A Product Line Architecture for a Network Product

Dewayne E. Perry

Electrical and Computer Engineering
The University of Texas at Austin
Austin TX 78712 USA
+1.512.471.2050
perry@ece.utexas.edu
www.ece.utexas.edu/~perry/

ABSTRACT

Given a set of related (and existing) network products, the goal of this architectural exercise was to define a generic architecture that was sufficient to encompass existing and future products in such a way as to satisfy the following two requirements: 1) represent the range of products from single board, centralized systems to multiple board, distributed systems; and 2) support dynamic reconfigurability.

We first describe the basic system abstractions and the typical organization for these kinds of projects. We then describe our generic architecture and show how these two requirements have been met. Our approach using late binding, reflection, indirection and location transparency combines the two requirements neatly into an interdependent solution – though they could be easily separated into independent ones.

We then address the ubiquitous problem of how to deal with multiple dimensions of organization. In many types of systems there are several competing ways in which the system might be organized. We show how architectural styles can be an effective mechanism for dealing with such issues as initialization and exception handling in a uniform way across the system components.

Finally, we summarize the lessons learned from this experience.

0.1 Keywords

Software Architecture Case Study, Dynamic Reconfiguration, Distribution-Free Architecture, Architecture Styles, Multiple Dimensions of Organization

1 Introduction

This study represents a snapshot in the process of constructing a generic architecture for a product line of network communications equipment. The intent of the project was to create the software architecture for the next generation of

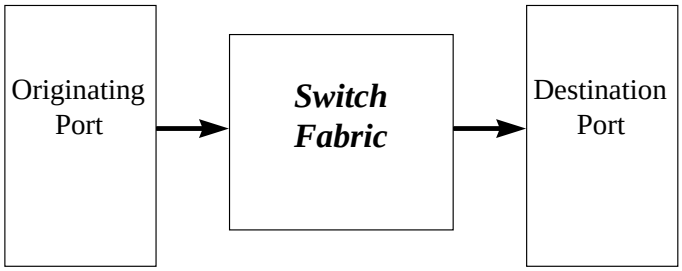

Fig. 1. Basic Abstraction: Connection. A connection consists of an originating port connected via a switch fabric to a destination port.

products in this domain using the existing set of products as the basis for that effort.

The project began in an unusual way: the software architecture team came to research looking for help with their project. They had the domain expertise for the network product as well as experience as architects and system builders. I had experience as a software designer and architect in a variety of domains (but not this one) as well as research expertise in software architecture in general and product line architectures in particular. The result was a fruitful collaboration that lasted about 9 months.

Several caveats are in order before we proceed to discuss the issues and their solutions.

- First, we do not describe the complete architecture. Instead, we concentrate only on the critical issues relevant to the product line and the implications of these issues.
- Second, we present only enough of the domain specific architecture to provide an appropriate context for the part of the architecture and the issues we focus on.
- Third, we address only three architectural issues and describe several architectural techniques that solve these issues in interesting ways.
- Fourth, we do not here discuss issues of analysis such as performance. The architects already did that very well and, as a researcher, that was not where my expertise was applicable (it was in the areas of basic abstractions and generic descriptions). The primary performance issue related to the discussion below was about the efficiency of current commercially ORBs — the one selected appeared to satisfy the required constraints.
- Fifth and finally, we do not provide a full evaluation of the architecture (for example, how well did it work in the end) primarily because, for a variety of reasons, the project was not completed. We do, however, offer the positive consensus of the project architects and their satisfaction with the resulting solutions we discuss below.

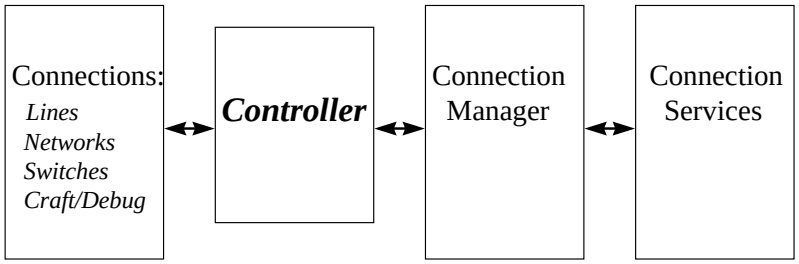

Fig. 2. Basic Hardware/Software System: consists of four logical elements: connections, controllers, connection manager and connection services.

