# Role Based Programming Systems

**James C. Browne, Kevin Kane and Nasim Mahmood**

The University of Texas at Austin
Department of Computer Sciences
1 University Station C0500
Austin, Texas 78712-0233
{browne, kane, nmtanim}@cs.utexas.edu

**Abstract**

This paper describes a programming model where programs are generated by composition of components based on the roles the components play in the execution of the program and two instantiations of this programming model. In this context, roles are specifications of the properties, behaviors and modes of interaction (semantics) of the components and the requirements of the components upon their environment. Components and their roles are specified in terms of ontologies derived from domain analyses of the application areas for the programs. The relationships between roles, naming models, programming models, coordination models and compositional development are explored.

## Ontologies and Roles in Programming Models and Systems

This paper takes the perspective that a system is an entity which is composed from other entities to accomplish some goal. The entities which are to be composed each play some role or roles in the accomplishment of the goal of the system. This paper discusses and illustrates the application of roles in programming models where the system is a program and the entities which are composed to form the program are components. In the context of programming systems, roles are specifications of the properties and behaviors (semantics) of components. A software architecture is a specification of the structure of a system in terms of roles. Program development becomes selection of entities which realize the roles in the architecture. Roles are specified in terms of ontologies for the domains of the systems which are to be realized. In the context of programming models and systems an ontology defines the functional and interaction behaviors of components of a system and a set of attributes in which the properties and behaviors of the components and the relationships/interactions among the components can be specified. Previous work on programming models and systems (Bayerdorffer 1995), (Browne, Kane, and Tian 2002), (Mahmood, Deng, and Browne 2003), (Kane and Browne 2004), (Mahmood, Feng, and Browne 2005a), (Mahmood, Feng, and Browne 2005b) are given as illustrations of the use of role concepts in programming models. The breadth of applicability of the role concept is suggested by the fact that papers on these systems have been published in five different research areas (naming models, parallel programming, high performance computation, coordination models and languages and performance evaluation).

The next section of the paper gives the fundamental concepts underlying the programming systems and how roles are defined and used. Following this the two programming systems and their applications are defined. Then the relationships we have recognized between our research in programming models and systems and other applications of roles are briefly discussed. A short summary of programming models and systems is given. The paper concludes with a discussion of the directions of future research.

## Role Specifications for Component-based Programming

The programming models and systems used here as illustrations of the application of roles in programming models and systems originated in a study of naming models in programming systems (Bayerdorffer 1993). A lattice taxonomy based on an ontology of naming models was developed. The metric for ordering the lattice was the set of communications which could be directly implemented as atomic operations. The most powerful naming model in the lattice, associative broadcast, was based on coupling dynamically binding "descriptive names" to the entities being named with a broadcast-based search and resolution mechanism. A "descriptive name" is equivalent to the specification of the current role being played by the entity in the system or coordination set. This identification will be described later in this section. A programming system based upon associative broadcast was reported (Bayerdorffer 1995).

### Role Specifications - Associative Interfaces

A role specifies the services and behaviors of a component and the dependencies of a component on other components. The behaviors and dependencies together determine the role or roles that component can play in a composition or coordination system. The first requirement is a language in which to specify the roles of components. We specify roles in terms of an interface definition language which is used to encapsulate an executable segment of code. These role specifications are called associative interfaces because they enable association, or composition, of components on the basis of their roles.

**Component**: A component is an entity which fulfills one or more requirements, and plays a possibly dynamic set of roles, for the execution of a system of which it is a part.

A role specification, which we implement as an associative interface),consists of an **accepts** specification and a **requires** specification.

**Accepts Specification**: An accepts interface specifies the set of interactions in which a component is willing to participate. The accepts interface for a component is a three-tuple (profile, coordination/interaction, protocol).

- A profile incorporates the information necessary to characterize the properties and behaviors of a component and to enable the compositional mechanism to select components meeting the requirements for efficient implementation of a given instance of an application family for a given execution environment. A profile may dynamically change during execution to reflect changes of state of a component. A profile is a set of attribute/value pairs.
- The coordination/interaction of components is managed by a state machine implemented in the profile each component. Each state of the state machine incorporates is a guarded command with a condition for the execution of the function and a function signature including the data types, functionality and parameters of the unit of work to be executed and a specification for the component state in which execution of this function is enabled. The state machine is defined in the form of expressions in a linear propositional temporal logic over the attributes and state variables of the component. A function signature and its enabling condition are called a transaction. A transaction can be enabled or disabled based on its current state and its current state can be used in runtime binding of the components. The state machine can be used to represent complex interactions such as precedence of transactions, "and" relationships among transactions and "or" relationships among enabling states and transactions.
- A protocol defines a sequence of simple interactions necessary to complete the interaction specified by the profile. The most basic protocol is data-flow, which is defined as executing the functionality of a component and transmitting the output to a set of successors defined by the requires interface at that component without returning to the invoking component.

