bottom-up testing and vice versa.

Critical module testing is a compromise between the top-down and bottom-up approaches of integration testing. In this approach, the critical modules of a system are identified and tested first, after which the remaining modules are integrated. The criteria for selecting the critical modules depend on the application.

Other integration methods, such as modified top-down, sandwich, and modified sandwich are reviewed in [Myers]. Carey and Bendick suggest a strategy called build testing [Bendick]. The "mixed bag" strategy, which is a combination of bottom-up, top-down, Big-bang, and build testing is presented in [Beizer].

The third category is the redundancy based integration, which is commonly used to introduce new technology. Old components are gradually phased out by using them as a stand-by while trying out the subsystems using the new technology.

3. A Graph Theoretical Methodology for Scheduling

Various strategies for integrating a complex software system have been suggested and applied in the software industry, as described in Section 2. However, the successful implementation of a particular integration strategy depends heavily on its application characteristics. For example, Big-bang integration is more suitable for small-scale software systems. Top-down testing, on the other hand, fits well with applications where frequent change of requirement or design is anticipated. Notice that these approaches are not mutually exclusive; it is often feasible to adopt a compromised or a combined version of different strategies. Choosing the appropriate strategy, however, has mostly been subjective and dependent on the manager's intuition.

Integration planning is seldom high on the list of software development activities. However, its impact on development cost and product quality is significant. As all software development projects must deal with the conflicting considerations of resources versus quality, it is important to investigate the problem of integration more carefully.

What is needed is a conceptual model on which we can quantitatively explore tradeoffs and compose a solution. Furthermore, the model itself must be general enough to encompass the entire solution space. In this section we propose a graph-theoretic model along with related techniques to achieve this goal.

Past research on modeling software systems has made extensive use of graph theory. Studies on automating the testing process are mainly based on the manipulation of control flow graphs. Control- flow graphs give the precise sequence of execution, namely the control paths in the programs. Knowing the control structure helps the manager to understand the extent of test coverage and to optimize the allocation of testing resource. Data flow graphs, on the other hand, describe the exchanges of information among modules and illustrate the propagation of errors or changes. We feel that although the control-flow model is suitable for unit testing, it does not provide the appropriate level of abstraction to describe relationships among modules in large software systems. Therefore we propose a model that is similar to the data flow model. However the inter-module relationship is not constrained to the information exchange between modules, as we will see later in the discussion.

In our model the software modules are represented as nodes (or vertices), and the edges describe the relationship between them. Instead of interpreting the relationship as data flow or control transfer, we take a more general point of view. The meaning of the edges can be assigned according to the application characteristics, as long as it serves as a quantitative measure of the degree of interaction between modules. For example, the following metrics could be used:

- Amount of data exchange between two loosely coupled module
- Amount of shared data between two tightly coupled modules
- Amount of shared library code between two modules
- Number of functions in which the two modules are both invoked
- Similarity in implementation technology (e.g, programming language or platform)
- Any combination of the above

Similarly, we may assign the following meaning to the nodes:

- Amount of code in the module
- Complexity measure of the module
- Amount of data imported, exported, or retained by the module
- Performance or deadline requirement, if any

We view integration as an iterative process, where at each step a collection of subsystems is selected for integration. At each subsequent step the subsystems formed at the previous step are integrated into a bigger cluster. The process is repeated until only one cluster is left, that is until all the modules are integrated into the total system.

The clustering steps can be thought of as successive abstractions of the modules. At the bottom level we have a weighted graph representing individual modules and their interactions, which we will call G. Weights are assigned to the nodes and vertices to reflect the complexity of testing. The clustering of modules is therefore equivalent to grouping the nodes and edges in the corresponding subgraph into a supervertex. The supervertex formed is opaque to the resulting graph - after the modules are successfully integrated we choose not to worry about the their interactions any more. The supervertices are further clustered in the same fashion until we are left with a single supervertex, which represents the successfully integrated system.

Let us denote the cost of integrating modules represented by a graph G as $Cost(G)$. The clusters (subsystems) B_i's are themselves subgraphs of G. After the B_i's are successfully integrated, they are abstracted into supervertices, Let us call the resulting graph (referred to as the reduced graph hereafter) G_p. Now the cost of integrating G becomes the cost of integrating modules in the individual clusters along with the cost of combining the clusters. The idea can be defined recursively in the equation that follows:

$$Cost(G) = (\sum Cost(B_i)) + Cost(G_p), \quad B_i \ in \ P$$

where $P = B_1 \cup B_2 \cup \cdots \cup B_n$ is a partition on G. G_p is the abstracted graph of G based on P.