We first provide the context for the study (the product line domain, the current and desired states of the product line, and a basic view of the products). We then explore the implications of the selected system goals and what is needed at the architectural level to satisfy these goals. On this basis, we the describe our architectural solutions and the motivation behind our choices. Finally, we summarize what we have done and lessons we learned in the process.

2 Product Domain

The product line consists of network communication products that are hardware event-driven, real-time embedded systems. They have high reliability and integrity constraints and as such must be fault-tolerant and fault-recoverable. Since they must operate in a variety of physical environments, they are *hardened* as well.

These products are located somewhere between the house and the network switch. They may sit on the side of a building or on some other outside location (for example, a telephone pole), or partly there and partly near a network switch, depending on how complicated the product is (that is, depending on the complexity of the services provided and the number of network lines handled).

The current state of the products in this product line is that each one is built to a customer's specifications. Evolution of these products consists of building both the hardware and software for new configurations.

Central to a satisfactory architecture are the fundamental domain abstractions. They provide the basic organizing principles. Here the key abstraction is that of a *connection*. A connection consists of an originating communications line port connected through a switch fabric (appropriate for the type of network service provided) to a destination port. The connections range from static ones (which once made remain in existence until the lines or devices attached to the ports are removed) to dynamic ones (which range from simple to very complex connections that vary in the duration of their existence) — see Figure 1.

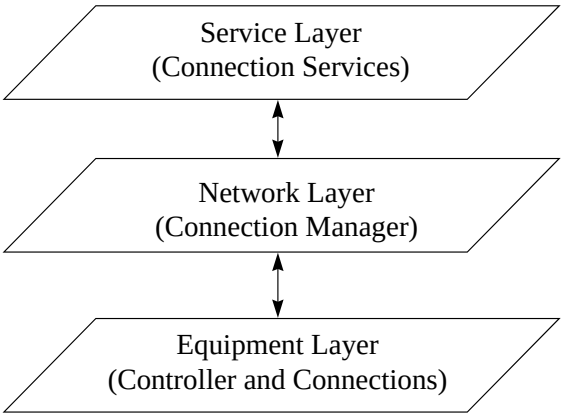

Fig. 3. Typical Domain-Specific Architecture: a structure of three layers consistent with the standard network model.

The typical system structure for these products (see Figure 2) consists of a set of connections such as communication lines, switches, other network connections, and craft and debugging interfaces. These devices have various appropriate controllers that are handled by a connection manager which establishes and removes connections according to hardware control events and coordinates the services required to support the connections.

Figure 3 shows a typical architecture for such network communication products layered into service, network and equipment layers. Within each layer are the appropriate components for the functionality relevant to that layer.

3 Basic System Goals

The basic requirements for the product line architecture we seek are:

- Requirement 1. To cover the large set of diverse product instances that currently exist and that may be desired in the future
- Requirement 2. To support dynamic reconfiguration so that the products existing in the field can evolve as demands change for new and different kinds of communication.

Thus the desired state of the product line is that products can be reconfigured as needed with as little disruption as possible (but not requiring continuous service). For the hardware, this entails common interfaces for the various communication devices and plug compatible components. This part of the project was addressed by the hardware designers and architects. For the software, this entails a generic architecture for the complete set of products and software support for dynamic reconfiguration of the system. This part is what we addressed.

