Architectural Unification

Ralph Melton and David Garlan*

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

{ralph,garlan}@cs.cmu.edu

Phone: (412) 268-7673
Fax: (412) 268-5576

31 January 1997

Submitted For Publication, January 1997

Abstract

Many software designs are produced by combining and elaborating existing architectural design fragments. These fragments may be design patterns, partially thought-out ideas, or portions of some previously-developed system design. To provide mechanized support for this activity it is necessary to have a precise characterization of when and how two or more architectural fragments can be combined. In this paper we describe extensions to notations for software architecture to represent incomplete design fragments, and algorithms for combining fragments in a process analogous to unification in logic.

1. Introduction

Software architecture is increasingly recognized as an important level of design for software systems. At this design level systems are usually represented as a set of coarse-grained interacting components such as databases, clients, servers, filters, blackboards [GP95, GS93, PW92]. Architectural design describes complex systems at a sufficiently high level of abstraction that their conceptual integrity and other key system properties can be clearly understood early in the design cycle.

One of the central benefits of architectural design is support for design reuse. Many systems are built by combining and elaborating existing architectural design fragments [BJ94, BMR+96]. These fragments may be design patterns, partially thought-out ideas, or portions of some previously-developed system design.

Currently the practice of reusing architectural fragments has a weak engineering basis. Architectural design fragments are usually represented informally and the principles by which partial designs are combined are informal and ad hoc. In particular, given an architectural design fragment it is typically not clear either (a) what aspects of that fragment must be filled in to complete the design, or (b) how to determine whether a given context can satisfy its requirements.

*. The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grants F33615-93-1-1330 and N66001-95-C-8623; and by National Science Foundation Grant CCR-9357792. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, or the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Recently, considerable progress has been made in developing ways to represent software architectures. A number of architectural description languages have been developed, together with tools for manipulating those descriptions, formally analyzing them, and storing reusable architectural elements [GAO94, MORT96, LAK+95, MQR95, SDK+95, AG94].

While the ability to formally represent architectural designs is a necessary first step, notations and tools for architectural representation do not by themselves solve the problems of combining architectural design fragments. First, these notations usually lack any means for characterizing "partiality" in architectural design—that is, explicit identification of missing parts and context requirement. Second, they do not provide effective criteria or mechanisms for combining several fragments into a single coherent design.

In this paper we attempt to remedy the situation. First, building on standard representational schemes for software architecture, we show how partial designs can be characterized using *architectural placeholders*. Second, building on theoretical underpinnings from unification in logic, we provide an algorithm for combining multiple such fragments.

The underlying ideas behind our approach are straightforward: Placeholders in an architectural design serve as variables in an architectural expression. Legal combination of several such "expressions" is determined by a form of unification that matches variables to instances. However, as we will show, the domain of architectural representations raises a number of challenges that make the naive application of the traditional unification algorithm difficult (if not impossible). We then show how those challenges can be met by extending the algorithm in several significant ways.

2. Related Work

Three areas of research are closely related to this work: software architecture, reuse, and logic.

2.1. Software Architecture

Within the emerging field of software architecture, two related subareas are architectural description languages (ADLs) and architecture-based reuse.

Architecture Description Languages

A large number of ADLs have been proposed (at last count over a dozen). While these languages differ from each other in many significant ways, virtually all share a common view that a software system's architecture describes the system's structure as a hierarchical configuration of interacting components. Beyond this, different ADLs typically add flesh to the structural skeleton by allowing additional semantic information that characterizes such things as abstract behavior, implementation choices, and extra-functional behavior (performance, space requirements, etc.)

Our work builds on architecture description languages by adopting their common structural basis (detailed in Section 4). Auxiliary information is handled in our framework by viewing that information as residing as attributes of the architectural structure. (The same approach is taken by a number of other languages, including, Aesop, and UniCon.) In this way our results are largely ADL-neutral: to the extent that an existing ADL adopts our generic structural basis for architectural representation, our results will apply. However, as we will see our work also extends the usual architectural representation schemes, by introducing an explicit notion of a placeholder and mechanisms for filling those placeholders.

Architecture-based Reuse

Research in software architecture has focused primarily on two aspects of reuse.

