INTEGRATION IN SOFTWARE INTENSIVE SYSTEMS

Victoria Stavridou
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025
USA
email: victoria@csl.sri.com

Abstract

This article is concerned with systems integration and its impact on software intensive projects. It contains a review and a taxonomy of integration concepts as applied to software intensive systems. We identify pertinent technical integration issues and review and classify integration models, strategies, mechanisms and architectures. We argue that integration is part of the design activity and propose existing best integration practice for project management and engineering of large software intensive systems.

1 INTRODUCTION

"Integration" is an often misused term. Typically, it means different things to different communities with predictably confusing results. This confusion is very harmful; by not understanding what integration is, opportunities for planning and budgeting for it are not used leading to development schedule overruns and ultimately, we believe, to inferior software systems. The size and complexity of typical software intensive systems is such that this problem is likely to be exacerbated in the future.

This article is particularly concerned with and goes to some lengths to clarify integration terminology. As a first step, we need to distinguish between:

- *What integration is*, that is the conceptual *models* of integration as well as the *kinds* of integration in use.
- *How integration is achieved*, that is, development *strategies*, *mechanisms* enabling integration and integrated system *architectures*.
- *How integration is verified*, that is, *testing* of a developed system to ascertain whether the integration goals have been achieved.

These distinctions, although rather obvious, are ignored surprisingly often in the literature. For instance, B. Beizer routinely equates "integration" with "integration testing" and "integration strategy" (Beizer, 1984). Some useful distinctions are drawn by the European Software Factory team in (Nilsson *et al.*, 1990).

In this paper we are concerned with what integration is and how it is achieved. In order to set the scene early on, the next section contains a number of important definitions clarifying concepts and kinds of integration. Sections 3 through 6 examine in detail the achievement of integration through its constituent parts of strategies, mechanisms and architectures. We propose a method for achieving integration in section 7 and conclude with a discussion in section 8.

2 A taxonomy of integration concepts

2.1 What is systems integration?

Systems Integration is the practice of combining the functions of a set of subsystems, be it software, hardware or both, to produce a single, unified system that satisfies some need of an organisation (paraphrased from (Kuhn, 1990)).

This definition is intentionally very general and its components would, in principle, involve not just digital systems but also analog components, power supplies, enclosures, connectors, tools and so on. However, in this article we use the term "systems integration" to denote "digital systems integration". The components of digital systems integration typically are:

- software subsystems (applications, system, custom)
- hardware subsystems (single/multi processor(s))

An example, typical of such integration activity, might be an air defence system consisting of components as shown in figure 1.

2.2 Software integration, tool integration and integration testing

Systems integration is neither *software integration* nor *tool integration* (as is implied for example by (Ramamoorthy *et al.*, June 1992)). And, despite (Beizer, 1984), it is not *integration testing* either.

> **Software integration** is the practice of assembling a set of software components/subsystems to produce a single, unified software system that supports some need of an organisation.

Typical software integration components are applications, reusable modules, bespoke modules, Commercial-Off-the-Shelf systems (COTS) and Government Off-the-Shelf systems (GOTS).

The term *tool integration* is virtually exclusively used in the literature to denote Computer Aided Software Engineering (CASE) tool integration. Therefore we define it as follows:

> **Tool Integration** is the practice of combining a set of software development tools to produce a single, unified software development environment.

Typical tool integration components are analysis tools, design and coding tools, requirements traceability tools, configuration management tools and so on.

To summarise then, tool integration is a form of software integration which in turn is a form of systems integration, as shown in figure 2.

Integration testing on the other hand is a very different notion and can be defined as:

> **Integration Testing** is a set of procedures designed to verify whether a given assembly of components (be they systems, software or tools) does indeed satisfy the requirements of an organisation.

2.3 Integration "in the small" *vs* Integration "in the large"

As well as identifying the true types of integration, it is useful to draw a distinction between the nature of integration depending on the granularity of the components involved.

