Software Connectors and Refinement in Family Architectures

Alexander Egyed Nikunj Mehta Nenad Medvidovic
{aegyed, mehta, neno}@sunset.usc.edu

Department of Computer Science
University of Southern California
Los Angeles, CA 90089-0781, USA

Abstract

Product families promote reuse of software artifacts such as architectures, designs and implementations. Product family architectures are difficult to create due to the need to support variations. Traditional approaches emphasize the identification and description of generic components which prove too rigid to support variations in each product. This paper presents an approach that supports analyzable family architectures using generic software connectors that provide bounded ambiguity and support flexible product families. It describes the transformation from a family architecture to a product design through a four-way refinement and evolution process.

1. Introduction

Large, complex systems are often developed in the context of *product families*[1]. This enables developers to maximize reuse, accelerate the development process while reducing costs, and deliver products that are generally more reliable. At the same time, explicit focus on common *architectural* idioms has the potential to fundamentally transform the nature of software development, as component *integration* replaces implementation as the predominant development activity. The promise of software architectures is that better software systems can be built in this manner more quickly by modeling their important aspects throughout, and especially early in the development. Coupling the benefits of product family-based and architecture-based development has been recognized as an area with a great potential payoff, as evidenced by a growing number of conferences, workshops, and symposia that focus on this subject [2, 3, 4, 8, 12].

The existing body of research in the area of software architectures for product families is characterized by two major foci:

1. specification of generic, product family architectures (also referred to as *reference* architectures) and their instantiation into application architectures (e.g., [11]); and
2. identification and integration of reusable components that comprise different members of a product family (e.g., [5]).

In this paper, we focus on two additional issues that have not been addressed by existing approaches and that are useful complements to those identified above:

1. the role of software connectors in specifying and ensuring the extra-functional properties of both a product family and individual applications within the family; and
2. refinement of an instantiated product architecture into a design and, eventually, an implementation.

The role of *connectors* in software architectures is to isolate all communication, coordination, and mediation [10]. As such, connectors do not provide any domain-specific functionality, but rather enable the functional elements (components) to interact. Thus, another hypothesis is that certain varying properties of applications within a family (e.g., deployment profile, concurrency, interoperability platform, performance, reliability, security, etc.) can be isolated within connectors. Also, certain types of connectors may occur regularly within a family. Our on-going work on classifying software connectors will serve as a vehicle for exploring these issues.

To enable the *refinement* of an architecture into its implementation, we leverage our work on transforming architecture-level constructs (specified in an architecture description language, or ADL [9]), into design-level constructs (specified in the Unified Modeling Language, or UML [6]), and enabling the refinement of the resulting high-level design in a property-preserving manner [1, 7]. We introduce the notion of *product family design*, analogous at the design level to a product family architecture. A product family de-

[1] In this paper, we use the following phrases interchangeably: families, application families, product families, product lines, and domain-specific software.

sign captures recurring design patterns across components in a family. Our hypothesis is that both product family designs and product architectures are needed to enable effective refinement.

The paper is organized as follows. Section 2 identifies the relationships between products and families as well as architectures and designs. Section 3 outlines the role of software connectors in family architectures. Instantiation of a product family architecture and refinement of the resulting product architecture into its implementation is discussed in more detail in Section 4. Section 5 presents an example illustrating the approach. Conclusions and a discussion of open issues round out the paper.

2. Relationship of Products and Families

Software architectures can be described using components, connectors and configurations [9]. Architecture description identifies the obligations and freedoms of the software built to that architecture. Obligations allow a high level analysis of system properties and correctness, whereas the freedoms allow developers to design and implement the system according to the characteristics and constraints of the underlying infrastructure. Since a product family consists of products with commonalities and differences, it is useful to capture these aspects of the individual products in *family architecture*. Further, use of the same architectural elements to describe family architectures and individual product architectures aids in keeping these artifacts consistent.