**Requires Specification**: A requires interface specifies the set of interactions which a component must initiate if it is to complete the interactions it has agreed to accept. The requires interface is a set of three-tuples (selector, transaction, protocol). A component can have multiple tuples in its requires interface to implement its required functionality.

- A selector is a conditional expression over the attributes of the components in the domain.
- Transaction specifications are similar to those for accepts specifications except that the state machine is a single state.
- Protocol specifications are as given for accepts specifications.

**Start Component**: A start component is a component that has at least one requires interface and no accepts interface. Every program requires a start component. There can be only one start component in a program which provides a starting point for the program.

**Stop Component**: A stop component is a component that has at least one accepts interface and no requires interface. A stop component is also a requirement for termination of a program. There can be more than one stop component of a program denoting multiple ending points for the program. A non-terminating program may reach a stop node only in the case of an error or fault.

## Composition and Coordination

The conditional expression of a selector is a template which has slots for attribute names and values. The names and values are specified in the profiles of other components of the domain. Each attribute name in the selector expression of a component behaves as a variable. The attribute variables in a selector are instantiated with the values defined in the profile of another component. The profile and the selector are said to match when the instantiated conditional expression evaluates to true.

Composition is usually defined in terms of binding invocations of a method or procedure by one component to a method or procedure in an implementing component. Composition may, although the mechanisms differ, occur at compile time, at link/load time, and at run time such as through RPC. Composition at compile time and/or link time is accomplished through associative search of libraries as defined in the first paragraph of this section.

Communication/interaction and thus coordination among components at runtime is by (conceptually) broadcasting messages where the address of the message is a selector clause from the requires interface. This communication model is called associative broadcast. The selector in each broadcast message is locally matched with the profile of each object in the broadcast domain. Messages are received only by those components whose profiles cause the selector of the message to evaluate to true. Matching of a selector to a component profile is atomic with respect to a profile's modification. It is therefore possible for a sender's target set to change during transmission if a profile is changed, but the matching criteria will not change for a component once matching begins. Matching is done locally and asynchronously upon "arrival" of the message. In practice, associative naming and binding are used as a compilation process to select an initial set of bindings among components prior to runtime.

Both compile time and runtime composition are based on the specification of roles in the form of associative interfaces and upon matching of the requirements of one component with a role played by some other component. Therefore the associative model of interaction unifies coordination and composition.

## Programming Model

The programming model has two phases: development of families of components and specification of the instances of the family of programs which can be instantiated from the sets of components.

### Component Development

A role may be realized by a several different components. The choice of which component to use to fulfill the role may depend on the implementation of the component. Therefore, role specifications are extended to include non-functional properties such as performance or possible parallel execution. The set of components which enables construction of a family of application programs may include components which utilize different algorithms for different problem instances or different implementation strategies for different execution environments. A program for a given problem instance or given execution environment is composed from appropriate components by selecting desired properties for the components and the properties of the execution environment in the Start component.

The steps for developing components are:

a. Ontology Development – Execute the necessary domain analyses. It is usually the case that applications require components from multiple domains.
b. Component Development – Specify and either design and implement or discover in existing libraries, the family of components identified in the domain analysis in an appropriate sequential procedural language.
c. Encapsulation – Encapsulate the components with associative role specifications. The interfaces must differentiate the components by identifying their properties in terms of the attributes defined in the domain analysis.

### Program Instance Development

The steps in specifying a given instance of an application are:

a. Analyze the problem instance and the target execution environment. Identify the attributes and attribute values which characterize the components desired for this problem instance and execution environment.
b. Construct an architecture for the instance of the program in terms of roles extended with non-functional properties.
c. Identify the components from which the application instance will be composed. If the needed components are not available then some additional implementations of components may be necessary together with an extension of the domain analysis.
d. For parallel programs or programs to be executed in unreliable execution environments, specify the number of replications desired for parallelism and for fault-tolerance. Incorporate these specifications into the component interfaces or as parameters in the Start component if parameterized parallelism has been incorporated into the component interfaces.
e. Define a Start component which initializes the replication parameters, sets attribute values needed to ensure that the components implementing the appropriate implementations of roles are selected and matched.
f. Define at least one Stop component.