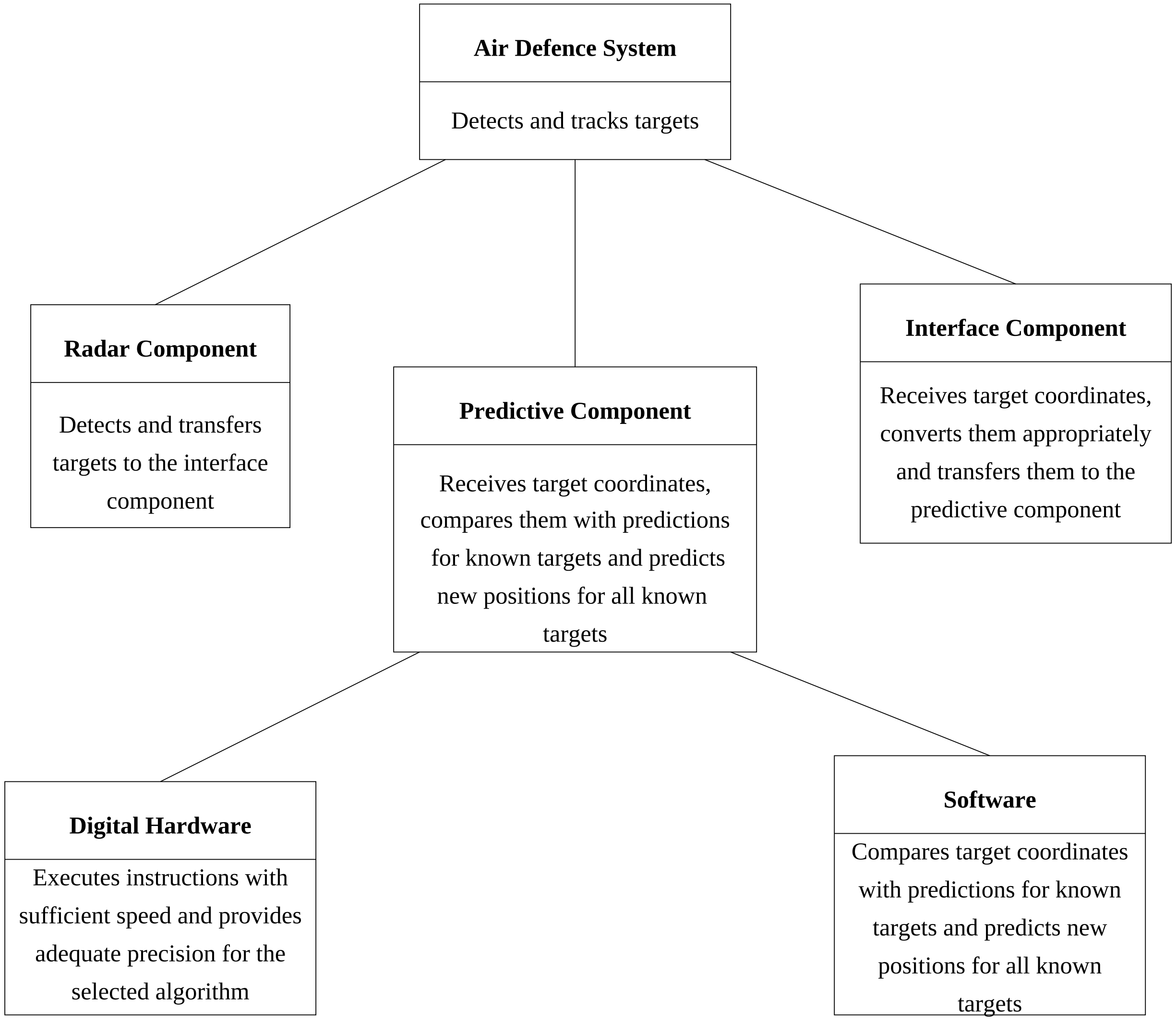

Figure 1: An example of systems integration

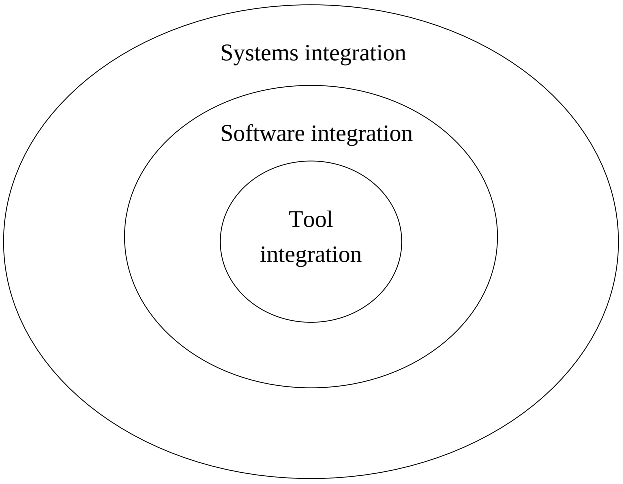

Figure 2: A hierarchy of integration

Fine grain integration, or *integration in the small*, typically involves software components without independent functionality. These components can only function as part of the integrated system. Examples are program functions, procedures and packages. The granularity of the components is important because it identifies the "glue" used to put them together. In the case of fine grained integration, the glue is provided by either programming language constructs or the compiler.

On the other hand, *coarse grained integration*, or *integration in the large*, normally would involve software or hardware components with well defined, independent functionality. Such components can function independently of the integrated system and might include applications software, reusable modules, bespoke modules, COTS, GOTS and so on. The glue in the case of coarse grained integration is much more varied and open ended than is the case with fine grain integration. This is where integration "glue" becomes much more of an interesting challenge.

In summary, systems integration aims to support some organisational *processes* through the supply of some *services* enabled by the existence of certain integration *mechanisms*. This view of integration is shown in figure 3. In the rest of this paper, we concentrate on coarse grained software integration.

3 CONCEPTUAL MODELS OF INTEGRATION

Most of the existing work on modelling coarse grained software integration concepts seems to have originated from the CASE community (Brown *et al.*, 1994). CASE researchers have sought a conceptual framework through which integration issues can be more easily understood and effectively

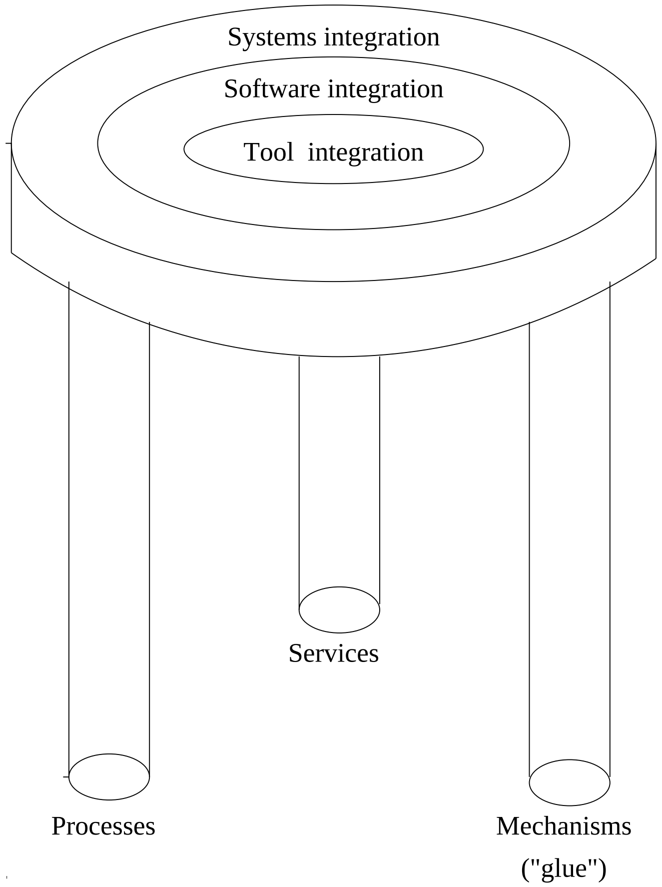

Figure 3: Integration and supporting notions

communicated. In this section, we generalise the CASE approach to integration concepts relevant to coarse grained software integration.

Four detailed conceptual models have been identified:

- Integration as a set of attributes.
- Integration as goals for the relationships between components.
- Integration based on a central repository through which components share data.
- Multilevel integration.

To a certain extent, these approaches are based on different notions of what integration actually is. We will describe each of the four approaches and conclude with a conceptual model best suited to our notion of integration as defined in the previous section.

3.1 Integration as a set of attributes

Wasserman (Wasserman, September 1989) views integration as a set of attributes. These attributes can be used as a means of evaluating, comparing and planning the process of integration. Essentially, this approach allows various aspects of integration to be examined separately through a set of orthogonal dimensions.

Wasserman proposed the following five attributes:

- **Platform integration** is the set of system services that provide network and operating systems transparency to the components.
- **Presentation integration** refers to the consistency and sophistication of the user interface.
- **Data integration** characterises data sharing between components, be it through a shared repository (database) or files and interprocess communication.
- **Control integration** relates to inter-component communication.
- **Process integration** characterises the extent to which the integrated system supports a user process in a well defined way.

The essential idea here is that any product of system integration can be evaluated in each of these dimensions. For example, in the data integration dimension, one system may use a filestore while another uses a database. These systems would have different data integration attributes. So Wasserman's approach allows some measure of comparison of different systems in terms of their integration approach. These ideas can be easily extended to apply to proposed system designs as well as finalised systems and may be quite useful in that context. Figure 4 illustrates three of the integration dimensions proposed by Wasserman.

As a mechanism of evaluating an integrated system, Wasserman's model is somewhat crude since the axes indicate sophistication of integration mechanisms rather than integration quality; more does not necessarily mean better. The five attributes themselves may be debated since the separation is not as clear as Wasserman suggests; for example there is a close and complex relationship between data and control integration. The true value of this model may well lie in identifying these different aspects which can thus be considered and acted upon during the design of an integrated system.