A family architecture provides generic information common to all the products of the family. In order to support variations in the individual products, a family architecture describes the architectural elements with a certain amount of ambiguity. The product architecture, on the other hand, identifies specific architectural choices for a single product and thus can be considered as an instantiation of the family architecture. The product architecture is also less ambiguous since all architectural elements have to be chosen for the sake of completeness. Proceeding in another direction, the family architecture can also be refined to create a set of family designs that can be used in individual products to obtain the recurring functional and extra-functional properties.

Figure 1 depicts a high-level framework that supports architectural modeling of product families and their improved refinement and evolution. The white boxes and arrows in the figure denote the traditional way of instantiating product architectures from family architectures, followed by the refinement of product architectures into their designs (and subsequent implementations). To complement this traditional approach, we introduces the concept of *family design*. A family design contains design related information about a product family that the architecture did not (or could not) specify. For instance, a family design could contain different design interpretations of architectural elements (e.g., in the form of design patterns). Merging the product architecture information with family design information can then lead to a product design. The product architecture then defines at a high level *what* needs to be designed and the family design provides information on *how* to design it. This four-way relationship between architecture and design, family and product implies that there are at least two alternate but complementary paths of creating designs for the products of a family.

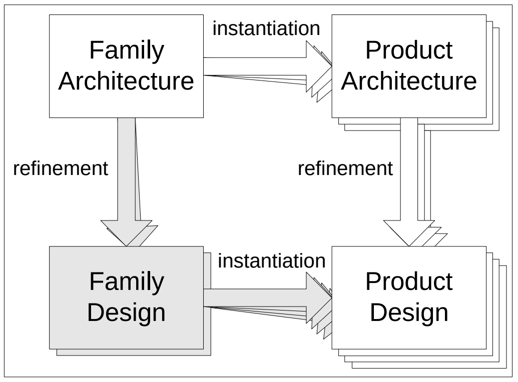

Figure 1. Design refinement and instantiation using product architecture and family design

3. Software Connectors in Family Architecture

Family architectures capture the essential properties and relations of the product family. They describe structural and behavioral freedom and model the functional and extra-functional aspects of a family. Behavioral freedom and functional aspects of the product family are typically captured in components. Our approach also supports the description and analysis of extra-functional properties, coupled with the identification of the structural freedoms through the use of semantically rich connectors.

As discussed above, family architectures need to describe commonalities as well as variations among family members. Commonalities can be captured through elements that are mandatory to all products. The real difficulty lies in modeling variations that have to be supported at the level of a product family. Various approaches have been proposed to describe family architectures including the use of styles, parameters, constraints and service provisioning [11]. However, none of these techniques adequately addresses the problem of supporting variations in the product family adequately. There is clearly a need for defining family architectures with a certain degree of *bounded ambiguity* in order to support product variations.

Consider the case of a customer service product family that needs to support two product domains, retail banking and telephony. These products require variations in terms of the underlying information, as well as in the interaction of the architectural components. The banking application requires online transactions, whereas the telephony product requires a batch update. A useful family architecture would be able to support the description of both kinds of products.

Software architecture captures the essential structural and behavioral information in the form of components and connectors. Family architectures are useful because they lead to better structured reuse and also because the bulk of the architectural analysis can be performed at the level of an entire family. At the level of a product family, components have to be vaguely described because family architectures need to support a variety of product features. This vagueness about components reduces our ability to reason about the family architecture. Components often have to be very rigidly represented in family architectures. Reuse technologies that depend on interchangeable software components which can be achieved only through considerable standardization efforts [21]. This is too rigid a constraint on the development of product families which are required to be developed rapidly to meet time-to-market requirements. Component-based reuse therefore tends to take longer to adapt and is applicable only to narrow domains. On the other hand, the software industry has very quickly embraced component integration frameworks such as DCE RPC [22], COM [23], CORBA[24] and Enterprise Java Beans [25]. We therefore focus our research on the role of software connectors in family architectures.

