Software Architecture in Industrial Applications

Dilip Soni, Robert L. Nord, and Christine Hofmeister

{dsoni, rnord, chofmeister}@scr.siemens.com
Siemens Corporate Research Inc.
755 College Road East, Princeton, NJ 08540

Abstract

To help us identify and focus on pragmatic and concrete issues related to the role of software architecture in large systems, we conducted a survey of a variety of software systems used in industrial applications. Our premise, which guided the examination of these systems, was that software architecture is concerned with capturing the structures of a system and the relationships among the elements both within and between structures. The structures we found fell into several broad categories: conceptual architecture, module interconnection architecture, code architecture, and execution architecture. These categories address different engineering concerns. The separation of such concerns, combined with specialized implementation techniques, decreased the complexity of implementation, and improved reuse and reconfiguration.

1. Introduction

Software architectures describe how a system is decomposed into components, how these components are interconnected, and how they communicate and interact with each other. When poorly understood, these aspects of design are major sources of errors.

The research area of software architecture is an emerging one with little agreement over the definition of architecture. The only consensus seems to be that architecture is related to the structure of a system and the interactions among its components [14, 25]. Some of the work focuses on building theory and models of architecture [2, 31], with an emphasis on taxonomy, description languages, and verification of architectural properties. Several efforts have been made to generate code based on parts of architecture [13, 27]. In general, the literature relies heavily on informal examples and anecdotes, and most of this work has not been proven to be scalable for large systems. Also missing from prior work is the focus on the pragmatic role of architecture in software development activities [10, 33].

ICSE '95, Seattle, Washington USA
© 1995 ACM 0-89791-708-1/95/0004...$3.50

In order to examine the pragmatic and concrete issues related to the role of architecture in the design and development of large systems, we conducted a survey of a variety of industrial software systems to understand architecture as it is practiced in the real world. The systems we surveyed included several image and signal processing systems, a real-time operating system, communication systems, and instrumentation and control systems [9, 13, 17]. These systems are listed in Table 1. Our premise, which guided the examination of these systems, was that software architecture is concerned with capturing the structures of a system and the relationships among the elements both within and between structures. Following this premise, we interviewed system architects[1] and designers, and examined design documents and code to look for important system structures and their use. Details of the survey process are described in Appendix A.

We observed that most of the systems made a distinction between architecture for the product-specific software and architecture for the platform software. Intuitively, the platform software provided a virtual machine or an infrastructure for the product-specific software.

For both the application software and the platform software, different structures were used at different stages of the development process. The structures we found fell into several broad categories: conceptual architecture, module interconnection architecture, execution architecture, and code architecture. Within each category, the structures describe the system from a different perspective:

- The **conceptual architecture** describes the system in terms of its major design elements and the relationships among them.
- The **module interconnection architecture**[2] encompasses two orthogonal structures: functional decomposition and layers.
- The **execution architecture** describes the dynamic structure of a system.

1. By architects, we mean those who designed the software architecture.
2. We will refer to the module interconnection architecture as module architecture for brevity and ease of reading.

System	Application Domain	Size	Important System Characteristics
A	user interface	small	window management
B*	signal processing	medium	monitoring, real-time
C	image and signal processing	medium	high throughput
D*	signal processing	medium	monitoring, real-time, safety-critical
E*	image and signal processing	very large	high throughput
F*	computing environment	medium	management of distributed information
G*	instrumentation and control	large	fault tolerance, multi-processing, safety-critical
H	instrumentation and control	large	multi-processing
I	operating system	large	real-time
J	communication	very large	multi-processing
K*	communication	very large	distributed, heterogenous, multi-processing

small: fewer than 100 KLOC; medium: 100-500 KLOC; large: 500-1 MLOC; very large: more than 1 MLOC

* These systems were selected for detailed study.

Table 1: Summary of the Surveyed Systems

- The **code architecture** describes how the source code, binaries, and libraries are organized in the development environment.

As shown in Table 2, not all of the systems have explicitly described architectures for each category: in several cases, the architectures are implicit in text, code, or one of the other architecture descriptions. We also found architectures that were a hybrid or that mirrored each other. When they did make a distinction among these architectures, the separation of engineering concerns, combined with specialized implementation techniques, decreased the complexity of implementation, and improved reuse and reconfiguration.

System	Conceptual	Module	Execution	Code
A				+
B		+[c]	+	+[b]
C		+[c]	+	+
D		+	+	+
E		+[c]	+	+
F	+	[d]	+	+
G	+	+	+	+[a,b]
H	+	+	+	+[a,b]
I			+	+
J		+		+[b]
K		+	+	+

a. mirrors conceptual, b. mirrors module,
c. mirrors execution, d. combined with execution

Table 2: Explicitly Described Architectures in the Surveyed Systems

We also observed that software architecture played an important role throughout the development process, and in achieving desired levels of non-functional system properties. However, the focus of this paper is to describe the software architectures we found. We will examine the role of software architecture in the software development process in more detail in a subsequent paper.

For this paper we describe portions of three of the systems surveyed: one was chosen as a prototypical example, and the others were chosen for the richness of the architectures they used. We have removed the proprietary details from these examples, and changed the names of the systems to remove any connection to actual systems.

We first give a prototypical example of one of these systems, System B, in Section 2. This system provides a good overview of the kinds of structures found in the software architectures, and of typical ways the architecture was used.

While System B's software architecture is a good introduction to the state of the practice, it does not include the full variety of the architecture we found. Section 3 describes portions of two systems that are at the leading edge in their use of software architecture. These systems demonstrate the advantages to be gained from the separation of concerns as realized in different structures, and how to avoid the problems that arise when the structures are combined.

The final section summarizes the description techniques that were used to describe the software architectures.

2. Four Distinct Software Architectures

Before giving a detailed description of the four kinds of software architectures we found, we first put them in context by relating them to the historical development of software and offering examples of why they are needed and how they are used (Table 3).