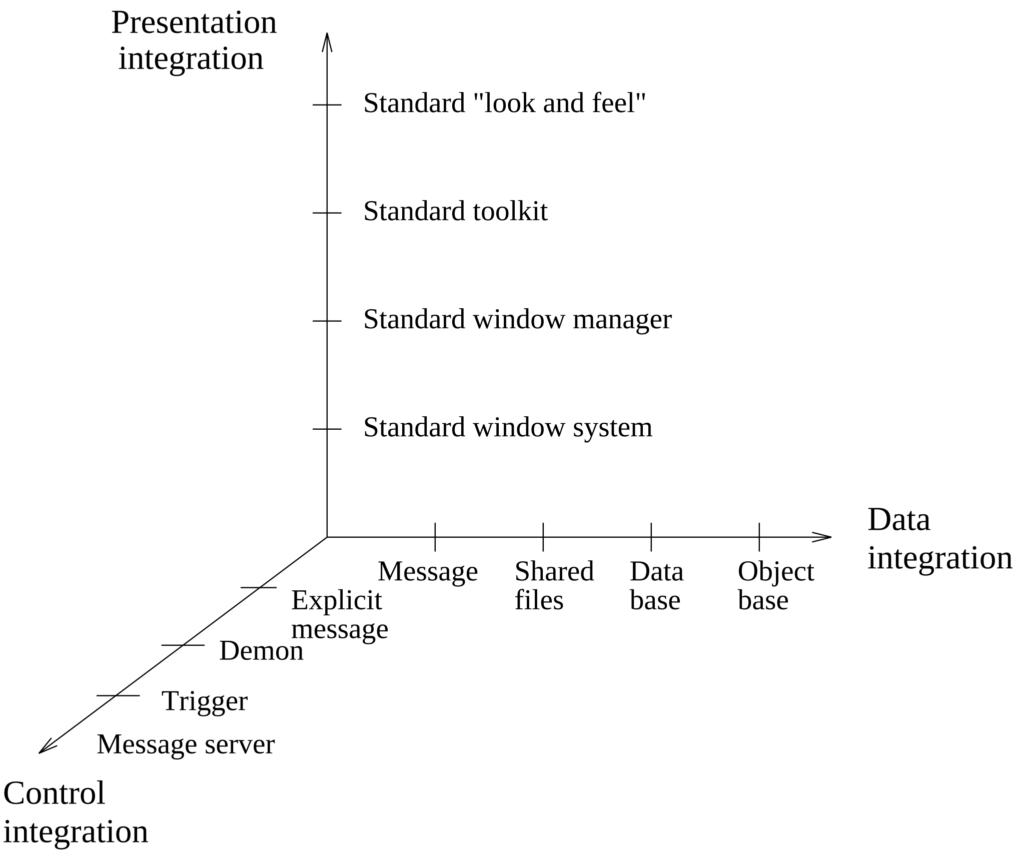

Figure 4: Three integration dimensions

3.2 Integration as a relationship

The approach of Thomas and Nejmeh (Thomas & Nejmeh, 1992) treats integration not as a property of a component, but as a property of the relationship between components. In particular, they are concerned with defining a set of goals between the components of an integrated system. By doing this, attention is focused on the goals of integration and on the information required to decide whether these goals have been reached. This conceptual model is an improvement on Wasserman's approach since now not only do we know which integration aspects to watch out for, but what information we need in order to decide whether the integration goals have been met.

Thomas and Nejmeh identify several types of inter-component relationships (Brown *et al.*, 1994):

- *Component to component* relationships which expand on Wasserman's dimensions of integration by discussing the ways in which individual components interrelate.
- *Component to framework* relationship which defines the manner in which each component

utilises the framework (e.g. a database or an operating system) services.

- *Component to process* relationship which captures how well each component supports the objectives of the integrated system. The relationship may be expressed in terms of a component's support for an individual step within the software lifecycle (e.g. requirements definition), and its support for multiple steps (e.g. the requirements, design and coding steps).

This aspect of the model is illustrated in figure 5.

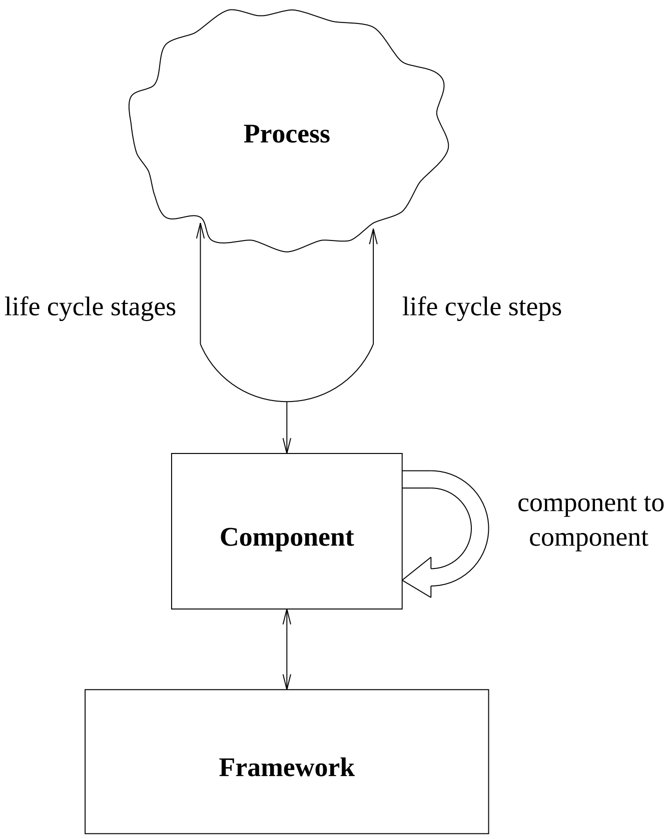

Figure 5: Integration relationships between environment components

Thomas and Nejmeh have enriched Wasserman's model in their efforts to characterise component to component integration. They are particularly concerned with the following relationships:

- Presentation
- Data
- Control

- Process

Figure 6 shows an entity relationship diagram depicting a component, the four relationships and the properties of each relationship.

In addition to the elaboration of integration relationships and their properties, Thomas and Nejmeh draw a distinction between two different viewpoints in the discussion of integration; the system user's and the system builder's. This is a useful distinction since it highlights the difference between the user's perspective (is this system well integrated?) and the builder's perspective (how easy is this system to integrate?). For example, there may be a consistent and intuitive user interface for interaction with all system components. On the other hand, a well integrated system is not necessarily easily integrable with respect to how easy it is for the system builders to assemble the system, tune it for particular needs, and replace one component with another. These two views of integration are independent; there is no guarantee that a system that is easy to assemble is enjoyable and productive to use and vice versa.

The more precise Thomas/Nejmeh integration model can be useful to users, component evaluators, component developers and framework technology builders. However it has two important shortcomings. First, although they identify the goals of integration, they do not propose a way of achieving these goals, what dependencies exist between them and what trade offs must be made. Second, a component is not equivalent to an end user service which divorces the discussion of component to component relationships from the reality of system use which is the final indicator of success or failure.

3.3 Repository based integration

Repository based integration views the issue primarily from the data sharing point of view. In fact there is a pervasive belief amongst many people that a repository of some sort should be the primary means of component integration (Brown *et al.*, 1994).

A repository has four characteristics:

- *A data storage mechanism.* These range from simple file systems to complex databases possibly distributed over a number of sites. The data storage mechanism is responsible for the safekeeping of persistent instances of data. Frequently used examples of data storage mechanisms are UNIX file systems and relational databases. More advanced instances of the