- **Code**: A number of systems (such as GenVoca [BO92] and UniCon [SDK+95] provide high-level compilers for certain classes of software architecture. These support reuse of implementations, primarily by exploiting shared implementation structures for some class or domain of systems. However, unlike our work they are less concerned with general mechanisms for reuse of architectures themselves.

- **Style**: Some systems (such as Aesop [GAO94] and Wright [AG94]) support the definition of families of architecture that adopt a common set of design constraints - sometimes called an architecture "style". For systems within such a family it is often possible to provide (and thereby reuse) specialized analysis tools, code generators, and design guidance. Styles typically prescribe general rules for architecture (such as a specialized design vocabulary and constraints on how the elements are composed). This differs from the work described here, which focuses on reuse of architectural fragments.

2.2. Reuse

Architecture aside, the general area of reuse has been an active research area for many years. Historically most of that work has focused on code-level reuse, with an emphasis on representation, retrieval, and adaptation of implementations.

More recently, work on design patterns has raised the level of concern to design reuse. In particular, design patterns have been extremely successful in capturing idioms of reuse that are intended to solve specific problems and that can be codified and packaged in handbooks [Bus93,GHJV95].

Our work is closely related to design patterns. However, there are two significant differences. First, we focus more narrowly on *architectural* patterns, a subset of the more general body of research on patterns. This more specialized focus allows us to exploit specialized representational schemes for architecture, and to consider carefully the specific needs for architectural design reuse. Second, the main thrust of the design patterns work has been on packaging of good intuitions and ideas about problem solving. Our work complements this by looking at formal representation schemes and algorithms for deciding whether the use of a pattern is valid in a given context.

2.3. Unification

Unification in logic has been an area of formal study for many years [Kni89]. In this paper we build on those results, extending and adapting them where necessary to fit the problems at hand. Our research is not intended to make novel contributions to this area, but rather to adapt what already exists to the domain software architecture.

3. Example

To illustrate the issues of representing and combining architectural design fragments, we use an example from a recent development project which constructed a specialized software development environment. The system was to provide persistent storage for programs that could be viewed and manipulated through a graphical user interface. The system was also intended to provide a reuse repository. In developing this system, we naturally found ourselves incorporating two fragments in the architectural design (see Figure 1).*

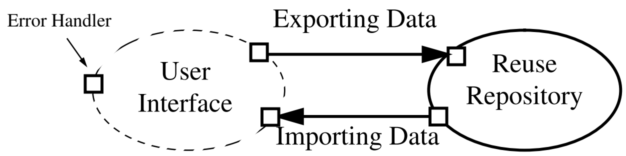

Figure 1a. UI--Repository Interaction

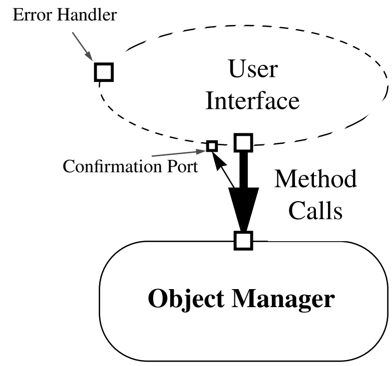

Figure 1b. UI—Object Manager Interaction

In one fragment (Figure 1a), we designed the reuse repository and the connections between it and the user interface, leaving the user interface largely unspecified. To import objects from the repository, the reuse repository passes a description of the object to the user interface. To export objects to the repository, the opposite path is taken. The architecture prescribes the details of the communication and repository implementation in some detail (not shown here), but leaves the user interface portion of the design largely unconstrained. It does, however, require that the user interface have an error handling interface (for various technical reasons).

In another fragment (Figure 1b), we designed an Object Manager and the protocol by which it interacts with a graphical user interface. This protocol is similar to a method call, but includes a backwards connection through which the Object Manager can request confirmation of a questionable action from the user. In the fragment, the Object Manager and the interaction protocol are fleshed out, while again the user interface is largely a placeholder.

