Compositional Development from Reusable Components Requires Connectors for Managing Both Protocols *and* Resources

Gul A. Agha
Department of Computer Science
Univ. of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
Phone: (217) 244-3087 Email: agha@cs.uiuc.edu

1 Introduction

Current component-based approaches for software architecture factor a system into a set of components, which encapsulate computation, and a set of connectors, which describe how components are integrated into the architecture. This separation of design concerns favors a compositional approach to system design; a methodology which is particularly important when specifying architectures for open (*e.g.* web-based) distributed systems. Heterogeneity, failure, and the potential for unpredictable interactions yield evolving systems which require complex management policies. Allowing architectural specifications in which these policies are separated into abstract connectors has clear advantages for system design, verification and reuse.

Note that policies for managing distributed systems (*e.g.* reliability protocols, load balance and placement, security constraints, coordination, etc.) not only assert properties on the connections between component interfaces, but must also enforce constraints on how resources are allocated to components. For example, a reliable server may be developed by adding a backup to an existing server and installing an instance of the primary backup protocol. In addition to recording interactions at the backup, the primary backup protocol must also ensure that the backup and server use separate, failure-independent resources (*e.g.* they must execute on separate processors). While component-based models have made great strides in recent years, we argue that the perceived gap between current middleware systems and the demands of large scale distributed applications is due in part to inadequate component models and the inability to define more flexible connectors that manage both resources and interactions. Similarly, we claim that architectural design from reusable components will remain an unfulfilled goal until architectural connectors become a mechanism for fitting components to architectural contexts, rather than defining interconnections between component interfaces.

Our argument is based on the observation that component interfaces abstract over functionality but not resource management and that therefore, connectors are limited in their ability to specify system-critical properties of distributed systems. We describe key aspects of our argument and potential solutions in the remainder of this paper.

1.1 Component Interfaces Have Limited Abstraction

Current notions of component interfaces are based on a functional representation of the services provided by a component. This abstraction is a natural extension of the object model. However, when placing an object in an architectural setting, this model fails to describe many important features such as:

- **Locality properties:** The distribution and communication behavior of internal computational elements.
- **Resource usage patterns:** Distinctions such as computation bound versus i/o bound elements, degree of concurrency, and the resources corresponding to critical and transient state.
- **Existing lower-level constraints:** The policies defined by sub-connectors in the case of components which encapsulate collections of sub-components.

In general, components should provide comprehensive model of *architectural context*: the relationships between component behavior and architectural features such as those described above. A natural solution would be to extend current interfaces with additional functional entry points for selecting, for example, placement policies, reliability features (*e.g.* fault-tolerance protocols), and so on. However, such an approach complicates component code by embedding orthogonal, context-specific concerns. The more preferable approach would be to design generalized components which may be customized to particular architectural contexts. Connectors would encapsulate these customizations, preserving compositional system development.

1.2 Connectors Not Composable Property Specifications

The structure of component interfaces has limited connectors to representing protocols between interfaces. As described above, however, this limitation prevents the design of even a simple primary backup protocol. We believe that connectors should have the role of enforcing *properties* over a collection of components, rather than adapting their interactions to one another. Given a more flexible model of components, connectors may specify both the individual protocols which govern interactions as well as global policies which control how components are deployed in a particular architectural setting.

In addition to serving as policy specifications, connectors may also be required to compose. Moreover, the policies required for a particular interaction may not be determined until run-time. For example, future web-based applications, in which the potential set of clients is not known until run-time, may require different collections of policies depending on the client. That is, policies may be required to compose dynamically. Note that connector composition differs from component composition. In particular, while components compose by connecting their interfaces, connector composition consists of multiple policies simultaneously applied to a collection of components. For example, where interactions are concerned, connector composition may be modeled as a protocol stack applied to the endpoints of a particular interaction. The composition of resource management policies, on the other hand, may be modeled as a collection of constraints applied to the resource acquisition behavior of a component. Understanding and designing mechanisms for connector composition is a critical pre-requisite for achieving software development from reusable components.