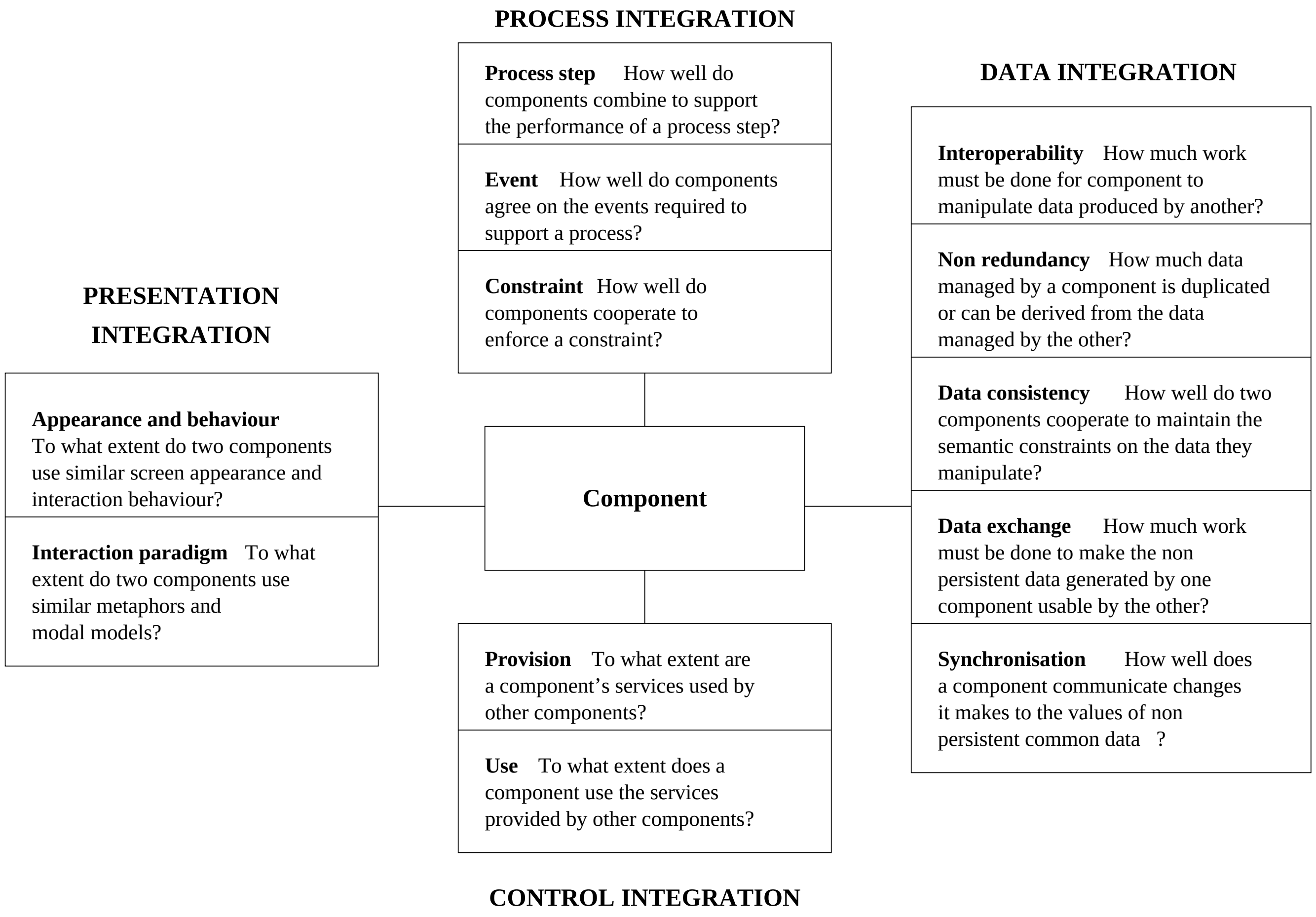

Figure 6: ER diagram of a single component, four relationships and their properties

mechanism, such as object-oriented databases and entity-relational object storage systems (such as PCTE) are rarely used.

- *An interface to persistent data.* There may be multiple interfaces to the same data or one interface may provide access to a number of different data storage mechanisms. The prevalent interfaces are SQL and UNIX (Posix) standards. In addition a number of bodies such as the IEEE, ANSI, ISO and the ECMA all have efforts to define new interfaces to facilitate CASE tool portability across different CASE environments.

- *A set of schemata or information models.* A schema is a description of stored data and typically records information about the different types of data in the repository and their relationships. There have been a number of attempts to define generic information models and there still is a great deal of research in the area mainly along the lines of "enterprise modelling and integration". Extensive efforts in this area include the IBM AD/Cycle Information Model and ISO Information Resource Dictionary System (IRDS).

- *A set of operations on the data.* This determines how the data in the repository are accessed, by whom and at what frequency. This information is critical to tuning the repository for performance.

We want to emphasise that a repository is not in itself a *guarantor* of integration; it is a *mechanism* which facilitates it. In order for the mechanism to be used successfully to integrate a system, the components must first come to an agreement on the storage mechanism, the interface, the schema and the operations.

3.4 Multilevel integration

(Brown *et al.*, 1994) have proposed a multilevel model of integration which combines most of the features of the previous models in a meaningful assembly. They distinguish three levels as shown in figure 7.

The central level consists of the *services* available to system end users. The bottom level consists of the *mechanisms* that implement the end user services. The top level of interest is that of the *process* which encodes the set of goals and constraints of the context in which the end user services will be used to meet some objective of an organisation.

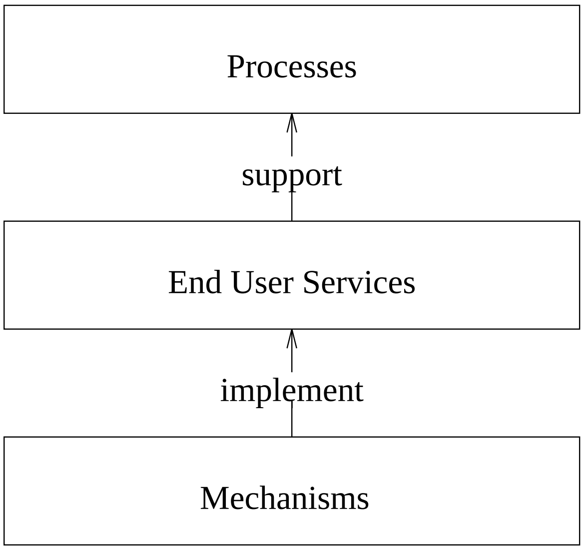

Figure 7: A multilevel integration model

- The *processes* level characterises the use of end user services in the context of a particular operating scenario, in other words, *what activities does this system support*. Integration at this level can be regarded as a kind of process specification of how the system will be used in the field. This specification may be in terms of:
  - *Goals* such as deliverables, improvement or measurement, and
  - *Constraints* due to the lifecycle used, the schedule involved and even the politics of operational system use.
- The *end user services* level corresponds to an abstract description of the functionality that the system offers to the end user. So integration at this level delineates *what the component assembly does*. These services may include amongst others:
  - *Mission related* services such as detection and tracking of targets in the case of the Air Defence System in figure 1.
  - *System management* services such as planning, estimating and tracking mission activities.
- The *mechanisms* level includes the architectural concerns and the technology components that will implement the integrated system. So this level describes how the various system components are connected. Concerns here might be:

- *Architecture* aspects such as internal component structure (e.g. client/server) and data management structure (e.g. data dictionary, database).
- *Technology* related aspects such as operating system interfaces and specific integration mechanisms and standards used to connect the various components (e.g. RDBMS, X, PCTE).

The multilevel model, although originally devised for application in CASE tool integration, is clearly capable of generalisation and is applicable to coarse grained software integration (as we discuss in section 7). The value of the model comes from systematising many previously *ad hoc* aspects of integration. It provides a powerful conceptual framework for relating the functionality required from an integrated system to the integration technology that realises it.

This is our preferred model of integration because it encourages and supports *integration as a design activity* rather than an after system development add-on activity.

4 ARCHITECTURES FOR INTEGRATION

Building an integrated system requires not only a conceptual model of what integration means, but also some practical guidance on how to go about constructing such a system. The *architecture* of the integrated system is the first design choice that needs to be made. This architecture defines the way that the components cooperate in order to achieve the objectives of the integrated system, and therefore plays a central role in the success or failure of the integration effort.

Two approaches to integration architectures are distinguished in the literature; *component coalitions* and *component federations*. These correspond to the notion of *closed* and *open* integration respectively and essentially determine the approach of an organisation to *open systems*. In the remainder of this section we describe the two approaches and discuss the impact that the choice of architecture has on the integration process for an organisation.