In the resulting system these fragments are combined (Figure 2). Here the two design fragments are merged. In particular, there is only one user interface, which satisfies the requirements for both design fragments. The error handler interface of both user interfaces are identified, although the Object Manager and Reuse Repository interfaces are kept separate. To complete the architectural design for the system, an actual user interface design must be provided to sat-

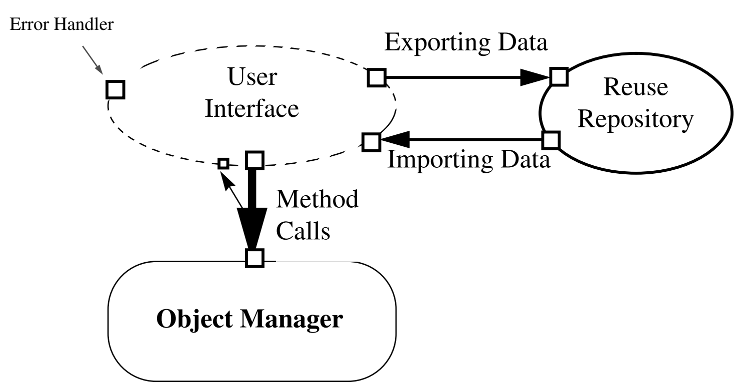

Figure 2. Combination of Design Fragments

*. For expository purposes, in this example we illustrate only a simplified top-level view. The actual design includes considerably more representational detail, and associated properties. See [GAO94] for a detailed description of the system.

isfy the joint requirements of the two fragments. In particular, it must support the interfaces for interaction with the Object Manager, the Reuse Repository and an Error Handler.

Several issues are worth noting about the way in which we (informally) designed and combined the fragments. First, neither architectural fragment can be viewed as a single encapsulated entity: each has substructure that must be exposed. This suggests that traditional techniques (e.g., using parameterization) for composing encapsulated objects do not apply here.

Second, design fragments specify not only parts of a design that they are *providing*, but also parts of the design they are *expecting*. In this example the Reuse Repository fragment provides the Repository design and the protocol for interaction with it. But it requires a User Interface to complete the fragment. Similarly, the Object Manager fragment provides the Object Manager and its interaction with the User Interface, but requires a User Interface.

Third, it is not enough to simply name the parts expected to be filled in by the surrounding context (such as the User Interface in Figure 1a). In addition it is also important to say something about the structure and properties of those missing parts. Even in this simple example, the Repository fragment indicated that the User Interface must have an error handling interface, and an interface that could communicate using the Exporting and Importing protocols required by the Repository.

Fourth, the question of which parts of the fragments should be combined (or identified) in the final design is not an obvious one to answer. Can a single user interface satisfy the needs of both designs? If so, should the combined User Interface have one or two Error Handler interfaces? Why is it that the two User Interfaces are combined, but the Object Manager isn't identified with the Reuse Repository? While the answers to these and similar questions may be intuitive at this informal level of exposition, formal criteria for making such decisions are clearly lacking. In the remainder of this paper we show how a simple representational scheme, together with an algorithm for composition of design fragments can address the issues in a precise way.

4. Representing Software Architecture

To talk about architectural designs and fragments at all, we first need some way to represent them. We will take as our starting point the structuring concepts more or less uniformly adopted by the software architecture research community. Specifically, in software architecture, designs are typically described in terms of the following generic vocabulary of architectural design *elements* [GP95]:

- A *component* is a locus of computation or a data store. (Examples include databases, filters, abstract data types, etc.)
- A *connector* is an interaction between two or more components. (Examples include procedure call, event broadcast, pipes, and interactions involving more complex protocols such as a client-server protocol.)
- A *port* is an interface to a component. A component may have multiple ports. (For example, a filter might have data-in and a data-out ports.)
- A *role* is an interface to a connector. (For example, a pipe might have reader and writer roles.)
- A *configuration* is a set of components and connectors along with a set of *attachments* between ports of the components and roles of the connectors.

Architectural descriptions may be hierarchical. This is accomplished by allowing any component (or connector) to have an associated architectural *representation*, which describes its internal structure or provides an implementation in some programming language.

In addition to this simple topological structure, we will assume that each of the architectural elements can have a set of associated *properties attributes* (or simply *attributes*). Properties describe such things as the protocol of a connector, the functional behavior of a component, the data type accepted by a port, or the maximum throughput of a filter.