Software Architecture	Examples of Uses	Examples of Influencing Factors
Code Architecture	configuration management, system building, OEM pricing	programming language, development tools and environment, external subsystems (e.g., X windows)
Module Architecture	management and control of module interfaces, change impact analysis, consistency checking of interface constraints, configuration management	enabling software technologies, organization structure, design principles
Execution Architecture	performance and schedulability analysis, static and dynamic configuration of the system, porting systems to different execution environments	hardware architecture, run-time environment performance criteria, communication mechanisms
Conceptual Architecture	design using domain-specific components and connectors, performance estimation, safety and reliability analysis, understanding the static and dynamic configurability of the system	application domain, abstract software paradigms, design methods

Table 3: Uses and Influencing Factors of Software Architectures

Initially, only the source code existed. Then interfaces were introduced for type checking by compilers or other tools. As software became larger it became useful to divide source code into files and introduce information hiding techniques. "Make"[12], configuration management [11], and systems building techniques [22] were introduced along with a flexible product structure to describe and manipulate the file level organization of source code. This information belongs in the code architecture.

As systems became even larger and multiple programmers worked on a project, abstraction mechanisms, modules, and module interconnection structures were introduced. The importance of module interconnection structures for programming-in-the-large was first recognized by DeRemer and Kron [7]. Several variants of module interconnection languages [26] have been defined that support various features of programming-in-the-large (e.g., Intercol [35], PIC [37], and NuMIL [24]). This corresponds to the module architecture in our categorization.

When systems became distributed, programmers needed to consider dynamic structure and communication, coordination, and synchronization. Functional components needed to be allocated within the dynamic structure [16]. Recently a number of interconnection languages have been introduced that address the issue of allocating components in a distributed environment. These include Conic [21], Durra [3], Enterprise [29], Polylith [27], and UNAS [28]. In our software architecture categories, these issues are addressed in the execution architecture.

Now research is progressing to higher abstraction levels. Software engineers would like to be able to compose software systems using communicating objects [1, 30] or assemblies of components and connectors [4,6,8,23,31]. We place these concerns in the conceptual architecture.

The remainder of this section describes System B, which is a good example of typical practices regarding the description and use of software architectures. System B collects data from remote sensor groups for a number of sites and generates reports of the monitored data. Safety and real-time performance are some of the important system characteristics for System B.

2.1 Conceptual Architecture

The conceptual architecture describes the system in terms of its major design elements and the relationships among them. This is a very high-level structure of the system, using design elements and relationships specific to the domain. Conceptual architectures are independent of implementation decisions and emphasize interaction protocols between design elements.

In most of the systems we surveyed, conceptual architectures were implicit and embedded within the documentation of other structures. Although the conceptual architecture of System B is not documented explicitly, the system architect drew informal diagrams depicting major system components when describing the system to us. We have derived a component-connector model of its conceptual architecture based on these discussions with the system architect and through extensive studies of the documentation and code. This model is informally described in Figure 1; the rounded boxes represent functional components while the rectangular boxes represent

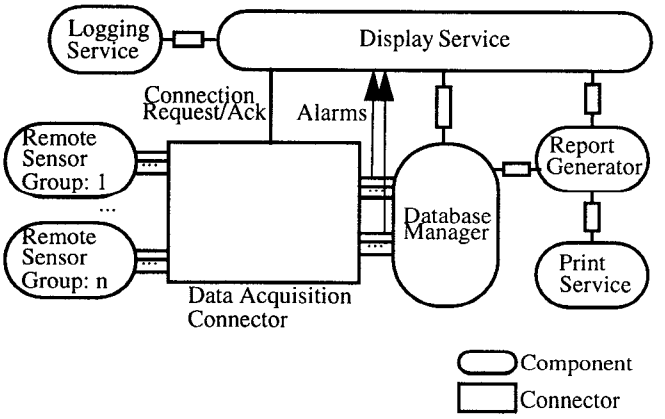

Figure 1: Conceptual Architecture of System B

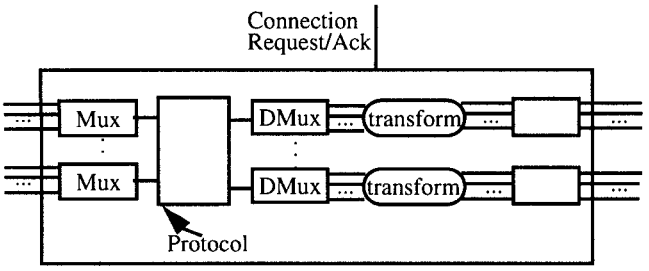

Figure 2: Decomposition of the Data Acquisition Connector in System B

connectors (i.e. interfaces and protocols) between components.

Both components and connectors can be further decomposed in terms of more primitive components and connectors. Figure 2 demonstrates how the data acquisition connector in Figure 1 can be further decomposed in terms of a connector representing a data acquisition protocol and other components. This protocol is described using state transition tables for the client and servers. Typical message flows are described using extended sequence charts. Details of message formats complete the description of the protocol.

2.2 Module Architecture

The module architecture supports programming-in-the-large [7]. It encompasses two orthogonal structures: functional decomposition and layers. Functional decomposition of a system captures the way the system is logically decomposed into subsystems, modules, and abstract program units. It captures the components' interrelationships in terms of exported and imported interfaces. Layers reflect design decisions based on allowable import/export relations and interfacing constraints. They reduce and isolate external (e.g., hardware and operating system) and internal dependencies, and facilitate bottom-up building and testing of various subsystems.

Unlike the conceptual architecture, the module architecture reflects implementation decisions. These implementation decisions are generally independent of any programming language. Thus this architecture can also be thought of as the ideal implementation structure.

In System B, as in most of the systems we studied, the module architecture is the most developed aspect of the system. The application software of System B is decomposed into a collection of modules for the major functions of the system (Figure 3). The platform software of the system is decomposed into system modules such as the real-time operating system, file management, and the device drivers. Details of the interfaces between pairs of modules are described in the architecture specification.