4.1 Component coalitions

Component coalitions are also often referred to as "point-to-point" integration architectures (Brown *et al.*, 1994; Hanratty, June 1995) as well as "component-based" integration architectures. (Marty, 1988) characterises this approach as *closed integration*. The essence of it is that independently developed components such as tools and applications are integrated through a careful "hardwiring" of their interfaces as shown in figure 8. Each such assembly of components requires a custom integration solution in order to achieve its objectives. The components maintain their independence as attested by a number of characteristics (Morris *et al.*, December 1991):

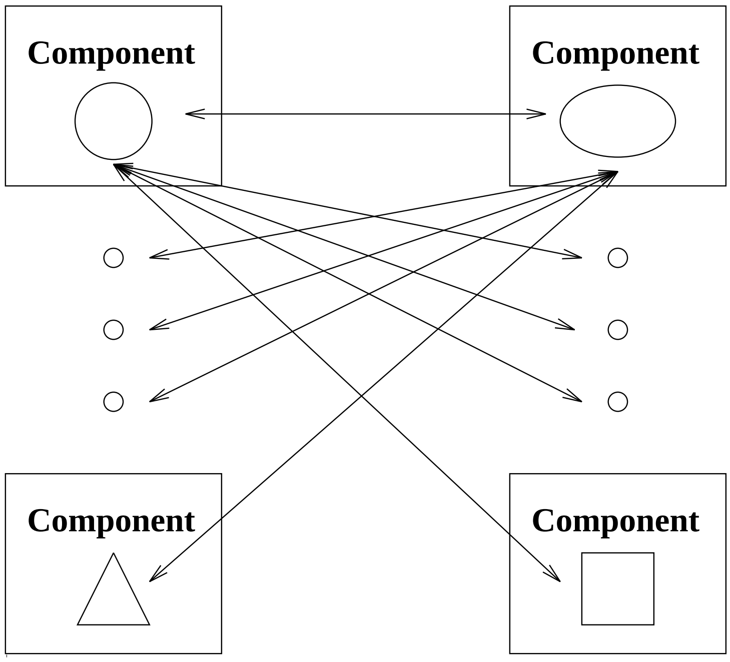

Figure 8: “Point-to-point” integration

- Each component maintains its own interface.
- Each component maintains independent control of its data and processing.
- Components may provide overlapping, even contradictory functionality or services.

Although component coalitions are extremely useful for many purposes, they have a number of key problems such as:

- *Lack of generality* since they can only be used for a fixed well defined objective.
- *Lack of evolvability* since change in either components or system purpose cannot be accommodated within the hardwired solution.
- *Component egocentricism* as reflected in uneven service provision.

Because of the lack of standardisation and the emphasis on "no change" policies, component coalitions can provide an immediate, but partial solution to systems integration. This is already evident in the CASE integration world, where vendor coalitions abound (for instance the coalition of IDE's Software Through Pictures and CodeCenter with Frame Technology's FrameMaker).

4.2 Component federations

Component federations refer to "integration through standards" (Hanratty, June 1995) architectures and *open integration* (Marty, 1988). The key issue in this approach is the provision of a "plug and play" platform which can support a variety of components conforming to a set of agreed standards. In contrast with component coalitions, one does not set out to build a custom interface harness to enable integration. Instead, the emphasis is on interface and other standards. Components conforming to such standards can be assembled in a "plug and play" architecture in a straightforward way, as shown in figure 9. An actual example of a component federation is the Eureka Software Factory reference architecture (Fernström *et al.*, 1992).

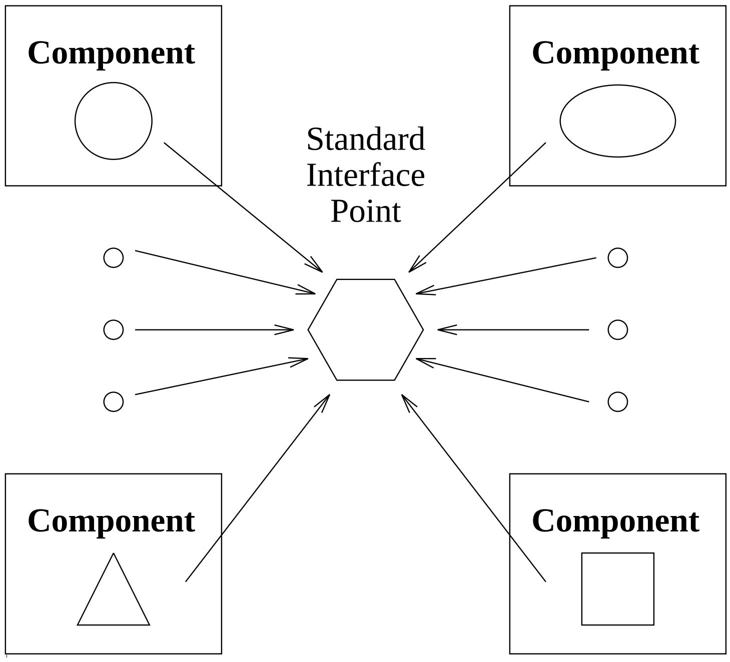

Figure 9: Integration through standards

Although components can maintain their own data storage and access mechanisms, the platform provides specialised services to allow components to communicate actions and intentions as well as to share data. Typically, the procuring organisation maintains a degree of flexibility in choosing (or substituting) components to be integrated in the system.

Thus, in contrast with component coalitions, component federations can be:

- *General* since components can be used in more than one configuration.
- *Amenable to change* since components are not hardwired together.

The key issue here is of course that component federations and the associated standards support the notion of *open systems* which are defined as extensible collections of plug-compatible components and their standards. The open systems approach is advocated by the US DoD through their Technical Architecture for Information Systems Management (TAFIM) programme (Hanratty, June 1995).

4.3 Coalitions or federations?

The issues involved in integration efforts include component integration, the development of a supportive framework as well as identification of standards to allow component replacement. The primary technical differences between the component coalition and component federation approaches are determined by how these issues are addressed.

Component coalitions provide a specific instance of tool-to-tool integration where sharing of services is tailored to the components in the coalition. Plug compatibility is not addressed or encouraged. Component federations on the other hand are supported by a growing number of standards (Kuhn, 1990; Morris *et al.*, December 1991) (as shown in table 1) and can offer greater flexibility as well as support for plug compatibility. Unfortunately, federations are just beginning to develop and in the short term, component coalitions will most likely provide a pragmatic link between the immediate demand for tight integration and the desire for more generalised and adaptable component federation solutions. In the long term, it is expected that federations will be the preferred option and various governmental organisations are actively preparing for it. For instance, the US National Institute of Standards and Technology (NIST) has developed the Applications Portability Profile (APP), a family of related standards that can improve application portability and thus ease the integration of software developed by different organisations and/or based on a variety of hardware platforms (Martin, 1989; Kuhn, 1990). The US DoD has adopted a similar strategy which is already practised in the Defence Information Infrastructure (DII) Common Operating Environment (COE) effort (DIICOE, 1995; Stavridou, 1996).

5 INTEGRATION STRATEGIES

An integration strategy determines the way in which an integrated system is to be assembled. Choosing a suitable strategy depends on the type of system under consideration, in particular whether it is a legacy system or a new procurement.

5.1 Strategies for legacy systems

Hanratty (Hanratty, June 1995) distinguishes three main integration strategies for legacy systems:

- *Grafting* which involves grafting a function on to an existing system in much the same way that a limb is grafted on to a tree. An example of this strategy is the US Navy's AYK-14 computer procurements.
- *Substitution* which implies a more open systems approach in which one component is removed and another replaces it. The assumption is that some standardised interface permits this substitution.
- A *Hybrid approach* involving both grafting and substitution.

5.2 Strategies for new procurements

For brand new developments, such as the US Navy's new attack submarine program, an integration architecture such as a coalition or a federation can be used to hopefully achieve an open system with all the resulting advantages. Therefore, the integration *architecture models* of either coalitions or federations can be used. Of course having a goal architecture for a new system does not remove the need for an integration strategy. Assuming that the system components are available in a suitable form, there are a number of ways of integrating them together:

- *Top down integration* is used by the Cleanroom Software engineering approach (Mills *et al.*, 1987). The strategy is that one starts with the topmost element which is integrated with stubs representing all other components. As integration proceeds, the stubs are gradually replaced by the real components until the entire system has been built.
- *Bottom up integration* starts by assembling together the conceptual building blocks of the system and proceeds by conservatively extending this initial core system.