Software connectors describe the interactions among architectural components and support communication, coordination, conversion and facilitation needs of components. Connectors can be used to describe a range of interactions among components in family architectures. Furthermore, the extra functional properties can be neatly abstracted into semantically rich connector mechanisms such as events, distributors and arbitrators. Connectors have a high potential for reusability since they can be applied across problem domains. They are essential determinants of the extra functional properties of a system such as its availability, throughput, security and scalability. Various architecture-styles motivated by software connectors have been studied, e.g. pipe and filter [14], real-time data feeds [15], event-driven architecture [16], message-based style [17] and middleware-based style [18]. Architectural styles are an important mechanism for enabling reuse in family architectures [11, 13], which implies that software connectors have a major role to play in enabling architecture-based reuse.

Connectors provide bounded ambiguity that is necessary for supporting variations in family architectures. The ambiguity is contained in the various dimensions along which a connector can be characterized and the range of values that a connector can choose for each dimension. We have developed a preliminary taxonomy of software connectors that describes the connector types, dimensions and their possible values (see figure 2 for an extract). Family architectures can be described by using connectors without having to specify the values of their dimensions. Our taxonomy allows architects to choose the concrete connectors necessary to support interaction among the components and to provide the dimensions along which each specific product can choose a different variation of interaction. It is then possible to analyze family architectures for their extra-functional properties as

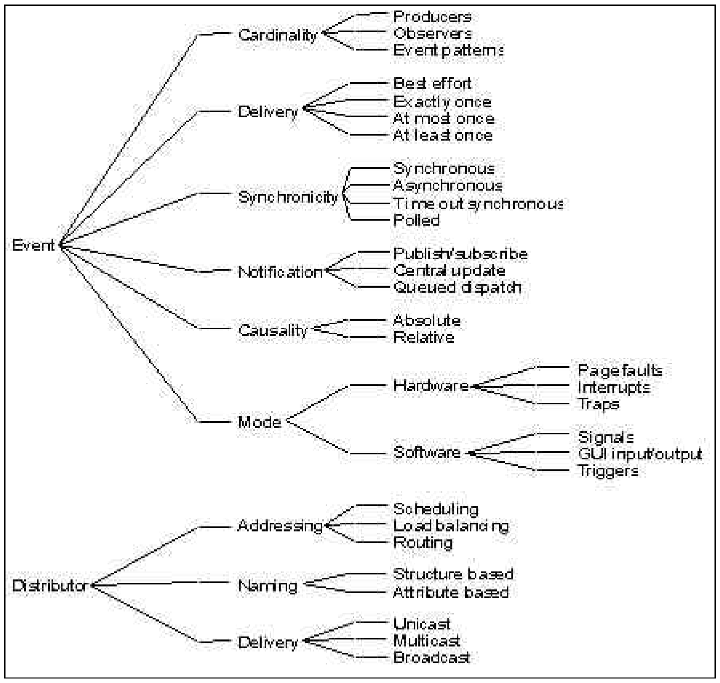

Figure 2. An example connector taxonomy showing types, dimensions and values.

the dimensions of connectors are known and the variation in these properties for each product architecture can be more readily assessed.

As a solution for the problem introduced above, the customer service product family architecture would describe the required component interactions in the form of an event connector that allows variations along dimensions shown in Figure 2. It then becomes possible to describe both forms of required interactions - online transactions and batch updates - based on the event dimensions of notification and synchronicity. It is possible to describe the distribution profile of the interactions using the distributor dimensions delivery and addressing.

We are currently developing an infrastructure for using and experimenting with connectors for implicit invocation, real time communication and parallel execution. This infrastructure builds upon our previous work with event connectors [17].

4. Four-Way Refinement and Evolution

Having discussed an approach to modeling family architectures and instantiating them into product architectures, we now discuss how to refine the resulting product architectures into the individual product design. A product architecture constitutes an effective milestone [19] for any project since it can be analyzed and simulated to ensure the presence (or absence) of properties of interest. Nevertheless, it is still a difficult task to refine those architectural models into designs and actual implementations.