The first question then is how do we create a generic architecture that covers the entire range of products in the product line — that is, how do we satisfy requirement 1? These products range from simple connection systems that consist

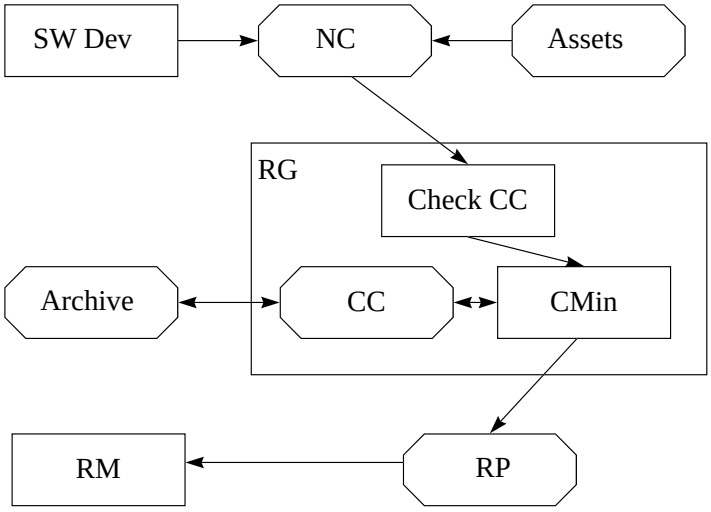

Fig. 4. Reconfiguration: Reconfiguration Generation is shown in detail: new architectural configuration (NC), check consistency and completeness (Check CC), minimize configuration (CMin), reconfiguration package (RP), and current configuration (CC) ; Reconfiguration Management (RM) is shown in detail in figure 5.

of a processor, associated controllers and devices, to complex connection systems that consist of multiple processors, associated controllers and devices which may be distributed over several locations.

The main question is how do we handle this range of variability in component placement and interaction? If we address the issue of distribution at the architectural level, then that implies that distribution is a characteristic of all the instances. What then do we do with simpler systems? A separate architecture for each different class of system defeats the goal of a single generic architecture for the entire product line.

One answer to this problem of variability is to create a *distribution independent* architecture [4] (requirement 1.1) and thus bury the handling of the distribution issues down into the design and implementation layers of the system. In this way, the distribution of components is not an architectural issue.

However, this decision does have significant implications at the architectural level about how the issues of distribution are to be solved. First, the system needs a model of itself that can be used by the appropriate components that must deal with issues of distribution. For example, the component handling system commands and requests must know where the components are located in order to schedule and invoke them. Thus, second, we need a command broker that provides *location transparent* communication, that is configurable, that is priority based and that is small and fast. So not only do we get a view of the architecture where distribution is not an issue, we get a component view of

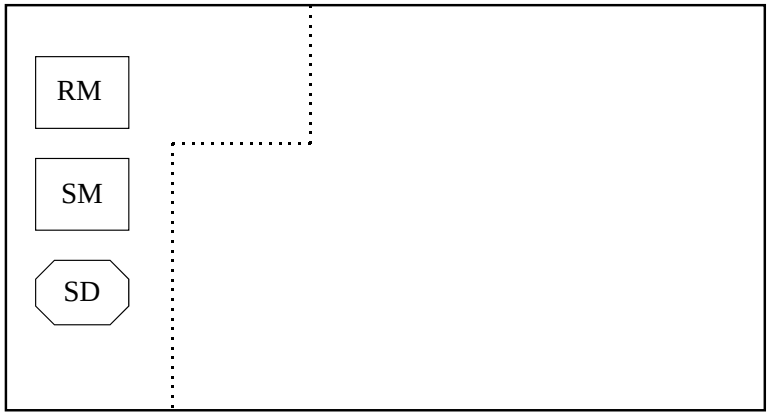

Fig. 5. Reconfiguration Components: System Model (SM), System Data (SD), and Reconfiguration Manager (RM). The dotted line separates the domain specific part from the reconfiguration and distribution-independence parts of the architecture.

communication where distribution is not an issue either. Finally, the components need to be location independent in order to be useful across the entire range of products.

To satisfy requirement 2 for dynamic reconfiguration, it is necessary only to minimize down time. We do not need to provide continuous service. However, we need to be able to reconfigure the system *in situ* in any number of ways from merely replacing line cards to adding significantly to the size and complexity of a system (for example, changing a simple system into a complex distributed one) in the hardware and from changing connection types to adding and deleting services in the software.