- *Sandwich integration* is recommended in (Myers, 1976). This is a compromise between the bottom up and top down strategies. Top down and bottom up integration begin at the same time, integrating the system from both the top and the bottom, eventually meeting somewhere in the middle.
- *Big bang integration* involves putting together all the components in one step and then testing them for conformance to the system specification.
- *A combination* of the top down, bottom up and big bang strategies is proposed by (Beizer, 1984). He advocates that small assemblies be integrated by bottom up techniques, control structures be put together by shallow top down and the core system ("backbone") be big banged together.
- *Incremental integration* is described by Deutsch (Deutsch, 1982) and involves introducing components one at a time with the system tested after each component is integrated.
- *Phased integration*, also described by (Deutsch, 1982), requires that components at the next level are integrated all at once and the system be tested before integrating all components at the following level.

It is important to note that all the above approaches are focused on *testing* of the integrated system without any discernible concern with the whole integration lifecycle as discussed in the introductory section of this article. It seems that there is little guidance on broad integration strategies in the literature, the notable exception being the efforts of the SEI group (Brown *et al.*, 1994).

But even the above integration/testing strategies do not meet with universal approval by the software engineering community. An empirical study by (Solheim & Rowland, 1993) contains some interesting comparisons of these approaches. Their results indicate that top down integration strategies are most effective in terms of defect correction, but also exercise the greatest number of components. Bottom up strategies are generally least effective at correcting defects but exercise relatively few components. They also indicate that the top down and big bang strategies produce the most reliable systems, and the bottom up strategies produce the least reliable systems. Systems integrated by the sandwich strategy are moderately reliable by comparison. Their results do not support a preference for either phased or incremental strategies. It is also true that the extent of success of an integration strategy also depends on what is being integrated (e.g. software elements, tools etc.). The literature does not seem to offer any guidance on this point.

6 INTEGRATION MECHANISMS

Regardless of the integration model, architecture and strategy selected, system developers must also decide how to actually carry out the integration in terms of "nuts and bolts". The available choices in terms of potential mechanisms are numerous and include a full range of selections that provide varying degrees of effort and integrated capability. These choices span the breadth of the four integration classes proposed by (Thomas & Nejmeh, 1992), that is *data*, *control*, *presentation* and *process*.

It is important to realise that these are broad categories and particular mechanisms may not fall neatly in any of them. For instance, whenever integration happens in practice then:

- data are shared between the components,
- controlled interaction takes place between the components,
- presentation of information and an interface to operations must exist, and
- the integration takes place in the context of some process, whether explicit or implicit.

For example, to embed a diagram produced by an analysis and design tool into a document being written using a desktop publishing system, it must be possible to:

- embed the text and graphical representation of the diagram in the document flow,
- synchronise the transfer and sharing of the design,
- present an interface that allows the embedding to take place, and
- perform the embedding in a manner consistent with the organisation's process and documentation standards.

Understanding these relationships is particularly critical in the case of *data* and *control* integration mechanisms since these are by far the most prominent. At the most basic level, data and control are different: data are an artifact over which control is exerted. However data and control have important interactions without which neither is viable. (Brown *et al.*, 1994) conclude that the essential difference between mechanisms that integrate data and control lies in the *intention* and *emphasis* of the mechanism. That is:

- Data integration mechanisms emphasise storage and persistence, common data structures and shared understanding of these structures.
- Control integration mechanisms emphasise a common communication approach between components and a shared understanding of the events that initiate interaction.

In fact, as both control and data integration mechanisms become more mature, it is likely that the distinctions between them will become increasingly blurred. We already see this tendency in some object oriented systems ((Coleman *et al.*, 1994)), where the distinctions between data and control mechanisms that interact with the data are closely interwoven. In a typical object oriented system, objects are manipulated through a public interface (called a *method*) by sending messages. The message sent and the type of the object identify the method to be invoked. Using this approach, one component can invoke an operation by sending a message to another component that causes the second component to manipulate or share data, provided that the second component makes such an interface (method) available. Clearly, such component interaction shows characteristics of both data integration (i.e. a shared understanding of the data manipulated by the components) and control integration (i.e. a shared understanding of the mechanisms whereby one component can invoke another).

In the subsections that follow, we discuss the characteristics of data and control integration mechanisms as well as specific available mechanisms and their advantages and disadvantages.

6.1 Data Integration Mechanisms

Data integration mechanisms define the way that data is shared between system components. When components share data, two issues must be addressed. First, an agreement needs to be established between components concerning what data are stored, where and how they are accessed, that is *data persistence*. Second, the components must share an understanding of the meaning of data, that is *data semantics*. These two concepts largely provide the backdrop for two classes of data integration mechanisms, each emphasising data persistence or data semantics. Below we discuss these two major data integration strategies and present some actual examples of integrating mechanisms currently available.

6.1.1 Data persistence

There are two strategies supporting storage and sharing of persistent data (Brown *et al.*, 1994):

- *Import/Export* where components maintain separate databases for their unique data and sharing is accomplished by translating the internal data representation to an external form suitable for other components. Typically, data sharing occurs only in the form of *data extraction* (i.e. export only). Although this strategy is viable when a small number of components are involved, maintaining *consistency* between related data becomes unmanageable with larger volumes of data and number of components.

- *Common data store* which includes proprietary data stores in a tightly integrated system from a single vendor, traditional databases used to accumulate data from multiple components and common object bases developed explicitly to support the needs of system integration. The latter is the most general solution and a number of Object Management Systems (OMSs) have been developed recently, although they do not seem to be the preferred solution of integrated systems users (Rader *et al.*, July 1993), presumably because of issues such as performance, costs and lack of standards.

Table 2 summarises some integration characteristics of the import/export and common data store approaches. There does not seem to be a general answer as to which approach is better. This should be seen as a *process integration* issue (as defined by (Wasserman, September 1989)) reflecting the needs of a particular organisation.

6.1.2 Data semantics

The meaning of shared data can be defined in one of two ways:

- *A common schema* which identifies the types of items represented by the data, characteristics of these items and the types of relationships between items. Shared schemata are often referred to as *information models* since they focus on the meaning of data to be shared.

- *Common data formats* which represent either an agreement to store all data in a specified format at all times or to extract a subset of the data and make it available at specific times. The focus here is conveying semantic information based on the structure of the data rather than the meaning of the data to a human observer.

A problem both these approaches share is control of the evolution of the common schema or data format since vendor compromise and cooperation is required.

6.1.3 Currently available data integration mechanisms

Two well known common data stores are:

- *The Portable Common Tool Environment* (PCTE) is an interface specification of facilities that can be used by software tools which was standardised by the European Computer Manufacturers Association (ECMA) (PCTE, 1990). It exists as an abstract specification and in language bindings to C, Ada and C++. A subset of the full specification, known as PCTE 1.5 has been in existence as a product for several years. Components may be integrated into PCTE using foreign tool encapsulation, file level integration and structured integration.

- *A Tool Integration Standard* (ATIS) is a tool integration approach providing a set of standard interfaces to enable CASE tool integration. ATIS was developed by Atherton Technology and implementations are available from Digital and Atherton (CASE-ISBD, 1990). Component integration occurs either by encapsulation or by porting. In encapsulation, tool data are stored in the repository and exported and reimported prior to and after tool use. For porting, the source code of the tool is modified to make full use of ATIS services to create and modify components, as well as to perform a range of other activities.

The following two common data schemata are well known:

- *The Information Resource Dictionary System* (IRDS) is a data dictionary standard that is intended to support a consistent format for describing the structure and relationships of data within a database system (IRDS, 1988). For this reason, IRDS is considered as a standard for specifying *meta-data*.

- *The IBM Information model* is intended to support application development in the domain of information systems.