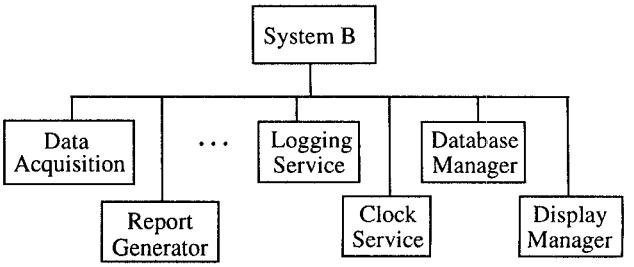

Figure 3: Functional Decomposition for Application Software of System B

The layer structure of System B shows how the application modules are assigned to the top two layers (Figure 4). The layers in the architecture represent allowable interfaces among the application modules as well as the modules implementing the platform software. Modules within a layer can communicate with each other. Modules in different layers can communicate with each other only if their respective layers are adjacent.

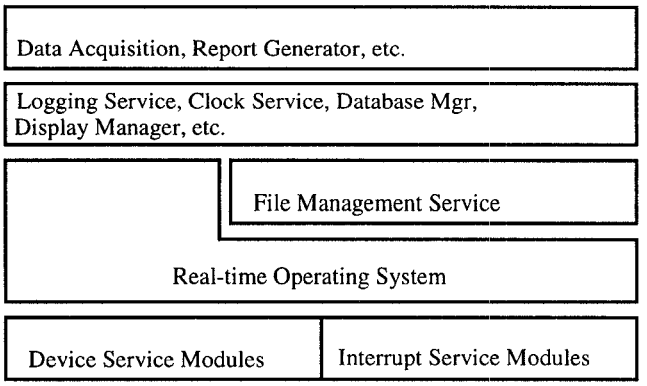

Figure 4: Layer Structure of System B

The dominant relationship between the conceptual and the module architectures is *implemented_by*: a component or connector in the conceptual architecture may be implemented by one or more modules in the module architecture. The implementation of a conceptual connector is usually distributed across several modules, and may be included in the implementations of components connected to it. For example, implementation of the data acquisition connector is distributed across both the remote sensor groups and the data acquisition module. A conceptual connector also imposes the interface requirements between its implementation and the implementations of its connecting components.

2.3 Execution Architecture

The execution architecture is used to describe the dynamic structure of a system in terms of its run-time elements (e.g., operating system tasks, processes, address spaces), communication mechanisms, assignment of func-

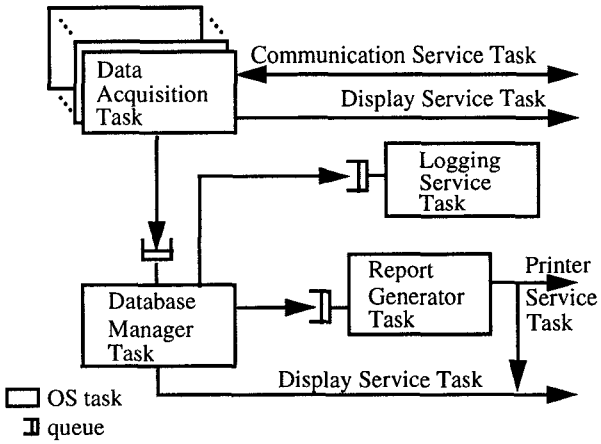

Figure 5: Execution Architecture of System B

tionality to run-time elements, and resource allocation. It also describes design decisions related to location, migration, and replication of a system's run-time elements.

The driving forces behind the structuring decisions here are performance, distribution requirements, and the run-time environment, which includes the underlying hardware and software platform. Thus, the execution architecture is likely to change more often because the underlying technology of the hardware/software platform changes frequently. Execution architectures are used for performance analysis, monitoring, and tuning as well as for debugging and service maintenance. The execution architecture is sometimes trivial, as for example when the system is a single process.

The execution architecture is a prominent part of the software architecture of all of the surveyed systems. System B illustrates a simple but typical example of an execution architecture (Figure 5). Each application module from the module architecture is assigned to its own operating system task, and sometimes a module is replicated by having multiple instantiations of its associated task. For example, the Data Acquisition task is replicated once for each remote sensor group. The real-time operating system provides the queues, semaphores, and mailboxes that are used for interprocess communication between tasks. Tasks, acting as producers or consumers, use both synchronous and asynchronous communication to exchange data. Figure 5 is derived from the detailed diagram of the execution architecture. More information about the interactions among the tasks is shown using control and data flow diagrams. The paths are numbered to reference accompanying text that describes the order of communication among the tasks. These diagrams are similar to object interaction diagrams used in object-oriented analysis [18].

The relationship between the conceptual and execution architectures is two fold. First, each conceptual component or a connector (or its implementation) can be related to a run-time element in the execution architecture using the relation *assigned_to*. Second, the communication mechanisms selected in the execution architecture constrain the

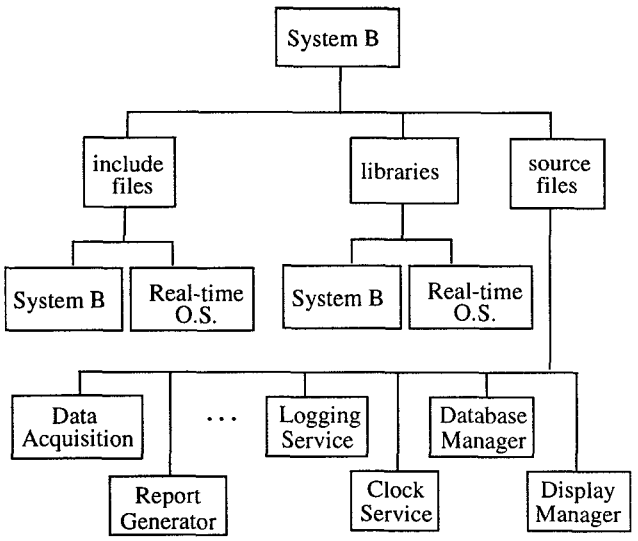

Figure 6: Code Architecture of System B

implementation of conceptual components and connectors. In System B, for example, the selection of the operating system and communication mechanisms constrains how the data acquisition connector is implemented.

The relationship between module and execution architectures is similar. Modules in the module architecture are *assigned_to* run-time elements in the execution architecture. In addition, each of the run-time elements in the execution architecture is *implemented_by* specific modules in the module architecture.