Finally, architectural designs may have an associated set of *constraints* that regulate how they may be modified. For example, a filter component in some design may have the constraint that all of its ports are either input ports or output ports; a configuration of filters connected by pipes may have the constraints that input ports of filters may only be attached to reader roles of pipes and output ports to writer roles.

To relate this representational vocabulary to the earlier example, the User Interface of Figure 2 is a *component* with five *ports*: a data exporting port, a data importing port, a method-calling port, and a port for handling confirmation of method calls. The Method Calls *connector* has three *roles*: the caller role that initiates the method, the callee role that executes the method, and the confirmation role that confirms or cancels operations that the caller role considers questionable. The method-calling port of the UI is attached to the caller role of the Method Calls connector, and the confirmation port of the UI is attached to the confirmation role of the connector.

Architectures characterized in terms of this vocabulary are representable in any number of architecture description languages. For concreteness, Appendix 1 shows a textual specification for Figure 1a in the ACME architecture description language [GMW95]. However, it is our intent not to focus here on the details of a particular concrete notation; the techniques that we present should be applicable to all architecture description languages that use a similar set of concepts.

5. Architectural Design Fragments

There are two major reasons why designers naturally use design fragments to design software architectures. The first reason is *comprehensibility*. Even though a software architecture is an abstract description of a software system, a large system may be too complex for a person to understand and design in a single bite. Design fragments allow one to focus on smaller (overlapping) chunks of a system. Often, the architecture is sketched first at an extremely abstract level, and then portions of that abstract architecture are refined in terms of fragments that will then be combined to make a detailed whole.

The second reason designers naturally use design fragments is one of *reuse*. Many good reusable ideas cannot be encapsulated in a single architectural element (or class). The recent development of *design patterns* exploits this observation, for example.

Within the context of software architecture, it makes sense to describe architectural design fragments as "pieces of architecture," that is, partially specified or incomplete architectural designs. Such design fragments can be represented as mini-configurations of components, connectors, and attachments and characterized in any number or architecture description languages.

Unlike conventional architectural descriptions, however, design fragments are by their nature incomplete: they leave some things deliberately unspecified. Therefore, to fully capture the intentions of fragment creators, it is necessary to explicitly indicate what parts of the design are intended to be filled in later.

To represent the portions of a design fragment that are left deliberately unspecified we introduce the representational concept: that of placeholder elements. A *placeholder element* is an partially-specified architectural element[*]; it is intended that more detail will be added to it in a complete design. (We will refer to such a non-placeholder as a *real element* or *fully-specified element*.) Using this approach any of the parts of an architectural design can be designated as placeholders. A placeholder designation thus indicates that the architectural details associated with that element are requirements that must be satisfied by the eventual design.[†]

*. When we say "architectural element", or simply "element" we are referring to any of the kinds of architectural entities described in the previous section: components, connectors, ports, roles, representations, etc.

†. It should be clear that there are no a priori criteria for determining whether a part of an architectural design is a placeholder or real. Wether an element in a design is fully specified is a question that is determined by the designer. If the designer intends that a part of a design fragment should be elaborated further it can be designated as a placeholder; otherwise not.

For instance, in the UI–Repository fragment of Figure 1a, the User Interface is a placeholder representing functionality that needs to be implemented. The real element that eventually fills that place in the final design must have the ports that are specified on the placeholder: a port that receives architectural descriptions that is attached to a role on the ‘Importing Objects’ connector, a port that sends architectural descriptions that it attached to a role on the ‘Exporting Objects’ connector, and a port for handling errors.

Note that this scheme permits the description of arbitrary complex structures that are required by a fragment for its completion. For instance, it would be easy to characterize a fragment for which a particular real architectural component C must be connected to a component D, which must in turn be connected to a component E in the final design. Here D and E are placeholder elements. The amount of detail associated with them will determine how tightly they constrain the parts that are eventually substituted for them.

It would be possible to start from a single design fragment and simply elaborate it, adding detail to the placeholders as needed. However, more likely several fragments are designed, each indicating some placeholders (and requirements on them). The key question then becomes how can we “merge” such fragments into a single design?

