Classifying Architectural Elements as a Foundation for Mechanism Matching

Rick Kazman, Paul Clements, Len Bass
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, PA USA 15213

Gregory Abowd
College of Computing, Georgia Institute of Technology
Atlanta, Georgia USA 30332

Abstract

Building a system at the architectural level can be thought of as decomposition into components followed by a series of exercises in *matching*. Components must be composed with each other via matching *mechanisms*; matching *signatures* within those mechanisms ensure that data and control will flow through the system; and matching *semantics* among the components ensures that the system will meet its behavioral requirements. The standard concepts of software architecture (components, connectors, styles) have been widely used with little more than intuitive understanding of their meaning. Mechanism matching is currently an *ad hoc* exercise that relies on the peculiarities of programming language facilities. This paper presents a set of well known but informally described software architectural elements used in system composition, and taxonomizes them under a basic set of characteristic features. This classification allows us to describe legal combinations of architectural elements by performing a simple matching exercise on the relevant features of the member elements. This classification also allows us to identify architectural elements that can be substituted for each other and satisfy the same mechanism matching requirements. This leads to delayed binding of architectural mechanisms, which in turns provides increased flexibility and greater opportunities for reuse of units of computation.

1 Introduction

Developing a software architecture for a system involves decomposition into components, defining interactions among the components, and then a series of exercises in matching. *Mechanism matching* involves instantiating the components and interactions into language mechanisms that facilitate implementation of and communication among the components. For instance, components may be instantiated as objects, programs, processes, or tasks, which may interact with each other via remote procedure call, shared memory, sockets, pipes, or rendezvous. Without mechanism matching elements cannot be composed into executable systems. *Signature matching* refers to the agreements on the form of the data that flows among matched elements—types, structures, number of containers (e.g., parameters), etc. *Semantic matching* is required among the elements to assure that the computations will together satisfy the behavioral and resource utilization requirements of the system.

This paper provides a set of foundational concepts for mechanism matching. It does so by classifying a set of architectural elements (which is our term for what are commonly but informally referred to as components and connectors) according to a set of language-independent, behavior-describing criteria which we call *features*. The elements we have chosen to classify are ones treated as components or connectors in the relevant literature.

The features describe how these elements resemble and differ from each other. The features describe only externally-visible (non-implementation-dependent) characteristics of the elements for classification. They provide a basis for comparing elements in ways to assist in system composition (by defining legal combinations and configurations of elements) and substitution (by identifying what elements can serve as satisfactory replacements for other elements). In particular, the objective classification of architectural elements will facilitate architecture-level system composition in each of the ways described below.

- Viewing systems at the architectural level helps us think of components and interactions in terms of their essential behaviors and interactions, and not in terms of idiosyncratic programming-language constructs such as parameter-passing or rendezvous. However, components and the systems they compose must still be programmed. We are therefore still captive to the element mechanisms provided by the languages we use, and this captivity carries a cost: it forces us to program components that are less general (less reusable) and more programming-language-specific than they should be. Architectural choices should determine the choice of mechanism, as opposed to the choice of

mechanism determining the properties of the interaction. By making behavioral features of architectural elements explicit, we enable programmers to specify what is important to them *and no more*. They can defer the binding of features to mechanisms until late in the system creation process.

- By understanding the essential features of an architectural element, we can determine constraints on putting elements together. What is a layer? What kinds of clients can a particular server connect to? Can a client connect to a pipe? These questions can be answered, in part, by examining the features of elements.
- Just as the periodic table of chemical elements was used to predict the existence of new elements before they had been discovered in nature, a classification of architectural elements via canonical features should allow us to envision and create new kinds of elements that exhibit feature combinations not previously observed.

By observing naturally-occurring clusters that emerge from the taxonomy, we may gain new insights into the nature of architectural elements. For example, the relationship between procedure call and RPC is well known (and emerges from our classification as well); perhaps analogous and unexpected relationships will emerge between other elements. A general classification scheme will allow us to identify and investigate element clusters in a systematic fashion using a consistent vocabulary.

2 Features of architectural elements

When crafting the set of features, we found two system views to be useful. The *temporal* view considers the execution of the element as an episode in time; discriminating features that precipitate from this view describe the properties of these temporal segments. Conceptually, temporal features are those that we can observe by watching the element execute. The *static* view yields features that can be discerned without observing the element execute, but by examining its specification (in the extreme case, its code). The static view summarizes an element's invariants.