2.4 Code Architecture

The code architecture is used to organize the source code into language level modules, directories, files, and libraries. This architecture is influenced by the choice of the programming language, the development tools and environment (e.g., configuration management), and the structure of the project (e.g., inter-corporation) and the organization. Although this architecture is explicit in all the surveyed systems, it is generally not described as part of the software architecture. We found that the major use of this architecture is to facilitate system building, system installation, and configuration management. Other uses are to minimize dependencies (e.g., compilation) among subprojects and to enforce import/export constraints specified in the module architecture.

For System B, the organization of the source code mirrors the functional decomposition of the module architecture of the application. The code architecture (Figure 6) consists of directories for include files (defining interfaces), common library functions and source code. The include files and libraries for the application and system software are separated.

The relationship between the module and code architectures is simply characterized by *implemented_by*. Each abstract program unit in the module architecture is *implemented_by* concrete program units in the code architecture.

There is also a relationship between run-time elements in the execution architecture and elements such as executables and configuration files in the code architecture.

2.5 Summary

In this section, we have described the four categories of software architectures found in the surveyed systems. We have also described the relationships among the conceptual, module, execution, and code architectures; these relationships are summarized in Figure 7.

The four kinds of architectures address different though inter-related engineering concerns:

- Each contains the results of engineering decisions taken at different times in the development process (e.g., design time, implementation time, build time, or run time).
- Each is influenced by different engineering and organizational factors such as clarity, reducing dependencies among programming tasks, evolution, or performance.
- Each is developed and used for different purposes by different teams (e.g., system architects, programmers, system builders, or service personnel).
- The degree to which each architecture reflects implementation decisions varies.
- The degree to which each architecture is expected to change varies. For example, the execution architecture is likely to change more often than the conceptual architecture.

Each architecture category can be characterized using a set of components, their interconnections, and the criteria for evaluating them. Table 4 gives examples of typical components and interconnections for architectures in each of the above categories. The challenge, as we see it, is to precisely describe these architectures, separate their descriptions to manage complexity, establish correspondences between the descriptions, and facilitate analysis and manipulation of the descriptions [32].

3. Separation of Concerns

It has long been recognized by hardware designers that separation of concerns has powerful advantages when developing large, complex systems. They have reached agreements about separating design artifacts into levels (e.g., architecture, logic design, physical layouts and masks) describing different concerns, and about the precise description of these design artifacts [5, 19].

Separation of concerns is also used to advantage in reflective systems [20]. In the Apertos operating system [38] for example, an object is defined independently from its execution environment, in order to facilitate object migration. A similar approach is used to deal with the complexity of concurrent systems [1]. Concurrency abstractions simplify the task of programming by separating design concerns such as locality and synchronization. The separation of design concerns is accomplished by using abstractions to specify coordination patterns, and by separating functionality from resource management.

Separation of engineering concerns facilitated the three architecture-related objectives that are common among the systems we studied:

- exploitation of commonalities in product families,
- faster turn around time for enhancement of application functionality, and
- evolution due to evolving hardware and software technologies.

Our observation is that the use of different kinds of architectures arose in part to address these common objectives. Many of the reported successes of the surveyed sys-

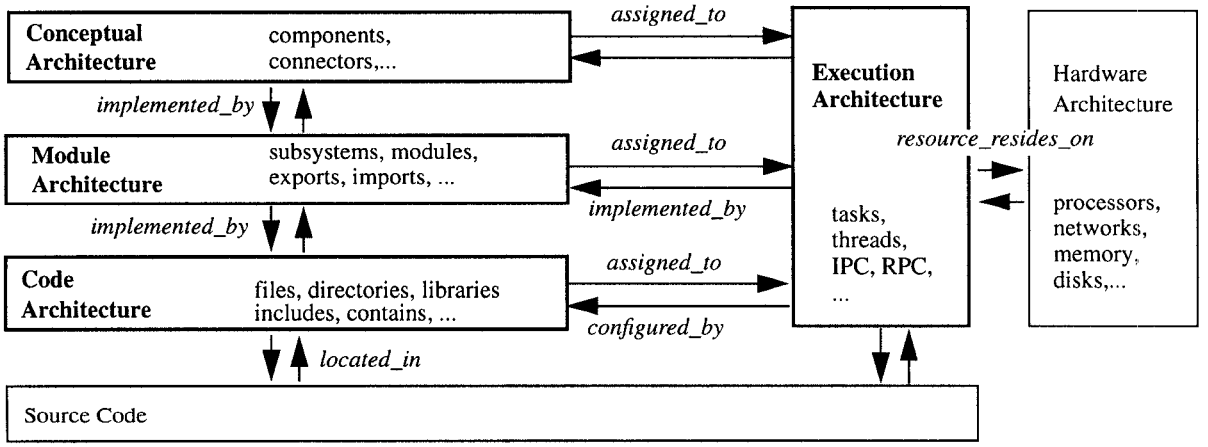

Figure 7: Relationships among the Software Architectures

Software Architecture	Components	Interconnections	Engineering Criteria
Conceptual Architecture	domain-specific components	domain-specific interconnections	constraints on cycles, requirements on throughput
Module Architecture	systems, subsystems, modules, layers	component_of, import_from, export_to	functional decomposition methods, criteria such as information hiding and abstraction, when to use multiple interfaces, constraints imposed by layering strategy
Execution Architecture	processes, tasks, threads, clients, servers, buffers, message queues	inter-process communication, remote procedure call	criteria for priority assignment, constraints imposed by run-time environment
Code Architecture	files, directories, linker libraries, packages, program libraries	member_of, includes, contains, linked_with, compiled_into, with clause, use clause	compilation and build time, criteria related to project management and configuration management tools and the development environment

Table 4: Characterizations of the Software Architectures

tems resulted from allowing the different architectures to develop independently while maintaining the relationship between them. Similarly, some of the problems they reported were a direct result of merging or intermingling these architectures.

In Section 2, we described the typical software architectures found in the surveyed systems. Next we describe some of the more elegant examples of software architectures, and the advantages of separating them.