Intuitively, to combine two design fragments, we must replace some or all of the placeholders in one fragment with elements from the other fragment. More specifically, we must be able to do two things:

- decide whether a real element matches a placeholder, to ensure that any properties that hold of a placeholder still hold when a real element is inserted in its place.
- combine two placeholders to get a third, more detailed placeholder.

In the next sections, we describe how to adapt the idea of unification from logic to accomplish this.

6. Unification in Logic

The problem of merging the parts from two architectural design fragments is essentially one of resolving multiple, possibly incomplete descriptions of elements, to find new elements that satisfy both descriptions -- or report that no common element can be found. *Unification* is a technique from the first-order predicate logic that does just that for logical terms.

In the terminology of logic, a *term* is a constant, a variable, or a function applied to other terms. Examples of terms are ‘3’, ‘x’, and ‘f(x,3)’. A *substitution* is a function from terms to terms that replaces variables with other terms. For example, if the substitution {x -> g(y)} is applied to the term ‘f(x,3)’, the result is ‘f(g(y), 3)’.

Unification works by finding a substitution of expressions for free variables that can be applied to both expressions to make them identical. Because of the nature of the substitution, any logical property of the expressions prior to the substitution still holds after the substitution is globally applied. If no substitution can be found, the unification algorithm determines that there is no possible match between the two expressions.

For example, given ‘f(x) = g(x)’ and ‘g(3) = 4’ to show ‘f(3) = 4’, one would apply an appropriate substitution to make the expressions ‘g(x)’ and ‘g(3)’ identical. Unifying ‘g(x)’ and ‘g(4)’ returns the substitution {x -> 3}. This substitution can then be applied to both axioms, yielding ‘f(3) = g(3)’ and ‘g(3) = 4’. Transitivity of equality can then be used to obtain the desired result.

Here is the usual algorithm for unification in the domain of logic.

```
Unify (u1, u2: term; var subst: substitution): boolean
    let u1' = subst (u1); let u2' = subst(u2)
    if u1' = u2' then return true;
    if u1' is a variable v which does not occur in u2'
        add (v -> u2') to subst;
        return true;
```

```
(Do the same for u2')
if u1' and u2' are constants that are not equal, return false
If u1' is a function application f(x1, x2, ... xn) and u2 is a function
   application g(y1, y2, ..., ym),
   if f and g are not the same function with the same number of arguments
      then return false
   else
      for each pair of arguments (x_i, y_i)
         if not Unify(x_i, y_i, subst)
            return false
      return true
```

The algorithm returns a boolean expressing whether the two terms can be unified, and also returns in a variable , *subst*, the substitution that unifies them. The term that matches both descriptions is obtained by applying the resulting substitution to either of the original arguments.

7. Architectural Unification

Unification solves the following problem: "Given two descriptions *x* and *y*, can we find an object *z* that fits both descriptions?" [Kni89]. As noted above, in the domain of logic these descriptions are expressions that may contain variables. We can apply this idea to software architecture; corresponding to expressions and variables in logic are architectural descriptions and placeholder elements (respectively).

A naive approach would be this: let Component(...), Connector(...), Port(...), Role(...), and Configuration(...) be functions, each with a fixed number of arguments. (For example, Component(Name, Port1, Port2, Port3, Representation1) would be a function that would take a name, three ports, and a representation, and return a component.) Placeholders would be represented by using variables as arguments of those functions. We could then use the straightforward unification algorithm from logic to combine those structures.

This use of unification to combine design elements would have several desirable properties. The algorithm would determine when two placeholders can be combined and provide the resulting combination. It would also determine when a fully-specified fragment (i.e., one without placeholders) can be combined with one that contains placeholders. (Note that as a corollary, the algorithm implies that two fully-specialized fragments are unifiable only if they are identical.)

Unfortunately, this naive approach will not work for four reasons. First, fixed-arity, functional terms are inadequate to represent architectural structures with extensible lists of attributes. Second, architectural structures naturally contain sets; these require special attention. Third, architectural structures may contain information that cannot be easily expressed as terms containing variables; it is desirable to be able to combine that information also. Fourth, architectural structures may be subject to auxiliary constraints (such as those imposed by an architectural style), and unification must respect those constraints. In the sections that follow, we extend the simple unification algorithm to handle these issues.