As with the issue of distribution, reconfigurability requires a model of the system and its resources, and obviously, a reconfiguration manager that directs the entire reconfiguration process both systematically and reliably. For this to work properly, the components have to have certain properties akin to location independence for a distribution-free system. In this case, we need configurable components. We shall see below that these necessary properties can be concisely described in an architectural style [1].

4 Architectural Organization

By and large, a product line architecture is the result of pulling together various existing systems into a coherent set of products. It is essentially a legacy endeavor: begin with existing systems and generalize into a product line. There are of course exceptions, but in this case the products preceded the product line.

The appropriate place to start considering the generic architecture is to look at what had been done before. In this case we drew on the experience of two

teams for two different products and use their experience to guide us in our decisions.

As in many complex systems, there are multiple ways of organizing [2] both the functionality and the various ways of supporting nonfunctional properties. In this case, we see two more or less orthogonal dimensions of organization: system objects and system functionality. System objects reflect the basic hardware orientation of these systems: packs, slots, protection groups, cables, lines, switches, etc. System functionalities reflect the things that the system does: configuration, connection, fault handling, protection, synchronization, initialization, recovery, etc.

Given the two dimensions, the strategy in the two developments was to organize along one dimension and distribute the other throughout that dimension's components. In the one case, they chose the system object dimension, in the other they chose the system functionality dimension. In the former, the system functionality is distributed across the system objects — for example, each system object takes care of its own initialization, fault tolerance, etc. In the latter, the handling of the various system objects is distributed throughout the system functions — for example, initialization handles the initialization for all the objects in the system.

Both groups felt their solutions were unsatisfactory and were going to choose the other dimension on their next development.

Our strategy then was to take a hybrid approach: choose the components that are considered to be central at the architectural level and then distribute the others throughout those components — a mix and match approach. The question then is how to gain consistency for the secondary components that get distributed over the architectural components. We illustrate the use of architectural styles as a solution to this problem in two interesting cases below.

5 Architectural Solution

We discuss our solutions to the issues we have raised and show how these different solutions fit together to resolve these issues in interesting ways. We discuss first the architectural components needed to support dynamic reconfigurability. We then discuss how distribution independence can be integrated with reconfigurability. We then delineate the general shape of the domain-specific part of the generic architecture and describe how the entire architecture fits together. We then discuss the two primary connectors: one for reconfiguration and one for system execution. Finally, we present two architectural styles to illustrate the distribution of the secondary dimension objects across the primary dimension of organization.

5.1 Reconfiguration (Requirement 2)

Reconfiguration is split into two parts: reconfiguration generation and reconfiguration management. The reconfiguration generator is outside the architecture

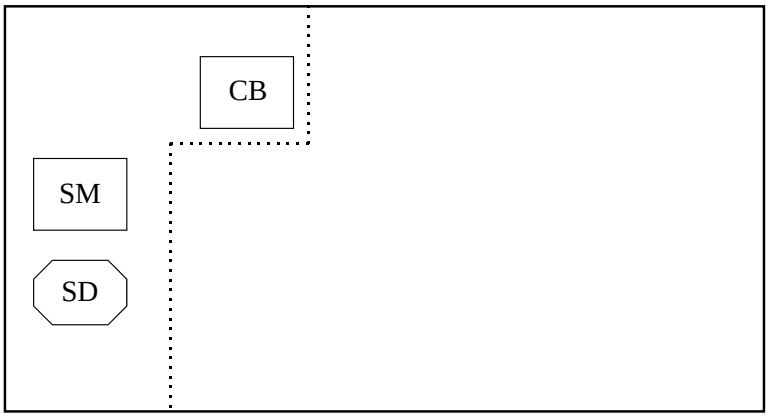

Fig. 6. Distribution Independence Components: System Model (SM), System Data (SD), and Command Broker (CB).

of the system and ensures two primary requirements: first (requirement 2.1), that the reconfiguration constraints for completeness and consistency of a configuration are satisfied; second (requirement 2.2), that the configured system is minimal [3], a requirement due to both space and time limitations.