3.1 Conceptual and Execution Architectures in System G

The architecture of System G has an explicit conceptual architecture. One of the design goals for this system is to have a formal (i.e., complete and unambiguous) representation that both serves as documentation for the users, and allows verification by engineers of various disciplines. Such verification may include verification against the system requirements, checking syntactic consistency, and validation with a system simulator.

The conceptual architecture is decomposed into a network of several thousand interconnected functional diagrams. The network organizes functional diagrams into areas of measurement of control variables, and open/closed loop control of devices. Each functional diagram consists of a small number of interconnected basic functional blocks (Figure 8). These functional blocks are the basic units of functionality, each performing a well-defined mathematical or control function. The conceptual architecture also includes protocols for synchronization and fault tolerance; these protocols are the most complex parts of the software.

System G explicitly distinguishes between the conceptual and execution architectures. To organize the execution architecture, it uses two kinds of components, a Group Module, and a Run-time Environment. A Group Module is a group of functional diagrams that are executed on a virtual machine provided by the Run-time Environment. Each Group Module is assigned to the Run-time Environment on a single processor (Figure 9). The local execution architecture for each processor is separately described in terms of its Group Module, its Run-time Environment, processes executing other software modules, buffers, and communication mechanisms.

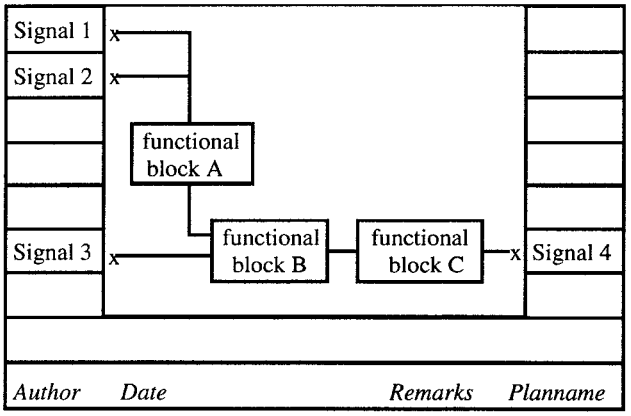

Figure 8: Example of a Functional Diagram in System G

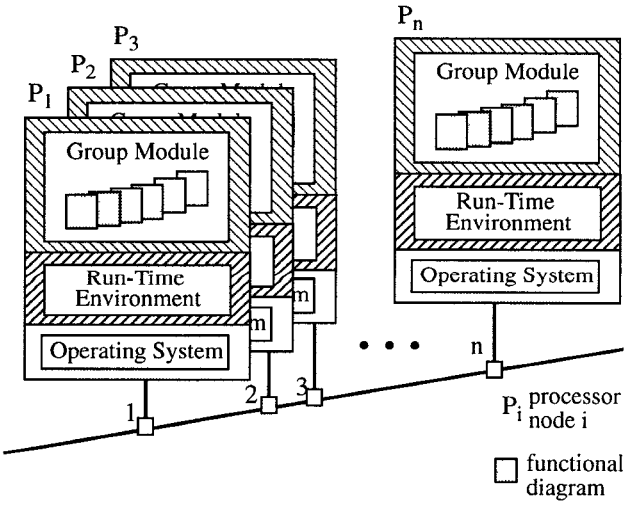

Figure 9: An Overview of the Execution Architecture in System G

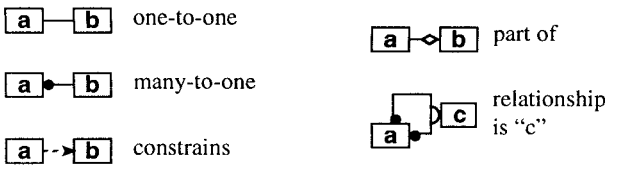

Figure 10: Entity-Relationship-Attribute Notation for Relationships

System G is the best illustration for the benefits of separation of concerns. It clearly separates the instrumentation and control (I&C) functions from the complex protocols related to fault tolerance and synchronization, and separates the execution architecture from the rest of the software. This approach has made the introduction of digital technology both tractable and feasible.

Separation of the conceptual architecture facilitates the design, analysis, verification, and update of I&C functions by process engineers and I&C engineers. The conceptual architecture is preserved and/or traceable in the code, making it possible to update and replace any Group Module at build time.

Separating out the complex protocols supporting fault-tolerance and synchronization made the protocols amenable to formal modeling and analysis, and simplified their implementation and verification. Through the use of extended finite state machines (e.g. Statecharts [15]), several errors in the protocols were caught early in the design phases. This separation also simplified the manual implementation of functional blocks and the Run-time Environment, since the code related to synchronization and fault-tolerance is located elsewhere.

Finally, separation of the execution architecture from the other architectures facilitated reconfiguration of functional diagrams into Group Modules and their processor assignments. The use of a Run-time Environment as a virtual machine that executes functional diagrams also made it easier to change the platform software (e.g. operating system and communication) without affecting the application software (cf. Chimera [34] with its multilevel reconfigurable software framework).

3.2 Module and Execution Architectures in System K

System K is another system with a particularly interesting architecture: here the module and execution architectures are notable. To show these architectures, we use an entity-relationship-attribute notation. Components are represented as entities (shown in boxes), and relationships between components are described with the symbols listed in Figure 10.

Like System B and several other systems, the module architecture of System K encompasses both functional decomposition and a layer structure (Figure 11). The application software of the system is decomposed into subsystems that are further decomposed into subsystems, until reaching the lowest level abstract modules called Service Provision Units (SPUs). The services provided by an SPU are defined by their abstract interfaces. SPUs can interact with each other only through these interfaces, thereby constraining the connections between SPUs. The connections between abstract modules are grouped into system-level connections at the subsystem level.