## Instantiations of Role-Based Programming

This section sketches two instantiations of programming systems utilizing role-based composition and coordination: P-COM$^2$ (Mahmood, Deng, and Browne 2003) and CoorSet (Kane and Browne 2004).

### P-COM$^2$

The principal elements of P-COM$^2$ are the interface definition language in which roles are defined as associative specification, a compiler which automatically selects components whose roles compose to implement a given system as previously described and a runtime system which enables composition to continue at runtime by substituting components which have different implementations of a given role.

The principal concerns and goals for the P-COM$^2$ project have been:

- to enable automation or at least partial automation of composition of programs from components,
- to integrate compile time and runtime composition of components,
- to generate parallel and distributed programs adapted to the problem instance and execution environment and
- to enable performance-oriented, evolutionary development of parallel and distributed programs.

The first and third goals are discussed in (Mahmood, Deng, and Browne 2003). The second goal is addressed in (Mahmood, Feng, and Browne 2005a) and the fourth goal is addressed in (Mahmood, Feng, and Browne 2005b).

The source program for the P-COM$^2$ compilation process is a start component with a sequential computation which implements initialization for the program and a requires interface which specifies the components implementing the first steps of the computation and one or more libraries to search for components which play the roles defined in the domain analysis. The set of components which is composed to form a program is primarily dependent on the requires interface of the start component.

The target language for the compilation process is a generalized data flow graph as defined in (Newton and Browne 1992). A node in this data flow graph consists of an initialization, a firing rule, a sequential computation and a routing rule for distribution of the outputs of the computation. There are two special node types, a start node and a stop node. Acceptable data flow graphs must begin with a start node and terminate on a stop node. The nodes of the data flow graph are the components, the arcs result from matching of accepts and requires interfaces and the firing rules are derived from the state machines associated with the accepts specification.

Components can be replaced at runtime by a runtime modification of a requires interface. The runtime system, when it encounters the need to link a new component, utilizes the dynamic linkers available in most operating systems. Details can be found in (Mahmood, Feng, and Browne 2005a).

Roles also enable composition of programs where some components are abstract models such as performance models and other components are concrete implementations of full functionality. The specification for the roles played by a component is extended to include “abstract” or “concrete” and the compiler and runtime system extended accordingly. Details can be found in (Mahmood, Feng, and Browne 2005b).

## Coordination Models and CoorSet

The goal of this application of roles is to enable development of distributed applications where control is fully distributed. It is assumed the components from an application or family of applications is to be developed are located at different sites in a distributed system and are active. Therefore, coordination and composition of application becomes search to identify the needed components and then coordination among the components. The goal was accomplished in two steps. (Browne, Kane, and Tian 2002) reports a coordination model based on associative interface based specification of roles. A subsequent paper (Kane and Browne 2004) extends the coordination model to enable convenient programming of distributed applications and presents a language for specification of interfaces and initial system configurations.

The CoorSet programming system has four elements: an interface definition language in which to specify roles, a runtime system implementing associative broadcast, a compiler for generation of initial conditions and a support system for creating distributed configurations of components.

The coordination model will be referred to as Associatively Specified Interactions (ASI). ASI is a model of coordination amongst a group of components. A component is an implementation of a role required for the execution of the system. The ASI implementation of a component is one or more functions encapsulated by an interface which specifies the role of the component and its implementation. In contrast to P-COM$^2$ where components are explicitly linked, all interactions are through associatively broadcast messages. In the ASI protocols, a target set specification (the selector) travels with each message that is broadcast onto the network. The target set is determined for each message by the recipients whose local state satisfies the target specification just as described for the compilation in P-COM$^2$ but the bindings are transient. The sender does not know the membership of this set, and does not necessarily ever discover it.

Broadcast communication is unique in that it directly implements exhaustive search. It also can be shown to enable attainment of consensus in a distributed system under asynchronous execution. Protocols implementing acknowledgements can be implemented if needed for a given coordination problem.

