Composition of Software Architectures from Reusable Architecture Patterns

H. Gomaa
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444
E-mail: hgomaa@gmu.edu

G.A. Farrukh
Technology Research Group
Science Applications International Corporation
Sterling, VA 20164
E-mail: farrukh@saic1.com

ABSTRACT

In this paper a software architecture perspective is taken to designing reusable software applications. An application domain is defined as a family of systems that have some features in common and others that differentiate them. During domain engineering, reusable specifications, architectural design patterns, and component types are developed, which capture the similarities and variations of the family of systems that compose the application domain. This paper describes the composition of software architectures from reusable feature based domain specific architectural design patterns.

Keywords
Software architectures, domain models, software reuse, design patterns

1. Introduction

At George Mason University, a project is underway to investigate software engineering life cycles, methods, and environments that provide software reuse at the requirements and design phases of the software life cycle in addition to the coding phase. The reuse-oriented Evolutionary Domain Life Cycle (EDLC) [2] is a highly iterative life cycle that takes an application domain perspective allowing the development of families of systems. Earlier research addressed the analysis and specification phases of the EDLC [2,3] and a domain modeling environment to support generating target system specifications from a domain model [4,5]. More recent research is investigating the design and implementation phases of the EDLC, with the goal of configuring executable target systems from the reusable software architecture and a library of predefined component types [6,7]. This paper describes the composition of software architectures from reusable feature based domain specific architecture patterns.

2. Domain Models, Domain Specific Software Architectures, Features, and Architecture Patterns

The terms used in this paper are defined as follows:
A *domain model* is an object-oriented analysis model for a family of systems, which explicitly models the similarities and variations in a family of systems [3]. A *domain specific software architecture* is an architecture for a family of systems that describes the composition of the family in terms of components and their interconnections. The architecture is described using an Architecture Description Language (ADL). Components are kernel, optional, or variant. A *domain model* is mapped to a *domain specific software architecture.*

A *feature* is an end-user requirement [3,5,9] that is supported in the *domain model* and *domain specific software architecture.* In a family of systems, features may be *kernel,* i.e., required by all members of the family, or *optional,* i.e., required by only some members of the family. Optional features may also be mutually exclusive. A feature may require another feature as a prerequisite.

A *design pattern* describes a problem to be solved, a solution, and the context in which that solution works [1,8]. The description is in terms of communicating objects and classes that are customized to solve a general design problem in a particular context. An *architecture pattern* is defined in this paper to be a set of interconnected components specified in terms of their interfaces and interconnections.

In this paper, an approach is described for developing reusable architectures using domain specific architecture patterns in conjunction with features. The architecture is composed of domain specific architecture patterns. A feature describes the problem that an architecture pattern solves. The solution given by the architecture pattern is a set of interconnected components, with a description of the components, their interconnections and their pattern of communication. The context in which the pattern solution

works is described in terms of the feature / feature dependencies, which are the constraints to be applied when combining features and hence architecture patterns to compose the architecture of a target system (one of the members of the family). Thus one feature may be a prerequisite for another or one feature may be mutually exclusive with another.

3. Domain Architectural Design

During domain architectural design, a reusable architecture is developed for the family of systems. Each object type in the domain model is mapped to a component type and each feature is mapped to a domain specific architecture pattern, showing the composition of interconnected components required to satisfy the feature.

Architectural description languages (ADLs) [13] separate the description of the overall system structure in terms of components and their interconnections from the description of the internal details of the individual components. In this paper, components types and architecture patterns are specified using an ADL, Darwin [11], which is part of the Regis configuration environment for parallel and distributed programming developed at Imperial College, London [12]. The Regis environment uses the Darwin ADL for the external specification of each component type, while the internals of component types are programmed in C++.

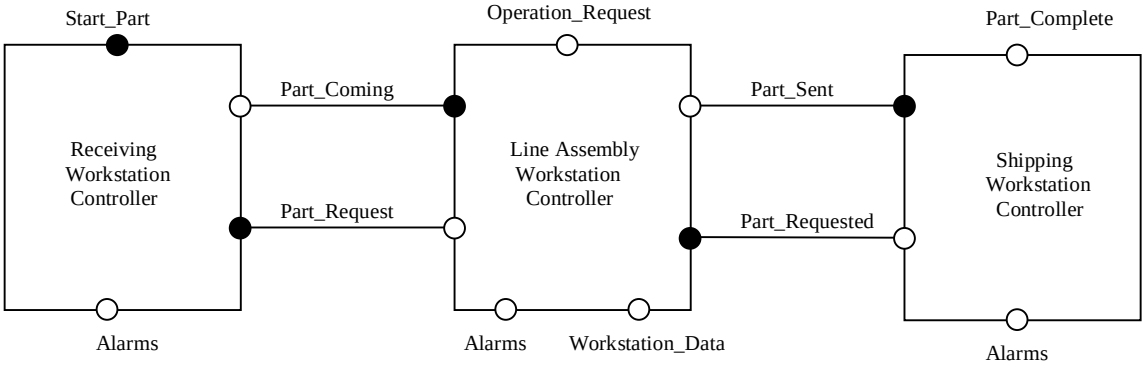
A Darwin component specification describes the external specification of the component type. For every simple kernel, optional, and variant object type in the domain model, an equivalent Darwin component type is developed. The component interfaces correspond to the object interfaces in the OCDs. Every Darwin component type is defined in terms of the interfaces it provides and requires from other Darwin component types. An example of a Darwin component type is:

component Line_Assembly_Workstation_Controller (char wkst_name)*
{
provide Part_Requested <port Part_Request_Type>;
provide Part_Coming <port Part_Type>;

require Workstation_Data <port Part_Type>;
require Part_Sent <port Part_Type>;
require Part_Request <port Part_Request_Type>;
require Operation_Request <entry OpReq_Type, OpRes_Type>;
require Alarms <port Alarm_Type>;
}

For every feature in the domain model, there exists an architecture pattern in the domain architectural design, which is specified using the Darwin ADL. There is also one architecture pattern supporting the kernel of the domain. The non-variant components and the interconnections between them are defined in the kernel architecture pattern. For each optional feature, an architecture pattern is developed, in which the optional and variant components needed to support it are defined as well as the interconnection between these components. In addition, interconnections are defined between the components in the architecture pattern and any kernel components used by these components. Interconnections are also defined to any optional or variant components defined in prerequisite architecture patterns. Specifically, an architecture pattern contains the following information:

- declaration of component types contained in this architecture pattern
- declaration of architecture patterns required by this pattern, including the kernel architecture pattern
- instantiation of components contained in this architecture pattern
- definition of component interconnections among components contained in this architecture pattern
- definition of component interconnections among components of this architecture pattern and components declared in required architecture patterns (i.e., prerequisite architecture patterns that this component depends on), including the kernel architecture pattern.



An example of a Darwin architectural description of an architecture pattern is given above. The High Volume architecture pattern has three component types: Receiving Workstation Controller, Line Assembly Workstation Controller, and Shipping Workstation Controller. The instantiation statements are used to create one or more instances of a component type:

inst
Receiving_Wkst: Receiving_Workstation_Controller;
Shipping_Wkst: Shipping_Workstation_Controller;
Line_Wkst:
Line_Assembly_Workstation_Controller(wkst_name);

Communication between two components in the pattern involves connecting the *require* interface of the sending component to the *provide* interface of the receiving component using the Darwin bind statement. For example, the *Line Assembly Workstation Controller* sends the *Part Sent* message to the *Shipping Workstation Controller*:

bind Line_Wkst.Part_Sent -- Shipping_Wkst.Part_Sent;

This pattern depends on two other architecture patterns: the kernel pattern, which contains the kernel components, and the Factory Production pattern, which contains two other components. Dependencies on components in required patterns must be explicitly defined. Communication between the *Line Assembly Workstation Controller* component in this pattern and the kernel *Alarm Handling Server* component contained in the kernel architecture pattern is also specified using a bind statement:

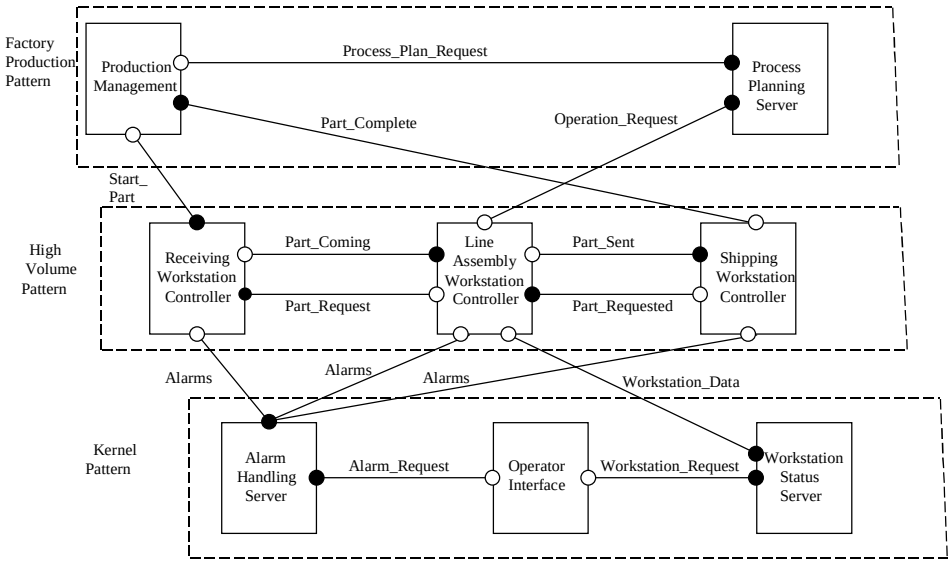
bind Line_Wkst.Alarms -- Alarm_Handler.Alarms;