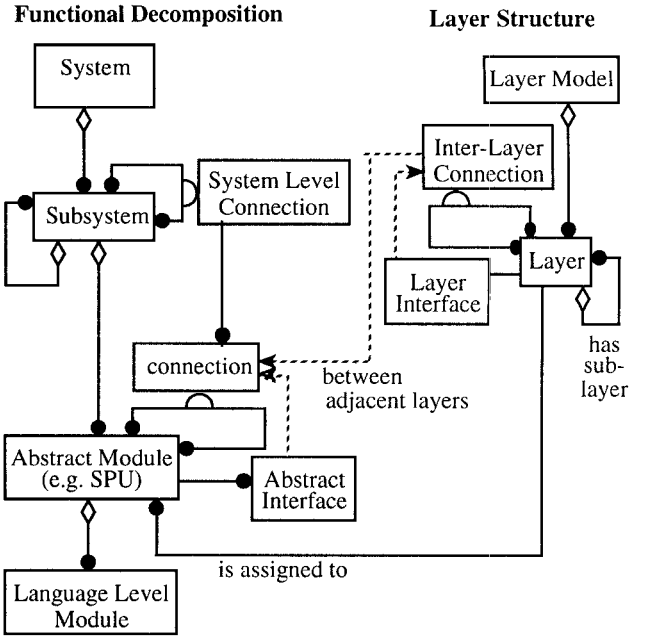

Figure 11: Skeletal Meta-Model of the Module Architecture in System K

An SPU is an abstract module class created by the system architects. SPUs contain a mix of implementation language elements to provide the designer with larger, more abstract functional building blocks. These promote software reusability at the level of services by tightly defining and restricting the scope of the SPUs.

System K illustrates an interesting interplay between the functional decomposition and the layers. The layer structure (Figure 11) consists of a number of layers, each of which may be further decomposed into sublayers. The layer model is designed before any functional decomposition takes place, and is used to guide partitioning of functionality into SPUs. The SPUs from the functional decomposition structure are then assigned to a layer, thereby constraining the SPUs' interfaces and dependencies. This assignment constrains an SPU to interact only with SPUs that are assigned to adjacent layers. The separation into layers allows changes to be made to components at one layer without affecting the components of the other layers. It also provides application independence from the evolving hardware. It also facilitates the bottom-up building and testing of various subsystems.

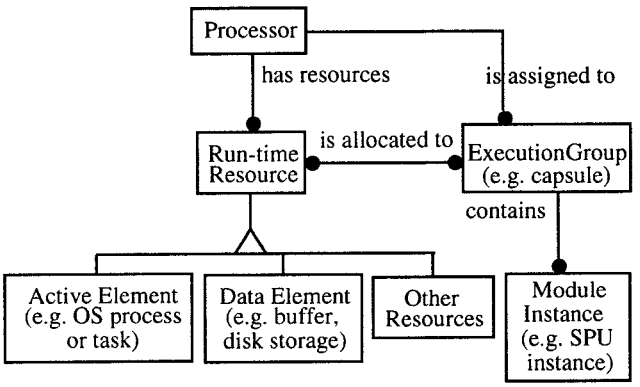

Figure 12: Skeletal Meta-Model of the Execution Architecture in System K

Like System G and several other systems, System K uses an execution grouping to organize the execution architecture (Figure 12). System K's execution group is called a "capsule," and it contains instances of one or more SPUs (see Figure 13). The SPUs that comprise the capsule execute on the same processor and are allocated a subset of the total processing resources (e.g., address space, memory pool, timers). The contents of a capsule are defined at build time. The allocation of SPU instances to capsule is based on grouping SPUs that have related functions and therefore a related cumulative need of computational resources. Capsules are then assigned to processors based on CPU resource and timing requirements.

The execution architecture of System K is explicitly separated from its module architecture in anticipation of potential changes to the execution architecture. In order to facilitate changes to the execution architecture, SPU implementations use loosely-coupled inter-process communication for interaction with other SPUs. This allows alternate grouping of SPUs into capsules and migration of capsules to other processors without any changes to the code.

3.3 Summary

Combining architectures can reduce understandability of the system. For example, combining module and execution architectures can group together functional components that are not logically related. This reduces understandability because decomposition reflects decisions related to task allocation rather than decisions based on functional relationships.

Combining architectures can also significantly increase the cost of porting to a different hardware or software platform, requiring significant rewriting of code. Porting often requires changes in the execution architecture. When the module architecture is made to mirror the execution architecture, the module architecture must change in response to changes in the execution architecture. Such changes in the module interfaces and the process boundaries may result in reworking of code.

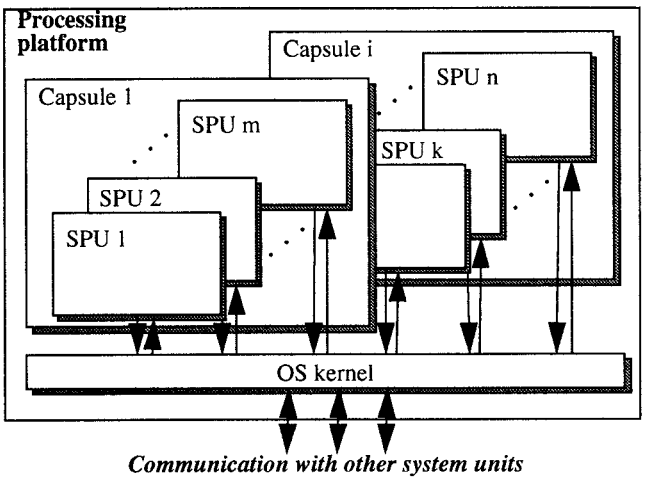

Figure 13: An Overview of the Execution Architecture in System K

In summary, combining architectures can limit evolvability, reconfigurability, and ability to change over to commercial, off-the-shelf technology. This is due to intermingling the implementation of the functionality with implementation decisions related to protocols, communication mechanisms, and resource allocation. Careful separation of such decisions facilitates the adaptation of reusable components to different execution architectures.

4. Architecture Description Techniques

We found that most architects and developers use a combination of informal and semi-formal techniques to describe software architectures. This is not surprising since rigorous architecture description techniques are not available. Entity-relationship-attribute (ERA) notation and finite state machines are two description techniques that were extremely useful for describing and modeling aspects of architecture. CASE tools are beginning to be used to construct functional decomposition diagrams. On the other hand, informal diagrams, tables, and natural language text with naming conventions are used to describe many of the other software structures. Even when a formal notation is used, it is often supplemented with informal and incomplete diagrams, in order to enhance the understanding of the formal model.