The question arises then as to where this part of reconfiguration should be. Given the space and economic considerations of the systems, we chose to have the consistency checking and reconfiguration minimization done outside the bounds of the system architecture.

In Figure 4, a new architectural configuration (NC) is created by combining new components from software development (if there are any) with existing assets and passing them to Reconfiguration Generation (RG). The new configuration is then checked for consistency and completeness (Check CC). Once it is established that those constraints are satisfied, the new configuration is compared against the current configuration to to determine which architectural components need to be added and deleted (C Min). The result is a Reconfiguration Package (RP) which is passed to the Reconfiguration Manager (RM) containing the instructions for dynamically reconfiguring the software part of the system.

To satisfy requirement 2 for system reconfigurability, we have the three components illustrated in Figure 5: the reconfiguration manager (RM), the system model (SM) and the system provisioning data (SD).

The system model and system data provide a logical model of the system, the logical to physical mapping of the various elements in the system configuration, and priority and timing constraints that have to be met in the scheduling and execution of system functions.

The reconfiguration manager directs the termination of components to be removed or replaced, performs the component deletion, addition or replacement, does the appropriate registration and mapping in the system model, and handles

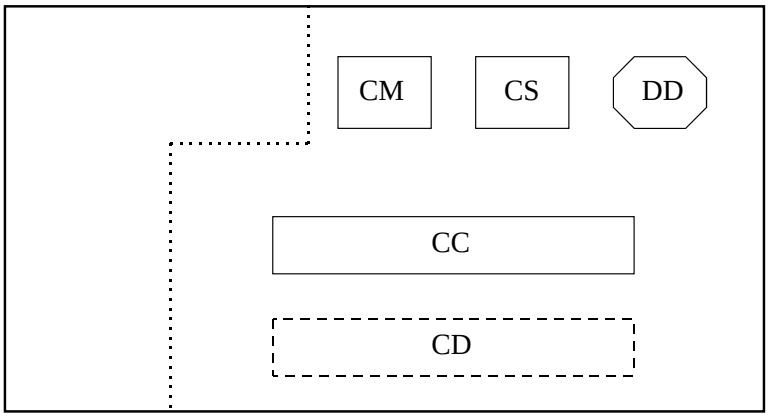

Fig. 7. Domain-Specific Components: Connection Manager (CM), Connection Services (CS), Dynamic Data (DD), Connection Controller (CC), and Connection Devices (CD).

startup and reinitialization of new and existing components. Special care has to be taken in the construction of the reconfiguration manager so that it can properly manage self-replacement, just as special care has to be taken in any major restructuring of the hardware and software.

5.2 Distribution Independence (Requirement 1.1)

For the satisfaction of the distribution independence requirements, we have the three components illustrated in Figure 6: the command broker (CB), the system model (SM) and the system provisioning data (SD). Note that the system model and the system provisioning data are the same as in the reconfiguration solution.

The command broker uses the system model and system provisioning data to drive its operation scheduling and invocation. System commands are made in terms of logical entities and the logical to physical mapping is what determines where the appropriate component is and how to schedule it and communicate with it.

5.3 The Domain Specific Components

For the domain-specific part of the architecture we have chosen as the basic architectural elements the connection manager (CM), the integrity manager (IM), the connection services component (CS), the dynamic data component (DD), the connection controllers (CC), and the connection devices (CD). These components represent our choices for the architectural abstractions of both the critical objects and the critical functionality necessary for our product line. Of these, the integrity manager is a logical component whose functionality is distributed throughout the other components shown in Figure 7.

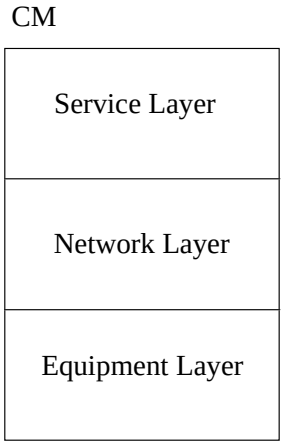

Fig. 8. Domain Specific Component Decomposition. The traditional layering forms the basis of the subarchitectures of several of the basic domain specific components. Here we see the decomposition of the Connection Manager (CM).