The most widely known common data format is *the CASE Data Interchange Format* (EIA CDIF) and it is an effort by the Electronics Industry Association (EIA) to define the specific formats for data that are transferred between CASE tools (Parker, May 1992). Although conceived as a mechanism to support import/export style data integration, CDIF is currently considered as a mechanism for transferring data between different implementations of data repositories.

6.2 Control Integration Mechanisms

Although data sharing can provide a high level of control and coordination between system components, it also imposes a significant overhead on the system, both because of poor performance of existing database mechanisms when used this way, and because of the necessary agreement required between the components to define a common data schema. An alternative integration approach is focused on *control* rather than shared data. In the *control integration approach*, a system is viewed as a collection of services provided by different components. Actions carried out by one component are announced to others via control signals. The receiving components can then decide if they need to respond in some way. This form of communication is known as *message passing* (a visualisation of which is shown in figure 10). In the remainder of this subsection, we briefly describe message passing and review a number of existing systems.

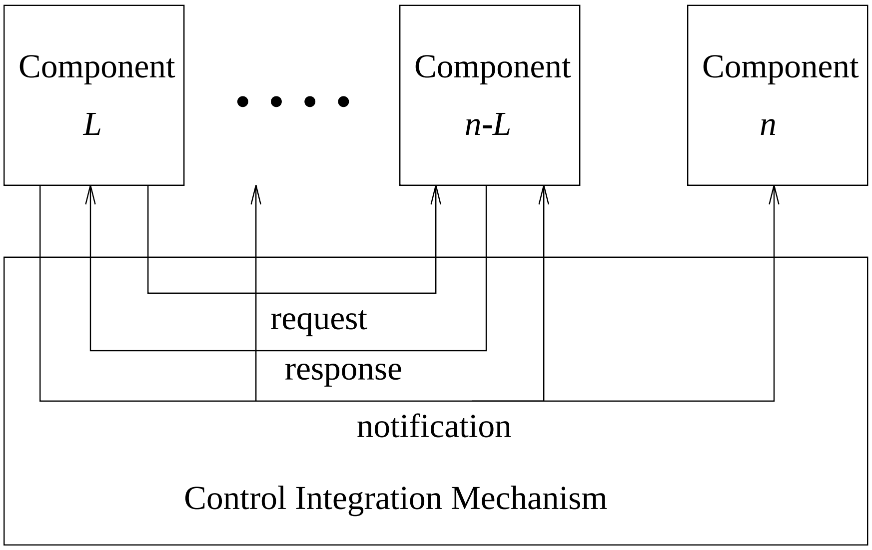

Figure 10: Integration through message passing

6.2.1 The message passing approach

In the message passing paradigm, components communicate by passing messages informing other components of their actions and requesting services from the other components. This requires a *mechanism* enabling the communication as well as a *protocol* defining the acceptable communication patterns amongst components.

The control integration mechanism shown in figure 10 is responsible for message distribution and is termed the *message server* or *message broker*. In contrast with the "point-to-point" architecture in section 4, components never communicate directly with each other. Instead they send messages to the server who is then responsible for selectively forwarding messages to appropriate components and for initiating execution of a component to handle the current message if necessary.

6.2.2 Examples of the message passing approach

A number of message passing frameworks have been proposed and built.

- *Friendly Integrated Environment for Learning and Development* (FIELD) is the origin of much of the work on the message passing approach with an initial implementation available as early as 1987 (Reiss, 1990). The two basic components of FIELD are a consistent graphical user interface as a front-end to all components and a simple message server *Msg*, implemented as a separate UNIX process, communicating with other processes via sockets.

- The *SoftBench* environment is a Hewlett-Packard product developed to provide an architecture for integrating CASE tools (Cagan, 1990). SoftBench provides tool communication and distributed support as well as user interface management. At the core of SoftBench is the *Broadcast Message Server* (BMS) which is similar to the FIELD server with the addition of a tool protocol. New components can be incorporated in SoftBench using HP's *Encapsulator* tool to develop an envelope within which the component can execute and communicate.

- *ToolTalk* from Sun Microsystems (Frankel, 1991) is a distributed message passing system similar to SoftBench (the most noticeable difference being the object-oriented emphasis used to describe the former). The system is implemented on top of SunSoft's Remote Procedure Call mechanism and runs on SunOS 4.1.1 or later. The message server is a special process, *ttsession*, per user session. Components interact with the ToolTalk service by calling functions defined in the ToolTalk *application programming interface* (API).

- The *Common Object Request Broker Architecture* (CORBA) is intended to provide a common interface that allows applications to exchange information in an object-oriented environment (Otte *et al.*, 1996). CORBA allows transparent access to applications operating on multiple computing platforms by requiring that each platform should support a compliant *Object Request Broker* (ORB). Unlike the other systems mentioned here, CORBA is not an implementation of a message passing system. Rather, it is a specification for communications between clients requesting a service and objects providing a service. CORBA is endorsed by the *Object Management Group* (OMG), an organisation that includes major computer manufacturers as well as software providers. Implementations of CORBA and tools compliant with it are now commercially available. CORBA is designed to operate in heterogeneous, distributed environments unlike the rest of the messaging systems which are primarily meant to operate with tools executing on a single platform.

- The *Common Open Software Environment* (COSE) is an attempt to provide an API for UNIX systems (COSE, 1994). The scope of the COSE effort includes work on a *common desktop environment* (CDE), graphics, multimedia, networking, object technology, systems management, naming conventions and data management. CDE's message passing is based on ToolTalk and HP's Encapsulator. The COSE consortium involves all major UNIX system and software vendors and has the potential of establishing consensus in the message passing system market place.

- *Object Linking and Embedding* (OLE) is Microsoft's offering and provides users with the ability to embed items created by one application within data produced by another one (Endrijonas, 1994). OLE provides a client-server model of computing and focuses on communication and data sharing between applications on a single desktop. It comes bundled with Windows NT and has, unsurprisingly, spurred competitors to band together to produce an alternative to Microsoft's technology. This effort has primarily focused on Apple's OpenDoc system which provides services similar to OLE in a distributed environment (Orfali *et al.*, 1995).

While a number of open issues remain, there is evidence supporting the message passing approach as the basis for systems integration. It also provides a good basis for further exploration of a number of issues including tool integration, process versus data versus user interface integration (Brown *et al.*, 1994).

7 A method for assembling integrated systems

The multilevel model of integration proposed by (Brown *et al.*, 1994) seems the most well developed in that it separates *mechanistic* issues (how the interrelationships amongst components are implemented) from *semantic* issues (what services are provided and how those services are related). In addition, this model takes into account the context in which the system will operate by facilitating the link between system services and user processes.

More importantly, this multilevel model facilitates the *view of integration as a design* rather than a V&V activity. Combining the idea of integration as part of design with Brown's model, yields a *method* for assembling an integrated system (Brown *et al.*, 1994):

- **Analyse and record the processes** that are to be supported as well as the constraints on carrying out these processes, such as for example mission characteristics and tactics or hardware capabilities.
- **Describe what services** the system must provide and how these will interact in order to support the identified processes. For example a radar system may provide the following services:
  - radar input
  - aircraft tracking
  - controller interface
- **Choose or develop the system components** necessary to realise the required services. Using the previous example, the aircraft tracking service might require components carrying out the following tasks:
  - get aircraft position
  - identify known aircraft
  - predict next position
  - update display

While these steps are described in a sequential fashion here, in most instances there will be significant feedback and iteration between them. For example, changes to the processes may be made to accommodate the components available, or component selections may be amended to accommodate new requirements. However, it is important to note that such changes take place within a context that has documented why such changes are necessary, provides a basis on which such decisions can be rationalised and maintains a history of how the system architecture has evolved in its current form.

The Software Engineering Institute (SEI) carried out a number of experiments involving the integration of CASE tools, including experiments in environment integration and replacing the message service in a CASE integration framework. Their findings support the utility of this approach to assembling integrated systems (Brown *et al.*, 1994).