The ASI coordination model has been extended to include the same data flow semantics as is used in the target language of P-COM$^2$. The model thus incorporates two additional features: complex conditions for enabling execution of a component and replication for both representation of SPMD parallelism and fault-tolerance.

These features are described (somewhat redundantly) here since they now appear in a data flow graph representation in the CoorSet interface definition language.

The conditions for executing and action of a component commonly include receipt of multiple messages. To maintain separation of concerns it is necessary to incorporate this requirement into the coordination model. We introduce the concept of a "firing rule" directly into the coordination model. A firing rule is a specification of the set of messages which must be received to initiate any action of a component. Additionally, since components may and often will have persistent state, there may be precedence relations among possible enabling message sequences. These extensions are accomplished by adding types to messages and incorporating a conditional expression over message types and local state into the associative interface.

The definition of firing rules used in the extended coordination model is taken from a data flow programming model (Newton and Browne 1992), where rather than waiting on a single input, a node in a data flow graph waits on multiple inputs, possibly in a particular order, before becoming enabled for execution. Firing rules are specified with a Java-like logical syntax. Specifying reception of *either* of two message types *R, S* is done with a rule "R || S". Reception of *both* of two message types is specified with a rule "R && S". Reception of *R followed by S* is specified with a rule "R < S". These rules can be compounded and grouped with parentheses, such as "(R < S) || (R < T)". The '<' operator has the lowest precedence, followed by `||', and `&&' has the highest precedence.

Replication is another feature that must be included in associative interaction specifications to enable facile specification of parallelism and fault-tolerance. SPMD parallelism can be readily implemented by replication of components. Replication of functionality for fault-tolerance can be made transparent and synchronization-free after initialization. If an initiating component starts several replicas of a given component to insure success in an unreliable environment and each of the replicated components responses by associative broadcast then the initiating component can safely proceed after the first successful result and set its profile to ignore the other completions. A component can be replicated by adding an index attribute to its profile and instantiating replicas in conformance to the index range. Once specified, a component can be started an arbitrary number of times. The runtime system provides unique identifiers in a predictable way so replicas can alter their behavior, or they can all execute in the same way depending on the needs of the application.

CoorSet has been applied to implementation of distributed computations including a distributed computation of Google's page rank (Sankaralingam, Sethumadhavan, and Browne 2003).

## Relationship of Role-Based Programming to other Uses of Roles

The most directly related research we have found is (Guarino 1998). Guarino constructs ontologically based relationships among components which can be composed to implement software systems. The possibility of ontologically based composition is discussed in the context of both "development" time and runtime. This work is entirely conceptual and reports no implementation or application.

### Coordination Models

The most direct relationship is with tuple-space based coordination models (Gelertner 1985). The profile contains the match variables of a tuple. Selectors play the role of tuple templates. Each process or component which is activated by a selector/tuple match may send messages as a result of the action it executes which may bind it to other components. Thus associative broadcast interactions can viewed as being based on implementation of a distributed tuple space with active linked tuples. Also the tuple space communication model also lies at the top of the lattice taxonomy of communication models. Finally, note that Linda was initially introduced as a parallel programming system.

### Other Role-Based Programming Systems

There are at least three active research communities in programming systems implicitly using role-based specifications for components: Web services, Grid systems, and high performance computing (Armstrong and Gannon 1999), (Bernholdt et. al. 2002), (Kane and Browne 2004).

The CoorSet programming model is equivalent to a production rule system (Wu, Miranker, and Browne 1996) where the entities in the working store are distributed and the search for entities which match rule is implemented through associative broadcast.

### Machine Learning

The characterization of entities used in machine learning and questioning answering systems (Clark and Porter 1997) have a startlingly similar format and content to the associatively specified roles used in P-COM$^2$ and CoorSet. The process of forming question answering trees where each "fact" plays its role in answering the question is a search similar to the search made for a component satisfying a role in a computation.

### Software Architecture and Composition Languages

Software architectures (Perry 1989) implicitly use roles to identify the structure of the systems defined by an architecture. In some cases, attributes are used to define roles. For example, ArchJava (Aldrich, Chambers, and Notkin 2002) annotates ports with provides and requires methods which helps the programmer to better understand the dependency relations among components by exposing it to the programmer. Darwin (Magee, Dulay, and Kramer 1993) is a composition and configuration language for parallel and distributed programs. Darwin uses a configuration script to compose programs from components.