While we have not used the typical network model as the primary organizing principle for the architecture, it does come into play in defining the hierarchy or decomposition of several of the basic domain specific system components: the connection manager (Figure 8 illustrates this decomposition of this component), the connection services, and the connection controller.

5.4 Connectors

The reconfiguration interactions shown in Figure 9 illustrate how the reconfiguration manager is intimately tied to both the system model and the system provisioning data. This part of the reconfiguration has to be handled with care in the right order to result in a consistent system. Further, the reconfiguration manager interacts with itself and the entire configuration as well as the individual components of the system: terminate first, preserve data, reconfigure the system model and system provisioning, and then reconfigure the components. There are integrity constraints on all of these interactions and connections.

A logical software bus provides the primary connector amongst the system components for both control and data access. The manager of the bus is the command broker. There are other connectors as well, but they have not been necessary for the exposition of the critical aspects of the generic architecture. There are both performance and reliability constraints that must be met by this primary connector. How to achieve these constraints was well within the practicing architects expertize and as such is not, as performance issues in general are not, within the scope of our research contributions nor the scope of this paper.

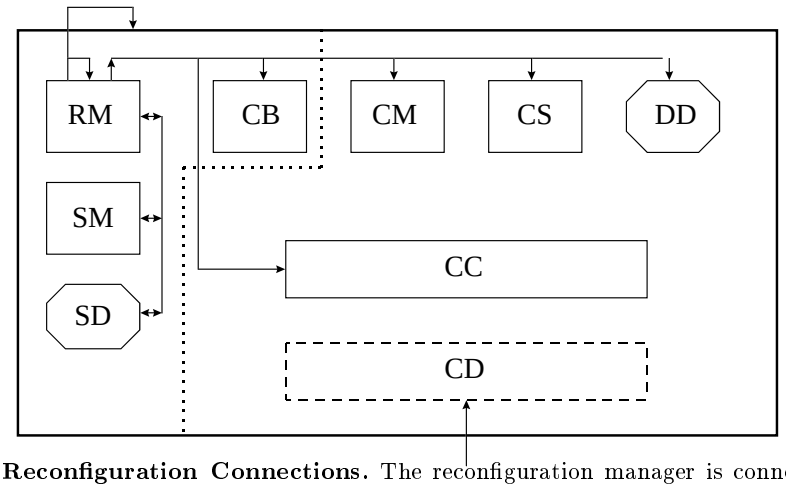

Fig. 9. Reconfiguration Connections. The reconfiguration manager is connected in various ways to all the components in the system, including itself and the system as a whole.

5.5 Architectural Styles

So far we have delineated the primary architectural components derived from the goals for reconfiguration or distribution independence, or from the two domain-specific dimensions of organization possible for this product. For those domain-specific components not chosen, we provide architectural styles to ensure their unform implementation across all the chosen components. We present two such styles as examples: a reconfigurable component style and an integrity management style.

The reconfigurable component architectural style that must be adhered to by all the reconfigurable components has the following constraints:

- The component must be location independent
- Initialization must provide facilities for start and restart, rebuilding dynamic data, allocating resources, and initializing the component
- Finalization must provide facilities for preserving dynamic data, releasing resources, and terminating the component

We had also mentioned earlier that the integrity manager was a logical component that was distributed across all the architectural components. As such there is an integrity connector that hooks all the integrity management components together in handling exceptions and recovering from faults. We had also indicated that the part of the integrity management would be defined as an architectural style that all the system components had to adhere to. This style is defined as follows:

- Recover when possible, otherwise reconfigure around the fault
- Isolate a fault without impacting other components
- Avoid false dispatches