8 Discussion

Integration is a complex task involving a number of trade-offs. Like other software qualities such as robustness and portability, answers about "good" integration are neither obvious nor simple; integration needs to be planned for and designed for.

For any given integration effort, we need to identify the goals of the project in order to discover the ways in which these goals become articulated within a defined process. Implicit and explicit organisational and project goals are manifested in the process that is used for developing software, together with the specific needs for integration. Different goal and process combinations will have different implications for integration purposes. These differences require different integration solutions.

The solution selected by an organisation must be tempered with the realities of the multitude of constraints guiding the successful adoption of that solution. Factors such as available budget, receptiveness to new technology and migration strategy for current tools and system will all need to be considered. The three level model of use services, integration mechanisms and process can be used as a reference point for evaluating alternative decisions and trade-offs.

In summary:

- The multilevel model of integration enables the evaluation of alternative integration decisions.
- Integration is best viewed as a design rather than a V&V activity.
- Component federations are the preferred integration architectures.
- Control integration mechanisms support generic integration solutions.
- The adoption of a framework of low level integration standards is prerequisite for building component federations.

Acknowledgements

This article and the work herein, would not have been possible without the inspiration and support of Kevin Geary. The manuscript has also benefited from the comments of the two anonymous referees.

References

Beizer, B. 1984. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold.

Brown, A.W., Carney, D.J., Morris, E.J., Smith, D.B., & Zarrella, P.F. 1994. *Principles of CASE Tool Integration.* Oxford University Press.

Cagan, M.R. 1990. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, **41**(3).

CASE-ISBD. 1990. *CASE Interface Services Base Document.* Nashua, NH, USA: Digital Equipment Corporation.

Coleman, D., *et al.* 1994. *Object-Oriented Development - The FUSION Method.* Prentice Hall.

COSE. 1994. COSE and OSF: making the new standards stick. *Open Systems Economics Letters*, **1**(6), 6–7.

Deutsch, M.S. 1982. *Software Verification and Validation: Realistic Project Approaches.* Englewood Cliffs, NJ: Prentice Hall.

DIICOE. 1995. *Defense Information Infrastructure (DII) Common Operating Environment (COE) Integration and Runtime Specification (I & RTS), Preliminary, Version 2.0.* Joint Interoperability and Engineering Organisation, Defense Information Systems Agency (DISA).

Endrijonas, J. 1994. OLE: not a cheer but a step to integration. *Managing Automation*, **9**(8), 30–32.

Fernström, C., Närfelt, K-H., & Ohlsson, L. 1992. Software Factory Principles, Architectures and Experiments. *IEEE Software*, March, 36–44.

Frankel, R. 1991. *Introduction to the ToolTalk Service.* Mountain View, CA, USA: Sun Microsystems Inc.

Hanratty, J. June 1995. Panel presentation on Open Systems, Standards and COTS & Integrating Large Systems. *In:* Brown, A.W., Carney, D.J., & McFalls, M.D. (eds), *Proceedings of the SEI/MCC Symposium on the Use of COTS in Systems Integration.* SEI Special Report CMU/SEI-95-SR-007.

IRDS. 1988. *A Technical Overview of the Information Resource Dictionary System.* 2nd edn. NBSIR 85-3164. Gaithersburg, MD, USA: The National Bureau of Standards.

Kuhn, D.R. 1990. On the Effective Use of Software Standards in Systems Integration. *Pages 455–461 of:* Ng, P.A., Ramamoorthy, C.V., Seifert, L.C., & Yeh, R.T. (eds), *Proceedings of 1st International Conference on Systems Integration.* IEEE Computer Society Press.

Martin, R.J. 1989. The Standards Set for Portability. *Datamation*, May 15.

Marty, R. 1988. Software Integration. *Computer Standards and Interfaces*, **8**, 9–14.

Mills, H.D., Dyer, M., & Linger, R.C. 1987. Cleanroom Software Engineering. *IEEE Software*, **4**(55), 19–25.

Morris, E., Feiler, P., & Smith, D. December 1991. *Case Studies in Environment Integration.* Tech. rept. CMU/SEI-91-TR-13. SEI, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

Myers, G.J. 1976. *Software Reliability: Principles and Practices.* New York: Wiley.

Nilsson, E.G., Nordhagen, E.K., & Oftedal, G. 1990. Aspects of Systems Integration. *Pages 434–443 of:* Ng, P.A., Ramamoorthy, C.V., Seifert, L.C., & Yeh, R.T. (eds), *Proceedings of 1st International Conference on Systems Integration*. IEEE Computer Society Press.

Orfali, R., Harkey, D., & Edwards, J. 1995. Client/Server Components: CORBA meets OpenDoc. *Object Magazine*, **5**(May).

Otte, R., Patrick, P., & Roy, M. 1996. *Understanding CORBA: the Common Object Request Broker Architecture*. Prentice-Hall.

Parker, B. May 1992. Introducing EIA-CDIF: The CASE Data Interchange Format Standard. *In: Procs of 2nd Symposium on the Assessment of Quality Software Development Tools*. New Orleans, LA, USA: IEEE Computer Society Press.

PCTE. 1990. *Portable Common Tool Environment (PCTE) Abstract Specification*. ECMA-149. European Computer Manufacturers' Association (ECMA).

Rader, J.A., Morris, E.J., & Brown, A.W. July 1993. An Investigation into the State of the Practice of CASE Tool Integration. *In: Proceedings of SEE93*. IEEE Computer Society Press.

Ramamoorthy, C.V., Chandra, C., Kim, H.G., Shim, Y.C., & Vij, V. June 1992. Systems Integration: Problems and Approaches. *Pages 522–529 of: Proceedings of 2nd International Conference on Systems Integration, Morristown, New Jersey*. IEEE Computer Society Press.

Reiss, S.P. 1990. Connecting Tools Using Message Passing in the FIELD Environment. *IEEE Software*, June, 57–99.

Solheim, J.A., & Rowland, J.H. 1993. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Transactions on Software Engineering*, **19**(10), 941–949.

Stavridou, V. 1996 (March). *A Study of Integration in Software Intensive Systems*. Internal Report, UK Ministry of Defence, MOD, Procurement Executive, SM836, Ash1, Abbeywood #93, PO Box 702, Bristol BS12 7DU.

Thomas, I., & Nejmeh, B.A. 1992. Definitions of Tool Integration for Environments. *IEEE Software*, March, 29–35.

Wasserman, A.I. September 1989. Tool Integration in Software Engineering Environments. *Pages 137–149 of:* Long, F. (ed), *Software Engineering Environments, volume LNCS 467.* Proceedings of International Workshop on Environments, Chinon, France.

Function	**Element**
Frameworks	Portable Common Tools Environment (PCTE) A Tool Integration Standard (ATIS)
Operating systems	(Extended) POSIX
User interface	X Windows Motif Open Look
Database management	SQL IRDS
Data interchange	Postscript CDIF ASCII CGM IGES & PDES/STEP(ISO 10303) SGML ODA, ODIF
Network services	OSI TFA (NFS)
Programming services	C FORTRAN COBOL Ada Pascal

Table 1: Identified low level integration standards

Data Integration characteristic	**Import/Export**	**Common Data Store**
Interoperability	Subject to vendor agreements	Common scheme and data interfaces
Degree of data duplication	No explicit approach	Minimised
Data consistency	Limited for data extraction	Assured
Extensibility and flexibility	Published data interfaces	Schema modifications and multiple views
Support for multiple users and Security	Varies from component to component	Centralised policy and mechanism
Transaction integrity	Mostly manual	Mostly automatic with recovery
Configuration management	On per-tool basis	Centralised policy and mechanism

Table 2: Comparison of Import/Export and Common Data Store