Similarly, a bind statement is needed between the *Line Assembly Workstation Controller* component in the High Volume Manufacturing pattern and the *Process Planning Server* component in the prerequisite Factory Production pattern:

bind Line_Wkst.Operation_Request - Process_Planner.Operation_Request;

4. Target System Architecture Composition

To compose a target system architecture, the user selects the optional features desired for the target system, subject to the feature/feature constraints. Automated support is provided for this process [5]. Once all the features have been selected for a target system, the target system architecture is composed from the corresponding architecture patterns. For a High Volume Manufacturing system, the High Volume, Factory Production, and kernel architecture patterns are needed, as shown below. The High Volume target system architecture consists of the composition of these three patterns involving the instantiation and interconnection among the components of the patterns as given below. In particular, note the interconnection between the components in the High Volume pattern with the components in the prerequisite patterns, as described above. For a Flexible Manufacturing target system, two of the three architecture patterns, the Factory Production and kernel architecture patterns, are reused.



Target system architectures are composed from domain specific architecture patterns, where the constraints for interconnecting architecture patterns are given by the feature/feature dependencies. Thus the relationship among domain features, which is defined during domain analysis, is preserved as constraints among architecture patterns. For example, the High Volume feature requires the kernel and Factory Production features, and correspondingly the High Volume architecture pattern requires the kernel and Factory Production architecture patterns. As both High Volume and Flexible Manufacturing systems require the kernel and Factory Production architecture patterns, components in these architecture patterns are reused in different target systems. Furthermore, the High Volume and Flexible Manufacturing architecture patterns are constrained to be mutually exclusive by the feature/feature dependencies.

5. Conclusions

This paper has described an approach for composing reusable software architectures from feature based domain specific architecture patterns. The architecture is composed of domain specific black-box architecture patterns. A feature describes the problem that an architecture pattern solves. The solution given by the architecture pattern is a set of interconnected components, with a description of the components, their interconnections and their pattern of communication. The context in which the pattern solution works is described in terms of the feature / feature dependencies, which are the constraints to be applied when combining architecture patterns to compose the architecture of a target system.

As part of this research, a software engineering environment has also been developed [7], which integrates the domain modeling environment [4,5] with the Regis distributed configuration and programming environment [11,12]. Using this environment, an executable target system can be composed from a reusable software architecture and a library of predefined component types.

6. Acknowledgments

The authors gratefully acknowledge several valuable discussions with J. Kramer and J. Magee on using Regis for this research. This research was supported in part by NASA Goddard Space Flight Center, the Virginia Center of Innovative Technology and DARPA.

7. References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[2] H. Gomaa, "A Reuse-oriented Approach for Structuring and Configuring Distributed Applications", *Software Engineering Journal*, March 1993.

[3] H. Gomaa, "Reusable Software Requirements and Architectures for Families of Systems", *Journal of Systems and Software*, April 1995.

[4] H. Gomaa and L. Kerschberg, "Domain Modeling for Software Reuse and Evolution," *Proc. IEEE CASE '95*, Toronto, Canada, July 1995.

[5] H. Gomaa, et. al. "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," *J. Automated Software Engineering*, Vol. 3, Nos. 3/4, August 1996.

[6] H. Gomaa and G.A. Farrukh, "An Approach for Configuring Distributed Applications from Reusable Architectures, *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, Montreal, Canada, Oct. 1996, pp. 442-449.

[7] H. Gomaa and G. Farrukh, "Automated Configuration of Distributed Applications from Reusable Software Architectures", Proceedings IEEE International Conference on Automated Software Engineering, Lake Tahoe, November 1997.

[8] R.E. Johnson, "Frameworks = (Components+Patterns)", *CACM*, Vol. 40, No. 10, October 1997, pp. 39-42.

[9] Kang K.C. et. al., "Feature-Oriented Domain Analysis", *Technical Report No. CMU/SEI-90-TR-21*, Software Engineering Institute, November 1990.

[10] J. Kramer, J Magee, M Sloman & N. Dulay, "Configuring Object-based Distributed Programs in REX", *Software Engineering Journal*, March 1992.

[11] J. Magee, N. Dulay, and J. Kramer, "Structuring parallel and distributed programs," *Software Engineering Journal*, March 1993, pp. 73-82.

[12] J. Magee, N. Dulay and J. Kramer, "Regis: A Constructive Development Environment for Parallel and Distributed Programs", *J. Distributed Systems Engineering*, 1994, pp. 304-312.

[13] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.