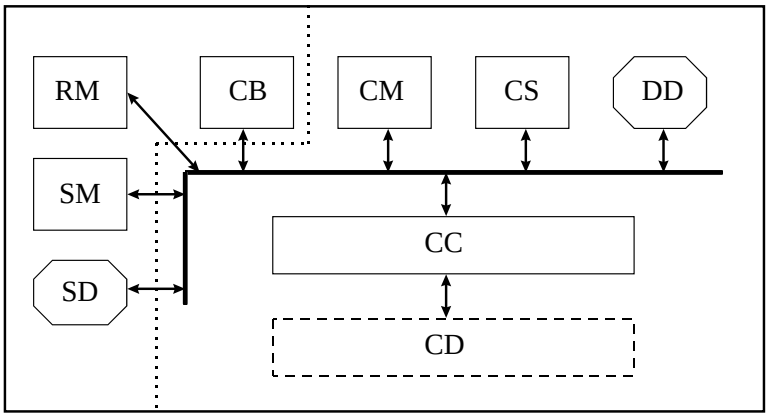

Fig. 10. Architectural Connections. A software bus provides the primary control and data connectors among the system components.

- Provide mechanisms for inhibiting any action
- Do not leave working components unavailable
- Enable working in the presence of faults
- Recover from single faults
- Protect agains rolling recoveries
- Collect and log appropriate information
- Map exceptions to faults
- Enable sequencing of recovery actions

Styles such as these function as requirements on architectural components to guarantee a consistent and uniform set of component properties and behaviors.

6 Summary and Lessons

We have explored several interesting techniques to achieve a generic architecture that satisfied both the domain-specific requirements and the product-line architecture requirements.

To delineate the appropriate domain-specific components, we used a hybrid approach in which we selected what we considered to be the critical elements from two orthogonal dimensions of organization. We then defined architectural styles to ensure the consistency of the secondary components distributed throughout the primary components.

We defined a logical software bus, subject to both performance and reliability constraints, as a general connector among the components. These constraints are especially important where the underlying implementation and organization is distributed across several independent physical components.

To achieve the appropriate goals of the generic architecture covering a wide variability of hardware architectures and enabling dynamic reconfiguration, we

chose a data-driven, late binding, location transparent and reflective approach. This enabled us to solve both the problem of centralized and distributed systems and the problem of reconfiguration with a set of shared and interdependent components.

As to lessons learned:

- There are many ways to organize an architecture, even a domain specific one. Because there are multiple possible dimensions of organization, some orthogonal, some interdependent, experience is a critical factor in the selection of critical architectural elements, even when considering only functional, much less when considering non-functional, properties.
- It is important for any architecture, design or implementation to have appropriate and relevant abstractions to help in the organizing of a system. An example in this study is that of a connection as the central abstraction. Concentration on the concepts and abstractions from the problem domain rather than the solution domain is critical to achieve these key abstractions.
- Properties such as distribution-independence or platform-independence are extremely useful in creating a generic product line architecture. They do, however, come at a cost in terms of requiring architectural components to implement the necessary properties of location transparency or platform transparency.
- Architectural styles are an extremely useful mechanism in ensuring uniform properties across architectural elements, especially for such considerations as initialization, exception handling and fault recovery where local knowledge is critical and isolated by various kinds of logical and physical boundaries. These styles define the requirements that the system components must satisfy to guarantee the properties and behaviors of the secondary components.

Acknowledgements

Nancy Lee was my liaison with the architectural group on this project. She helped in many ways, not the least of which was making project data and documents available for me to write up this case study. The system architects on the project as a whole were very tolerant of an outsider working with them. However, we achieved a good working relationship combining their domain expertize with my research investigations together with a willingness to explore alternative possibilities.

Thanks also to Ric Holt at Waterloo for his comments and suggestions.

References

1. Dewayne E. Perry and Alexander L Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992)
2. Dewayne E. Perry. Shared Dependencies. In *Proceedings of the 6th Software Configuration Management Workshop*, Berlin, Germany, March 1996.

3. Dewayne E. Perry. Maintaining Minimal Consistent Configurations. Position paper for the 7th Software Configuration Management Workshop, Boston Massachusetts, May 1997. Patent granted.
4. Dewayne E. Perry. Generic Architecture Descriptions. In *ARES II Product Line Architecture Workshop Proceedings*, Los Palmas, Spain, February 1998.