When G contains only one vertex then the cost is simply the cost of unit testing.

To minimize the cost of integration we must find a good clustering strategy, or equivalently, a plan to successively partition the system. Our goal is to achieve

$$Min\_Cost(G) = Min\ ((\sum Min\_Cost(B_i)) + cost(G_{p_j})),$$

over all P_j on G, where the P_j's are all possible partitionings of G.

An additional consideration when forming the clusters to be integrated is the semantic behavior of the system as well as the structural characteristics. That is, the weights in the graph commonly characterizes the structural properties of the program, but not the semantic aspects. For example, although two modules may be tightly coupled and exchange a lot of data, another module may be required to provide input to one of the modules. Therefore, these three modules may be best clustered together. We will not discuss this point further in this paper. However, an approach for dealing with the semantic behavior of a software system is discussed in [Yau], which proposes a method for distributing functional modules to different software components efficiently.

Prior to integration testing, a plan for grouping and scheduling the test effort for a collection of modules is crucial for a cost effective integration strategy. Such a scheduling strategy should minimize the amount of interaction and dependencies between modules. The functional units within modules must have a high degree of cohesion. They could make up a single control sequence. All the elements of a module could operate on the same input data or produce the same output data. Alternatively, the output from one element in the module could serve as the input for some other element. Yet another way of cohesion is that each unit in the module could be essential for the execution of a single function. The units could also have strong interconnections (e.g. make use of shared data) or exchange control information (communicate). They should be functionally related. This allows the program manager to have a better understanding of the integration process and its status. It also helps to reduce the testing cost in the sense that the information one has to collect and monitor is well defined. So we can also incorporate functionality semantics in assigning weights.

There is a close resemblance between our problem and the problem discussed in [Yau] of minimizing the communication overhead using clustering and shortening the critical path (execution time). The isomorphism is that in our approach, the weight of a node signifies the cost of testing, while the edge weight also signify the testing cost. In the distributed computing clustering and scheduling problem, the weights of nodes and edges signify the execution cost. In both approaches we try to minimize the respective costs.

Using this graph theoretic model, the costs of various integration strategies can be computed and compared for their efficiency. Consider an example using the subsystem graph in Figure 3. For our purposes, the numbers in the parentheses denote the module complexities, and the number on the edges denote the interface complexities.

In order to test a subsystem, stubs are required at the cut-off points, either to supply the inputs to the subsystem, or to monitor the output of the subsystem. The cost of subsystem integration, therefore depends on the following factors:

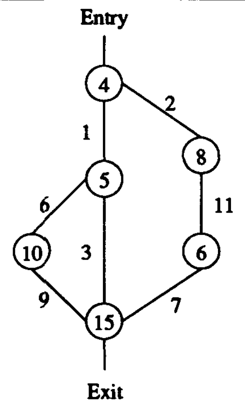


Figure 3. Subsystem Example

(a) The number of stubs to be inserted at the cut-off points.

(b) The test cases required for each stub.

(c) The cost of devising a test case.

In the Big-bang integration strategy, the entire subsystem is integrated at once. Therefore, the number of stubs required is 2, one used at the input, and one at the output. The number of test cases required is $1 * 6 * 9 + 1 * 3 + 2 * 11 * 7$, since there are three paths in the graph, and the test cases required for each path is the number of combinations of interface configurations along the path. The combined complexity of the modules is taken as the maximum path complexity, which is $4 + 5 + 10 + 15$ in our example. Assuming that the cost of a test case is a linear function of the combined subsystem complexity, then the cost of integration using the Big-bang strategy is:

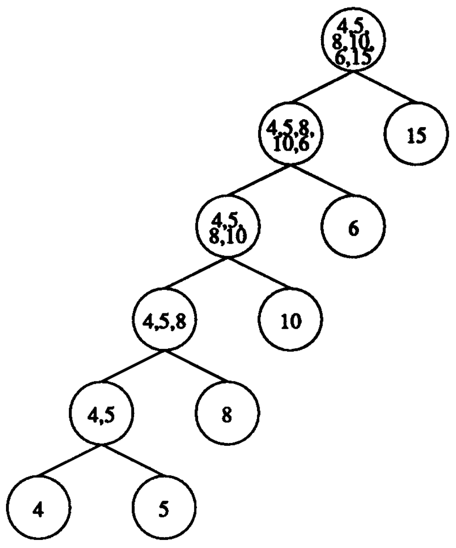
$$Cost = 2 * (1 * 6 * 9 + 1 * 3 + 2 * 11 * 7) * (4 + 5 + 10 + 15) * C = 14348 * C$$

where C is a constant representing the cost of running one test case.