The disadvantages of using informal description techniques are described in the literature, and this was borne out in the surveyed systems. Among these are the ambiguity of informal descriptions, and the problems of detecting inconsistencies and establishing traceability between the architecture and the code. In addition, without a rigorous and an unambiguous language to describe architectures, automated tools cannot be used to support architecture-based software development activities.

We found two other common problems when architecture was described informally. It is difficult to recognize and exploit similarities between architectures of other products within a product family. Conversely, it is easy to blur the distinction between quite dissimilar architectures when the descriptions are informal.

4.1 Describing Software Architectures

When explicitly described, the conceptual architecture is usually an informal block diagram depicting major system components and data and control paths between them. A notable exception is System G, in which the conceptual architecture is described using precise, application-specific notation, thus making it possible for tools to check the syntax of the architecture, analyze it for performance, and generate code. This notation is based on an entity-relationship-attribute model. System G uses extended finite state machines (e.g. StateCharts [15]) to model voter-checker protocols, and System K uses a similar formalism to model protocols.

The module architecture is described using a combination of techniques. The functional decomposition is described using decomposition hierarchies (e.g. trees, TeamWork process diagrams), tables, and text. Layers are usually described using stacked rectangular boxes. Adjacencies between these boxes represents allowable interfaces between components in different layers. When the layers have sublayers, they are either depicted in separate diagrams or described in text.

Execution architecture is often embedded in the code architecture or in configuration files used by system building tools. Execution architectures are informally documented using a variety of techniques, including text, tables, and informal diagrams. Such diagrams depict operating system processes, buffers, and communication mechanisms. Sometimes the relationship of execution architecture elements to the hardware elements such as processors and networks is also described. When the systems use module and execution architectures that mirror each other, the execution architecture is often superimposed on the module architecture. For example, the capsule structure (execution architecture) of System K is superimposed on its layer structure. The actual contents of each of the capsules are shown using a separate table.

Again a notable exception is System G, which uses a combination of formal and informal notations to describe the execution architecture. In the formal notation, Group Modules are constructed from collections of functional diagrams. The functional diagrams, Group Modules, run-time elements (e.g. telegrams, buffers) of the execution architecture, and the hardware architecture are modeled using an entity-relationship-attribute model.

Code architectures are generally described with the help of make files or equivalent models for system building.

4.2 Describing Relationships among Software Architectures

For describing the relationship and correspondences between the architectures, we found several methods being used: mirroring one architecture after another, hybrid structures, and correspondence using explicit mechanisms such as assignment of functionality. Several systems mirror one architecture with respect to another, often using naming conventions. For example, the functional decomposition of System B mirrors its execution architecture. That is, each task in the execution architecture is assigned a single module from the module architecture.

In several systems, hybrid structures are used to show the relationship between two architectures, and these hybrid structures can be derived automatically given representations for the architectures and the relationships between them. System K uses a combination of the layer structure and the execution architecture to describe the relationships among them.

One of the systems using an explicit mechanism to describe the relationship between architectures is System G, in which the relationships between the functional diagrams (in the conceptual architecture) and Group Modules (in the execution architecture) are formally based on the entity-relationship-attribute models of the two architectures. System G uses informal diagrams to combine the execution and module architectures. It also uses diagrams that superimpose the module architectures and the mode transition diagrams (from the execution architecture). These informal diagrams are used to better understand the formal model of the architectures and their relationships.

5. Conclusions

In this paper, we have presented a discussion of software architecture in industrial applications. This presentation was based on a survey of the software architecture of a number of systems, with six representative systems studied in depth. The systems represent a variety of application domains, and range in size from small (fewer than 100 KLOC) to very large (more than 1 MLOC). We found that while architecture descriptions and analysis techniques are a mix of informal and semi-formal, separate architectures were used to describe the different concerns addressed by a system's software architecture.

The various architectures describe the system from different perspectives, they are characterized by different needs and goals, and they are used at different times in the development process. Many of the reported successes of the surveyed systems resulted from allowing these architectures to develop independently while maintaining the relationship between the architectures. Similarly, some of

the problems they reported were a direct result of merging or intermingling these architectures.

In examining the components and interconnections in the systems, we found that the structures used to describe architecture fall into four broad categories. The conceptual architecture describes the structure of a system in terms of its functional components and interface connectors, the module architecture describes the ideal implementation structure of the system, the execution architecture describes the dynamic structure of a system in terms of its run-time elements, and the code architecture describes how the source code of the system is organized.

While these categories are not the final answer to the definition of software architecture, we believe that separating structures into more than one perspective is crucial. This separation accurately reflects the practice in development of real software systems, and the reasons for the separation become clear upon studying these systems.

We believe this separation of software architectures will facilitate the use of formal approaches (such as formal description and analysis) that are powerful, but have not readily produced scalable, practical results. Meanwhile, rigorous techniques need to be developed to describe these software architectures, so that they can be analyzed to predict and assure non-functional system properties. It then becomes possible to design, implement, test, and re-engineer a software system based on the rigorous description of its architecture. Finally, all of these architecture-based activities need to be incorporated in existing development processes for maximum effectiveness.

Acknowledgments

We thank all of the architects and developers of the systems we surveyed for their time and patience. We also thank Amitava Datta, Paul Drongowski, Igor Gordon, and Liang Hsu for valuable suggestions.

References

[1] G. Agha, P. Wegner, and A. Yonezawa (eds.). Research Directions in Concurrent Object-Oriented Programming, The MIT Press, 1993.

[2] R. Allen and D. Garlan. A formal approach to software architectures, *Algorithms, Software, Architecture*, J. van Leeuwen (editor), Information Processing 92, Volume 1, Elsevier Science Publishers B.V. (North-Holland), 1992.

[3] M.R. Barbacci, C.B. Weinstock, and J.M. Wing. Programming at the processor-memory-switch level. In *Proceedings of the 10th International Conference on Software Engineering*, IEEE Computer Society Press, pages 19-28, April 1988.