Other views of an element may yield interesting features as well, but for now these two views are the ones we will use to discriminate architectural elements.

2.1 Features derived from the temporal model

The temporal view of architectural elements describes the behavior of an element over time. It views the execution of an element as a series of contiguous temporal "episodes" in terms of threads of control that flow through the element.

Useful distinctions among architectural elements may be made in terms of this temporal view. For example, an element that does not allow data to be dispatched in the middle of an execution episode has quite different characteristics (and usages) from an element that does. An element that does not accept control during execution cannot be re-used in a host of situations in which a re-entrant counterpart can.

The temporal view immediately suggests a set of architectural features, some of which are enumerated below. Others are possible. We define the features as answers to a set of questions about the element being described: "Does every instance of this type of element dispatch data at t_e?", etc. Temporal features include:

- *Times of control acceptance*
- *Times of data acceptance*
- *Times of control and data transmission*
- *Forks*
- *State retention*

Table 1 summarizes the features derived from the temporal view, and the possible values for each.[1]

1. In this table A=always, N=never, NC=not criterial, and n/a=not applicable, S=same, D=different. t_s refers to the time at which an element is created (its *start* time) and t_e refers to the time at which an element is destroyed (its *end* time).

Architectural Element	TEMPORAL FEATURES							
	Accepts control at other than t_s?	Transmits control at other than t_e?	Accepts data		Transmits data		Forks?	Retains state?
			At t_s?	At other than t_s?	At t_e?	At other than t_e?		
Possible answers:	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	A N NC n/a	S D SD N

Table 1: Features derived from the temporal view, with possible values

2.2 Static view features

Architectural Element	STATIC FEATURES								
	Data scope	Control scope	Trans-forms data?	Binding time	Blocks?	Relin-quish?	Ports	Associations	
								Per conn	Lifetime
Possible answers	V P D n/a	V P D n/a	Y N n/a	S I E	Y N NC	Y N n/a	I O IO	*n*	*n*

Table 2: Features derived from the static view, with possible values

The static view of architectural elements compiles information based on static information, which does not change as a function of time or as a result of executing, using, or interacting with the element being described. The static structure of an architectural element does not change once it has been specified.

The static view suggests a set of features for architectural elements. Like the temporal features, the static features are posed as answers to questions about the element, in this case questions about its time-invariant capabilities and properties.

Features from the static view that are relevant to architectural elements include:

- *Data scope*
- Control scope
- *Transforms data*
- *Binding Time*
- *Blocks*
- *Relinquish*
- Ports
- Associations

Table 2 summarizes the features derived from the static view, and the possible values for each.[2]

3 Using the classification

In this section we discuss *element matching*, a process for comparing elements according to the classification criteria, that will help us achieve those goals. Element matching involves setting match criteria and then seeing which architectural elements in the classification satisfy the match. A match on a feature is either an *exact* match or a *satisfying* match. An exact match is when some feature of an architectural element has the same value as that imposed by the match criteria; e.g. [Data scope=P], or [Binding time=S], or there is an intersection of values on the feature, e.g. $[Bindingtime \in \{S,E\}]$ in the match criteria and [Binding time=I,E] for the element. A satisfying match means that a feature of the match criteria has a value of NC while in the element this feature has any value (e.g. [Data scope=NC] is satisfied by [Data scope=V]). An element matches a query exactly (satisfactorily) if all of its features match those of the query exactly (satisfactorily).

For some features we define an ordering of the possible enumerated values. For example, the data and control scope features have the possible values of: [V, P, D]. Any enumerated value can match with itself, or with a value earlier in the enumeration. For example, [Data scope=D] matches [Data scope=D], [Data scope=P, or [Data scope=V] whereas [Data scope=V] can only match with another element that has [Data scope=V] specified.

Finally, feature values of “n/a” only match with each other or with “NC”. This is because two elements can be connected together either if they both lack a particular property (say, control) or one lacks the property and the other one possesses it, but it is not criterial to the definition of the property. For the purposes of matching, these are considered to be exact matches.