Figure 4 gives an example of the cost of using the incremental integration strategy on the same subsystem graph. Other interpretations for the weights of nodes and edges are possible. These examples are only meant to illustrate how the graph theoretical model can be used to compare the various integration strategies for a particular system and to select an efficient integration schedule.

4. Maintaining Consistent Assumptions Using the Truth Maintenance System

The transformation between one stage of the software life cycle to the next often requires many iterations, where the client's fuzzy requirements are translated to formal specification, which in turn forms the basis for a design. During the transformation between stages, assumptions usually



Cost: 4x1x(4+5)C
+4x2x(4+5+8)C
+4x6x(4+5+8+10)C
+4x11x(4+5+8+10+6)C
+2x(9+3+7)x(4+5+8+10+6+15)C
= 4096C

Figure 4. Incremental Integration

have to be made about the interpretation of certain requirements or specification, in order to make progress in the next stage. This is especially true when requirements are written in natural language, which is inherently ambiguous.

When system developers are spread across geographical areas the problem of maintaining a consistent set of assumptions becomes worse, due to lack of interaction and communication. If incorrect or conflicting assumptions are made, they are either corrected during iterative refinements of the previous stage or they remain undetected until the system is tested, integrated, or put into use. To deal with this problem, we need a model of the dependency between the stages and the assumptions that are made at each stage. This dependency information is used to guide the system developer to make the required changes in subsequent stages, when refinements at a stage reveal some false assumptions.

A dependency network can be used to represent the dependencies between the requirement and specification, between the specification and design, and between the design and implementation. Using the dependency network, truth maintenance techniques should guide the designer to identify which part of the design and codes may need to be modified, when refinements have been made to the specification. Similarly, tuning the requirements or revising the design should make use of the same mechanism to determine the changes that are required at the later stages.

During the integration stage, the truth maintenance system can also be used to identify the set of false assumptions that cause a subsystem to fail. The truth maintenance mechanism then resolves contradicting assumptions and keeps the system in a consistent state. The details of the mechanism is beyond the scope of this paper and will be discussed further in a future paper.

5. Conclusion and Future Research

In this paper, we have suggested a theoretical approach for evaluating the different integration strategies and to decide among them for integrating a specific system. With our model, other integration strategies that do not fit into those that have been proposed in the literature are possible. A good integration plan should integrate tightly coupled subsystems first and allow as much parallelism during integration testing as possible. If time instead of labor is the scarce resource for a project, then the objective should be to minimize the cost (in this case the required time) of the critical path in the integration plan. Our graph model can be used to account for such requirements as well.

Much work remains to be done in systems integration. Research in the area of reusability and refinements of metrics guided methodologies would also contribute to the advancement of research in the systems integration field. As we mentioned previously, the software industry lacks the existence of standards. If software systems were built using reusable modules, then the integration effort would be greatly reduced. Experience in the systems integration area is also scarce. The collection of metrics which characterize a system with desirable properties for successful integration would be very helpful. In addition, further work on theoretical foundations for testing and systems integration is needed.

References:

[Beizer]

Beizer, B., Software System Testing and Quality Assurance <1984> New York: Van Nostrand Reinhold c1984.

[Myers]

Myers, G., Software Reliability, Principles and Practices, New York, Wiley, c1986., Business Data Processing.

[Myers]

Myers, G., The Art of Testing, New York, Wiley, c1989.

[Leung]

Leung, H., K., N., White, Lee, A Study of Integration Testing and Software Regression at the Integration Level, IEEE Proceedings of the Conference on Software Maintenance 1990,290-301 .

[Savolainen]

Savolainen, T.,Software Integration Technologies in CIM, Hewlett Packard Journal, 83-99(1990).

[Yau]

Yau, S., Wiharja, I. , An Approach to Module Distribution for the Design of Embedded Distributed Software Systems, Information Sciences 56, 1-22 (1991).

[Yau]

Yau, S., An Integrated Expert System Framework for Software Quality Assurance, IEEE Compsac 1990, 161-166