### Networking

Intentional names (Schwartz 1999) which are used in networks to support routing and search specify roles by using static forms of associatively specified names.

## Conclusions and Future Research

Role specifications can be viewed as specifications for the semantics of a component. Use of role specifications is the critical enabler for migration of programming to component composition. Without effective machine usable specification of semantics at the component level (which are provided by roles) programming by component composition cannot be attained.

We are extending the use of role specifications to include subsumption and containment relations on role specifications to enable more powerful selection of components for composition and coordination. We are also embedding role-based interactions in the SETI@home model of computation to extend it to a more general class of computations.

## Acknowledgments

This research was partially supported by NGS NSF grant number 0103725 “Performance-Driven Adaptive Software Design and Control”, NGS NSF Grant Number: ACI-0305644 “Montage: An Integrated End-to-End Design and Development Framework for Wireless Networks” and NSF Grant Number 0438967 “SoD Collaborative Research: Constraint-based Architecture Evaluation”

## References

Aldrich, J.; Chambers, C.; and Notkin, D. 2002. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 22nd International Conference on Software Engineering,* 187-197. International Conference on Software Engineering, Inc.

Armstrong, R. et al. 1999. Toward a Common Component Architecture for High-performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, 115-124. New York: Institute of Electrical and Electronics Engineers.

Bayerdorffer, B. 1993. Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems, Ph.D. diss., Dept. of Computer Sciences, University of Texas at Austin.

Bayerdorffer, B. 1995. Distributed Computing With Associative Broadcast. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences.* New York: Institute of Electrical and Electronics Engineers.

Bernholdt, D. et al. 2002. A Component Architecture for High-Performance Computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*.

Browne, J. C., Kane, K. and Tian, H. 2002. An Associative Broadcast Based Coordination Model for Distributed Processes. In *Proceedings of COORDINATION 2002,* LNCS 2315, 96-110. Berlin, New York: Springer-Verlag.

Clark, P., and Porter, B. 1997. Building Concept Representations from Reusable Components. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI’97)*. Menlo Park: AAAI Press.

Gelertner, D. 1985. Generative communication in Linda. *ACM Transactions on Programming Language Systems*. 7(1):80-112.

Guarino, N. 1998. Formal Ontology and Information Systems. In *Proceedings of FOIS’98*, 3-15. IOS Press.

Kane, K., and Browne, J.C. 2004. CoorSet: A Development Environment for Associatively Coordinated Components. In *Coordination Models and Languages, Proceedings of COORDINATION 2004*. Berlin, New York: Springer-Verlag.

Magee, J.; Dulay, N.; and Kramer, J. 1993. Structuring parallel and distributed programs. *Software Engineering Journal* 8(2): 73-82.

Mahmood, N.; Deng, G.; and Browne, J.C. 2003. Compositional Development of Parallel Programs. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing (LCPC’03)*. Berlin, New York: Springer-Verlag.

Mahmood, N.; Feng, Y.; and Browne, J.C. 2005a. A Case Study in Application Family Development by Automated

Component Composition: h-p Adaptive Finite Element Codes. In *Proceedings of the International Conference on Computational Science 2005*. Berlin, Hong Kong: Springer-Verlag. Forthcoming.

Mahmood, N.; Feng, Y.; and Browne, J. C. 2005b. Evolutionary Performance-Oriented Development of Parallel Programs by Composition of Components. In *Proceedings of the 5th International Workshop on Software and Performance*. Forthcoming.

Newton, P., and Browne, J. C. 1992. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the ACM International Conference on Supercomputing*. New York: ACM.

Perry, D. 1989. The Inscape Environment. In *Proceedings of the 11th ICSE*, 2-12. New York: IEEE Press.

Sankaralingam, K.; Sethumadhavan, S.; and Browne, J.C. 2003. Distributed Pageranks for P2P Systems. In In *Proceedings of the Twelfth IEEE International Symposium on High Performance Parallel and Distributed Systems,* 58-69. New York: Institute of Electrical and Electronics Engineers.

Schwartz, E. 1999. Design and Implementation of Intentional Names. Master's thesis, Laboratory for Computer Sciences, Massachusetts Institute of Technology.

Wu, S. Y.; Miranker, D. P.; and Browne, J. C. 1996. Decomposition Abstraction in Parallel Rule Languages. *IEEE Transactions on Parallel and Distributed Systems* 7(11):1164-1184.