[4] B.W. Beach. Connecting software components with declarative glue, In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[5] C.G. Bell and A. Newell. *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[6] N. Delisle and D. Garlan. A formal specification of an oscilloscope, *IEEE Software*, pages 29-36, September 1990.

[7] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2 (2): 80-86, June 1976.

[8] R. D'Ippolito and K. Lee. Modeling Software Systems by Domains, *AAAI-92 Workshop on Automating Software Design*, July 1992.

[9] P.J. Drongowski. Software architectures in real-time systems. In *Proceedings of the First Workshop on Real-Time Applications*, IEEE Computer Society, May 1993.

[10] M.S. Feather and A. van Lamsweerde. Succeedings of the Seventh International Workshop on Software Specification and Design. Software Engineering Notes, 19 (3): 18-22, July 1994.

[11] P. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie Mellon University, 1991.

[12] S.I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 9: 255-265, November 1979.

[13] H.D. Fischer. Special features of a computer-based German reactor protection system, in *Proceedings of Fault-Tolerant Computing Systems*, Springer-Verlag, Nürnberg, pp. 266 - 287, September 1991.

[14] D. Garlan and M. Shaw. An introduction to software architecture, in *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora (editors), Volume 1, World Scientific Publishing Company, New Jersey, 1993.

[15] D. Harel. Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, Volume 8, pages 231 - 274, 1987.

[16] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, 1988.

[17] C.H. Hoogendoorn and A.T. Maher. Enhanced software architecture for an ATM universal communication node, *Proceedings of the 14th International Switching Symposium*, Yokohama, Japan, October 1992.

[18] I. Jacobsen et al. Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1992.

[19] R.H. Katz, *Information Management for Engineering Design*, Springer-Verlag, 1985.

[20] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[21] J. Kramer. Configuration programming - a framework for the development of distributable systems, In *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, Israel, IEEE, May 1990.

[22] R. Lange, R.W. Schwanke. Software architecture analysis: A case study. *In Proceedings of the Third International Workshop on Software Configuration Management*, Trondheim, Norway, ACM Press, June 1991.

[23] E. Mettala and M.H. Graham. The domain-specific software architecture program, Technical Report, CMU/SEI-

92-SR-9, Carnegie Mellon University, Software Engineering Institute, June 1992.

[24] K. Narayanaswamy and W. Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, SE-13(3), March 1987.

[25] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes*, Volume 17, Number 4, pages 40-52, October 1992.

[26] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307-334, November 1986.

[27] J.M. Purtilo. The polylith software bus. To appear, *ACM Transactions on Programming Languages and Systems*, January 1994. Also available as Technical Report UMIACS-TR-90-65, University of Maryland, May 1990.

[28] W. Royce. TRW's Ada process model for incremental development of large software systems. In *Proceedings of the 12th International Conference on Software Engineering*, IEEE Computer Society Press, pages 2-11, March 1990.

[29] J. Schaeffer, D. Szafron, G. Lobe, I. Parsons. The Enterprise model for developing distributed applications, Dept. of Computing, University of Alberta, 1993.

[30] B. Selic, G. Gullekson, J. McGee, and I. Engelberg. ROOM: an object-oriented methodology for developing real-time systems, In *Proceedings of the CASE'92 Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, Canada, July 1992.

[31] M. Shaw. Larger scale systems require higher level abstractions, in *Proceedings of the Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, pages 143-146, May 1989.

[32] D. Soni, R.L. Nord, L. Hsu, P. Drongowski. Many faces of software architectures, In *Proceedings of the Workshop on Studies of Software Design*, Baltimore, May 1993 (to appear in LNCS).

[33] D. Soni, R.L. Nord, L. Hsu. An empirical approach to software architectures, in *Proceedings of the Seventh International Workshop on Software Specification and Design*, IEEE Computer Society, December 1993.

[34] D.B. Stewart, R.A. Volpe, and P.K. Khosla. Integration of software modules for reconfigurable sensor-based control systems, In *Proceedings of 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS '92), Raleigh, North Carolina, July 1992.

[35] W. Tichy. Software development control based on module interconnection. In *Proceedings of the Third International Conference on Software Engineering*, pages 29-41, IEEE Computer Society Press, May 1979.

[36] M. Weiser. Source Code, *IEEE Computer*, pages 66-73, November 1987.

[37] A. Wolf. Language and tool support for precise interface control. Technical Report COINS-TR-85-23, University of Massachusetts, 1985.

[38] Y. Yokote. The Apertos reflective operating system: the concept and its implementation, In *Proceedings of OOPSLA'92*, pages 414-434, October 1992.

A Process for the Study

The software systems we studied were all developed for industrial applications. Although the designers and developers of the various systems may disagree on exactly what the software architecture consists of, they all recognize its importance, and some are at the leading edge in their use of software architecture in industry. As a part of the study, we talked to architects and engineers, read design documents, and, in some cases, read parts of the source code. The sequence of steps we undertook is as follows:

1. We started with an initial list of questions to guide our study. The questions were related to how architecture was described and used in the development of a system. We developed a questionnaire based on these questions and sent it to the organizations we intended to visit. This questionnaire appears in Appendix B.

2. We visited the organizations to present our premise and the objectives of our study.

3. We collected and read design documents and, in some cases, code for the systems.

4. We held extended discussions with the architects and designers of these systems. In some cases, these discussions lasted several days over several visits. We used the questionnaire during these interviews to structure the initial discussion and to gather information. We also held a workshop on software architectures where the architects described and discussed the architecture of their systems.

5. We selected six of the systems for more detailed study, and carefully studied the available artifacts for these systems. These six, annotated with asterisks in Table 1, represent a wide variety of application domains, and they are among the systems for which we had the most information.

6. We distilled the information from these six systems using a structured template. This template appears in Appendix C.

7. We analyzed the distilled information to arrive at our findings. Based on this analysis, we restructured the description of the systems to improve understanding, and prepared the internal technical report.

8. We then sent the technical report to the designers and architects of the systems for their comments, and modified it based on these comments. The internal report provides the basis for this paper.