2 Research Directions

In order to explore solutions to the problems described above, we suggest two directions for further research: a more descriptive model of component computation, and a comprehensive meta-model which describes how distributed management policies may be used to customize the underlying component computation model.

2.1 A Uniform Computational Model for Components

In order to reason about architectural context, we require a model of component computation which represents component behavior in terms of interactions with a set of default system services. Relative to computational behavior, the semantics of these services will remain the same regardless of architectural context. However, the semantics observed by the *implementation* of these services will vary as components are placed in different architectures. This distinction allows compositional development in which generalized components are fitted to particular architectures not by changing their computational behavior (which would break encapsulation) but by customizing the architectural implementation of underlying services.

We propose the Actor model [1, 2] as a candidate for describing component behavior. Actors provide a general and flexible model of concurrency which may be used as an atomic unit for building typical architectural elements including procedural, functional, and object-oriented components. Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors coordinate by asynchronously sending messages to one another. Each actor has a unique *mail address* and a *mail buffer* to receive messages. Communication is point-to-point and is assumed to be weakly fair. Actors require three services from the underlying system: the ability to *send* messages asynchronously to other actors; the ability to *create* actors with specified behaviors; and the ability to become *ready* to receive the next message.

Components may be modeled as encapsulated collections of actors in which a distinguished subset, called *liaisons*, are used for interactions with other components. Interactions between liaisons correspond to current notions of component connection. In particular, by customizing these interactions, specific protocols may be enforced. Moreover, the architectural context of a component is represented by the service invocation behavior of internal (*i.e.* non-liaison) actors. Thus, the collective behavior of a component relative to architectural features is captured by the interactions through its liaisons and the resource access patterns of its internal actors. Both of these behaviors are represented uniformly in terms of invocations of the basic actor services described above, providing a clean representation for architectural customization.

2.2 Connectors as Protocol and Policy Specifications

By defining component behavior in terms of the invocation of basic system services, we allow flexible connectors which customize both interactions and resource management. In particular, we may model system services in terms of a meta-architecture which describes the implementation of these services on a per-actor basis. A connector then consists of a collection of meta-level behaviors that, when installed on a set of component actors, redefine the basic system services provided to those actors. By defining a notion of meta-level composition, we may compose connectors and thus enforce multiple policies on a collection of components.

Liaisons are the only externally visible elements of a component. Thus, connectors which specify protocols between components are naturally represented in terms of customizations applied to individual liaisons. However, connectors which specify resource management policies are more challenging because they customize internal component elements. In particular, we would like to specify arbitrary customizations of internal actors while respecting the encapsulation properties of a component. To this end, connectors are constructed from two types of meta-level behavior:

- **Roles:** A role is a specific customization applied to one or more liaisons. Roles are used to implement protocols on connections between components. For example, an encryption protocol may be implemented by customizing the "send" behavior of one liaison (*e.g.* to encrypt outgoing messages) and the "receive" behavior of another (*e.g.* to decrypt incoming messages). Roles are installed explicitly on a set of liaisons.
- **Context:** A context is a single meta-level behavior which customizes *all* actors within a component and is automatically installed on any dynamically created actors. Contexts are used to manage the allocation of resources. For example, a local load balancing strategy may be implemented by customizing the "create" behavior of all actors within a component.

3 Conclusion

Modeling components as hierarchical collections of actors provides a flexible mechanism for specifying arbitrarily concurrent, local computation while restricting remote communication to adhere to a well-defined interaction mechanism. The interface of a component is represented by a dynamic set of liaisons. Thus, component interfaces may change in response to run-time constraints by creating new liaisons to handle new connections.

While components describe an architecture at a functional level, we view connectors as lower level abstractions which define how an architecture is deployed in a particular execution environment. By accessing an open implementation of the interface between actors and their underlying system services, connectors implement transparent customizations of component behavior. We factor customizations into two categories: roles are explicit customizations of liaisons, while contexts are implicit customizations of all actors within a component. Roles allow the enforcement of interaction policies over connections between components. Contexts support component-wide resource management and coordination. Composition allows multiple connectors to be applied to a single component.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation.* MIT Press, 1986.

[2] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions. Chapman & Hall, 1997.