Refinement involves the creation of lower-level design models (and ultimately source code) and their continuous validation to ensure consistency. Refinement is hard and it often has to be done manually. This, however, implies that defects may be introduced while refining the product design from its architecture. Thus, we are faced with a major problem: we create the family and product architectures with the understanding that they describe certain desirable properties the end system should exhibit, however, if consistent refinement and evolution cannot be ensured, then there is no guarantee that the final product will indeed exhibit those properties. In other words, inconsistent refinement invalidates the purpose and usefulness of the product and family architecture. Meaningful architectural modeling *must* therefore ensure faithful refinement and evolution. This section will discuss an approach to improve the integrity of product models through the automation of refinement.

Automation during Refinement

The traditional way of using family architectural modeling involves instantiating a family architecture into a product architecture followed by refining that product architecture into a product design (Figure 3a – white area). This process may seem simpler in comparison with our proposed approach (Figure 3b – gray area) of using a family design, mainly because our approach additionally requires 1) modeling of a family design and 2) knowledge on how to relate it to the product architecture. However, we believe that these two additional activities ultimately simplify the overall refinement process. Using the traditional "family architecture to product architecture to product design" approach requires an instantiation from family architecture to product architecture and a refinement from the product architecture to a corresponding product design. The instantiation is relatively easy to do compared to the refinement activity, which is complicated by the lack of automation support. Even if the refinement could be automated, we would still be faced with the possibility of mismatch introduction at a later stage, e.g., when either the design or architecture is altered and those modifications are not properly propagated throughout the family.

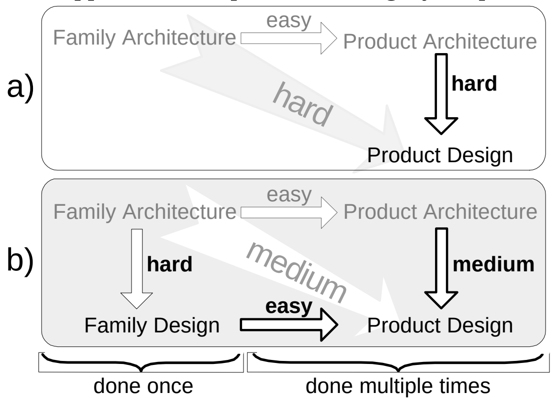

Figure 3. Two refinement approaches

Using a family design requires one additional instantiation activity – from a family design to a product design – which is again relatively straight forward. However, by replacing some (hard) refinement activities with (easy) instantiation activities, we achieve the added benefit of having to do less refinement which in turn, significantly simplifies doing product designs. Furthermore, we get the benefits of design reuse which complements architectural reuse (enabled through the use of family architecture). On the downside, arriving at a family design is not trivial or easily automated. The major advantage in using a family design is that we need to create it only *once* for each product family. Thus, once the family design is in place, each additional product can then be architected and designed much more easily (because of having simplified the refinement of product architectures). On the other hand, without a family design we may avoid the hard initial task of creating and instantiating it, but the task of refining product architecture would be harder. It is not difficult to see the return on investment of doing a one-time difficult task that simplifies a later repetitive task as opposed to avoiding that hard initial task but complicating the repetitive one. Therefore family design allows us to shift parts of the hard repetitive tasks from product architecture refinement to family architecture refinement.

Standard family designs can be used to realize different species of the same connector type. For example, design patterns for central dispatch, publish - subscribe and queued dispatch events can be described in the form of family designs that can realize the event connector, which can eventually be instantiated in the product designs, based on the specific mode of interaction required. Additional design patterns can describe the other dimensions. This technique requires integration of design patterns in the family design for specific values of the different dimensions into a single software product design.