3.1 Element matching for delayed and flexible selection of elements

If we allow system-builders to specify their requirements for architectural elements’ interactions rather than force them to over-constrain by using a programming-language mechanism, reuse and flexibility are enhanced, as argued in Section 1. The classification features of Section 2 provide a language for expressing the architectural requirements for inter-element interaction. Such an expression in turn forms a query for element matching to discover all of the architectural elements that will satisfy the system-builder’s stated essential requirements. In this case, element matching identifies equivalence classes of architectural elements that may be chosen at system composition time to satisfy a given set of interaction requirements expressed using the characterizing features.

2. In this table V=virtual, P=physical, D=distributed, S=specification, I=invocation, E=execution, I=input, O=output. The definitions of ports and associations largely follow that used in UniCon [3].

3.2 Element matching for composition

The idea of mechanism matching introduced in Section 1 requires us to know what architectural elements can bind with what other elements. We need this to be able to stipulate *well-formedness* properties of architectures. That is, we need to say why client-server is a valid architectural form, and not client-process, or object-server. We also need to be able to discuss what elements can be used in conjunction with what other elements to discuss both the design process and the re-use process.

The well-formedness constraint is a specialized form of element matching; it requires that every element in an architecture must have all of its ports matched by some other element (or by user input or output) by run-time. Thus, the graph representing the architecture must be completely connected.

If we are trying to compose two elements, A and B, then A and B must match on their ports, associations, and binding time. Other features will constrain the ways in which the elements can interact during execution. For example, the values for data acceptance/transmittal and control acceptance/transmittal will affect the relative ordering of the elements' computations at run-time, affecting control flow patterns.

Features such as data scope and control scope will affect how elements can connect together. For example, two elements with a data scope of "P" but which are located on distinct machines (for example, clients and servers), must have an intervening element with a data scope of "D" (such as a socket), in order to be connected.

In order to be composed with each other, elements do not need to match on all features. State retention, data transformation, blocks, forks, and relinquish are immaterial with respect to composition. (These features play a role in element matching for substitution, as we saw in Section 3.1, and in semantic matching as well.) It is reasonable to expect that a system will be composed of elements having a wide range of these properties.

4 Conclusions

By decoupling architectural decisions from decisions about the division of functionality, designers can modularize functional computation to provide for abstraction and information hiding and specify the architectural options in terms of the features we have identified here. This specification could be either through language extensions or through a complete development environment that supports architecture-level composition and language-level development as separate steps. Crucially, the architectural mechanisms are *not* bound by language features. This creates a powerful new system-building paradigm that enhances reusability of components and allows system builders to think in terms of the application rather than the programming language.

This categorization scheme can also shed insight into classes of elements. By uncovering different meanings that people assign to an element, and codifying those various meanings, classes emerge. Also, creating this classification has taught us that whether an element is a component or a connector is not an inherent property of the element itself, but almost always reflects a choice of how the element is modeled. We can find no criterial distinction between components and connectors; our features allow us to view many elements as either, providing flexibility and convenience.

Finally, these architectural features add a semantic element to the purely syntactic descriptions of idioms that is currently available from work such as that of Dean and Cordy [1].

Much further work remains to be done in applying the features we have identified:

- Further investigation into deferring architectural mechanism decisions is required. Do our features provide a rich enough requirements language for the decisions that architects building real systems make?
- The design space needs to be further explored for new elements and new combinations of old elements.
- Integration of this effort with work on evaluation of architectures for quality [2] is required, so that deferred decisions can be made on a principled basis.
- Mechanism matching needs to be integrated with signature matching and semantic matching to search for previously-created components that can be integrated to solve the problem at hand.
- Investigation is required into re-engineering a system by abstracting its architectural decisions and providing a basis for systematically modifying them. This could be a promising approach to achieving controlled architectural migration.

5 References

[1] Dean, T., Cordy, "A Syntactic Theory of Software Architecture", *Transactions on Software Engineering*, 21(4), April 1995, 302-313.

[2] Kazman, R., Abowd, G., Bass, L., Clements, P., "Scenario-Based Analysis of Software Architecture", *IEEE Software*, November 1996, 47-55.

[3] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G., "Abstractions for Software Architecture and Tools to Support Them", *Transactions on Software Engineering*, 21(4), April 1995, 314-335.