Continuing our previous example, the use of events in the customer support product family architecture as the means of component interaction would leave a lot of flexibility in the design of individual products; the family architecture can support a large number of variations in each product. The product architecture can be used to instantiate an event connector by selecting the dimensions of each connector instance in the family architecture. This is an easy step, as it would involve looking up the taxonomy of connectors and making choices for the dimensions of a connector. Family designs can then provide standard refinements in the form of design patterns of event-based interaction for different platforms and middleware environments. The product design would then select specific design patterns for the target environment and desired

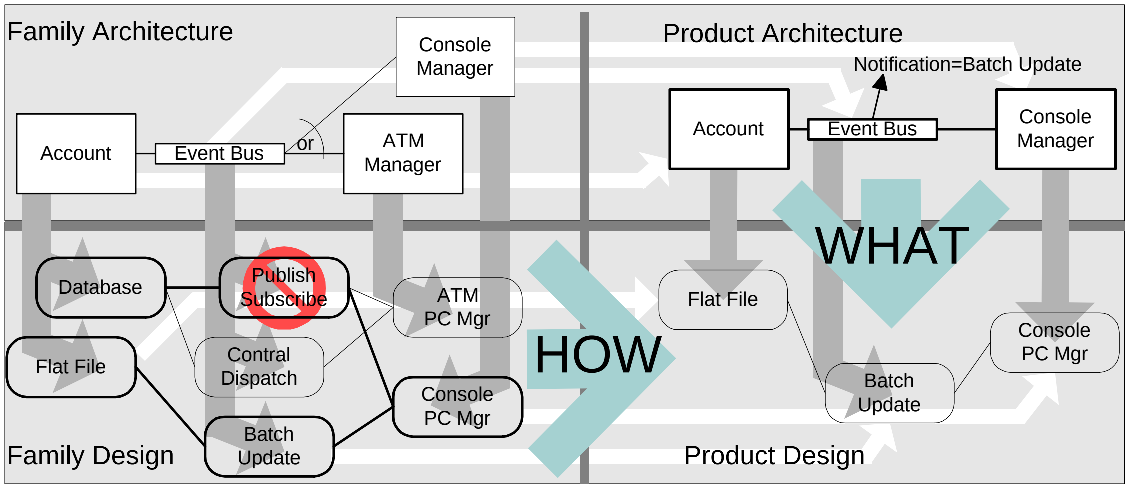

Figure 4. Example of instantiation and refinement

product properties of the system.

5. Example: Going from Family Architecture to Product Design

Figure 4 depicts a simple example on how to use generic connectors and the family design concept to generate a product design from a family architecture definition. The figure depicts a simple accounting family architecture (upper left) that supports access to accounting information via an event-based connector. This particular family architecture allows two types of interfaces, one for ATM machines and one for terminal consoles, but only one at a time. The family design (lower left) depicts possible realizations of above architectural elements. Note that there are realizations for both architectural components and connectors. Furthermore, we need to be able to deal with incomplete family design specifications including missing links (e.g., missing glue code) and missing realizations for some architectural elements. For instance, *Flat File* is a realization of *Account:* however, it does not work together with any realization of *Event Bus* (*publish-subscribe*, *control dispatch*, or *batch update*).

In our example we decided to instantiate a product architecture that consists of *Account*, *Event Bus*, and *Console Manager* (upper right). Using this product architecture as a reference and the family design as a resource database, we can now design and build that product using the predefined set of realizations. For instance, we could use the *Console PC Mgr* and either combine it with a *Publish-Subscribe* bus and a *Database* or combine it with a *Batch-Update* bus and a *Flat File*. When we specified the product architecture, we also specified some attributes the architectural connectors should demonstrate. For instance, we preselected the *Event* bus to be of the *Batch Update* style. With this additional information we can now automatically select a possible product design from the family design that would be compatible with the product architecture (lower right). The product architecture supplies information on *what* to design and the family design provides a details on *how* to design a product. Although this example enables full automation, we envision a possible need for human interaction.

6. Conclusions

This paper identified and addressed two significant challenges in product family development: modeling family architectures via generic connectors and supporting automatic architectural refinement via family designs. Our approach involves the use of a taxonomy of connectors to model the bounded ambiguity in family architectures. We do not claim that connectors are more important than components for enabling family architectural descriptions, however we found that connectors are significantly more flexible and adaptable than components.

To enable automatic refinement and evolution, we introduced the concept of family design. Family designs provide a set of realizations of architectural components and connectors (e.g., in form of design patterns). They simplify refinement by providing an additional path from a family architecture to a product design. We believe that combining product architecture and family design provides simplified and more precise refinement.

This paper also identifies issues requiring further work: creating a taxonomy of connectors, providing support for creating family designs and resolving mismatches among design and architectural views at the level of a product family.

Acknowledgements

The authors are thankful to Sandeep Phadke and Marija Rakic who provided useful feedback while discussing the topics presented in this paper.

References

[1] M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In *Proceedings of The Second International Conference on The Unified Modeling Language (UML'99)*, Fort Collins, CO, October 1999.

[2] ARES. *Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Navas del Marqués, Ávila, Spain, November 1996. http://hpv17.infosys.tuwien.ac.at/Projects/ARES/public/AWS/

[3] ARES II. F. van der Linden, editor. *Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Palmas de Gran Canaria, Spain, February 1998.

[4] ARES III. *The Third International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Palmas de Gran Canaria, Spain, February 2000.

[5] D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In Proceedings of AIAA Computing in Aerospace 10, San Antonio, 1995.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. The Unified Modeling Language User Guide. Addison-Wesley, 1998.

[7] A. Egyed and N. Medvidovic. A Formal Approach to Heterogeneous Software Modeling. Alexander Egyed and Nenad Medvidovic, submitted to FASE'2000

[8] R. Hayes-Roth and W. Tracz. DSSA Tool Requirements for Key Process Functions. ADAGE Technical Report, ADAGE-IBM-93-13B, October 1994.

[9] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. Accepted for publication in *IEEE Transactions on Software Engineering*, 1999. (To appear)

[10] D. E. Perry. Software Architecture and its Relevance to Software Engineering, Invited Talk. *Second International Conference on Coordination Models and Languages (COORD '97)*, Berlin, Germany, September 1997.

[11] D. E. Perry. Generic Descriptions for Product Line Architectures. In *Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families* (ARES II), Las Palmas de Gran Canaria, Spain, February 1998.

[12] The First Software Product Line Conference, August 28-31, 2000, Denver, Colorado, USA. http://www.sei.cmu.edu/plp/conf/SPLC.html

[13] N. Medvidovic and R. N. Taylor. Exploiting architectural style to develop a family of applications. In *IEE Proceedings Software Engineering*, Vol. 144 No 5-6, October 1997.

[14] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Upper Saddle River, NJ, 1996.

[15] N. Roodyn and W. Emmerich. An Architecural Style for Multiple Real-Time Data Feeds. *21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, May 1999.

[16] A. Carzaniga, E. Di Nitto, D. S. Rosenbloom and A. L. Wolf. Issues in Supporting Event-based Architectural Styles. *3rd International Software Architecture Workshop (ISAW3)*, Orlando FL, 1998.

[17] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead and J. E. Robbins. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 1996, 22(6), pp. 390-406.

[18] E. Di Nitto and D. Rosenbloom. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. *21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, May 1999.

[19] B. Boehm, Anchoring the Software Process, IEEE Software, July 1996.

[20] A. Egyed, "Integrating Architectural Views in UML," Qualifying Report, Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-99-514, http://sunset.usc.edu/TechRpts/Papers/usccse99-514/usccse99-514.pdf, 1999.

[21] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992, pp. 355-398.

[22] The Open Group, http://www.opengroup.org

[23] Object Management Group, http://www.omg.org

[24] Microsoft Corp. http://www.microsoft.com/com

[25] Sun Microsystems. http://java.sun.com/j2ee