7.1. Extensible attribute lists

The first problem with the classical unification algorithm is that fixed-arity terms (i.e., functions that take a fixed number of arguments) are inadequate for representing architectural structures. We wish to use unification to describe the combination of architectural design elements that may have varying lists of arbitrary attributes. These attributes arise as architectural designs are elaborated: when new properties are asserted or derived they are added to the existing design. For example, a tool might calculate a "throughput" value for a component that had no "throughput" attribute before. This poses a problem for unification of terms in which each function symbol has a fixed arity. If, for example, Component(...) is a function symbol with five arguments corresponding to a name, three ports, and a representation, there is no way to add an argument corresponding to throughput.

For our first adaptation of the canonical unification algorithm, we adopt the approach of using unification over *feature structures*, as introduced in [Kay79]. Instead of using function applications and their positional arguments to structure terms, feature structures identify substructures by symbolic names. Feature structures are thus similar to attribute-value lists. Unification of two feature structures combines the features of both structures, while attempting to unify the values of features with the same names.

To apply this idea to architectural representations, we first convert each element of the design element into a feature structure. For example, a feature structure to describe the User Interface of Figure 1a might be this:

```
[type: component,
 name: "User Interface",
 ports: {[type: file_handle,
            name: "Method Call Interface"],
         [type = ACME_output,
         name = "Data export"],
         [type = ACME_input,
         name = "Data import"]
        },
 Placeholder = true]
```

Note that the values of features may themselves be feature structures, as with the "ports" feature of the Object Manager[*].

Using feature structures instead of first-order terms requires that we adapt our unification algorithm. Where the previous algorithm handled the unification of function applications by recursively unifying their arguments, we now handle the unification of feature structures by recursively unifying the values of any attributes that are present in *both* feature structures, and then copying any attributes which appear in only one unificand. The following algorithm is based on the approach described in [Kni89].

```
Unify_feature_structures (u1, u2: structure; var subst: substitution): structure
   let u1' = subst (u1); let u2' = subst(u2)
   if u1' = u2' then return u2';
   if u1' is a variable v which does not occur in u2'
      add (v -> u2') to subst;
      return u2';
   (Do the same for u2')
   if u1' and u2' are constants that are not equal, fail
   if u1' and u2' are both feature structures:
      let u_new be a new feature structure.
      for each feature label L that is present in both u1' and u2'
         set u_new.L to be Unify_feature_structures( u1.L, u2.L )
         If the call to Unify_feature_structures failed, fail
      for each feature label L that is present in u1' but not present in u2'
         set u_new.L to u1'.L
      for each feature label L that is present in u2' but not present in u1'
         set u_new.L to u2'.L
      return u_new
   else fail
```

*. The ports feature is actually a set of feature structures. Sets will be covered in the next section.

7.2. Sets

The second problem is that architectural descriptions often involve sets of structures. Sets appear in three places in architectural descriptions: the ports of a component, the roles of a connector, and possibly the values of some attributes. For example, the *ports* feature of the User Interface component of Figure 1a is not a single port, but a set of ports. This requires further extensions to the unification algorithm, since the standard algorithm does not handle sets at all.

The simplest way to add support for sets to the unification algorithm is to treat sets as a type of constant; two sets unify if and only if they are identical. However, this is an excessively restrictive, since it excludes the possibility of unifying the elements of one set with those of the other. (In that case we would never be able to unify a port -- such as the Error Handler -- of one component with a port on another component.)

For unification of sets, we wish to preserve the following property: if sets X and Y are unified to yield set Z, every element x of X is represented by some element in the set Z that is a specialization of x (and similarly for Y). Further we would like to unify elements across the two sets, where possible. Therefore, to unify sets X and Y, we choose a set of pairs of elements (x_i, y_i) from the two sets such that each x_i can be unified with the corresponding y_i, and we add the unifications of those elements to the unification of the two sets. Then, we add all the elements from X and Y that were not part of any pair that was unified (in a way analogous to feature unification).

Unfortunately, there may be more than one pairing between X and Y that accomplishes this; different pairings will yield different unifications of the sets. How should one then pick the best pairing? Based on our experience with architectural fragments, we believe that the algorithm should have the two properties:

1. It unifies as many pairs of elements from the two sets as possible, so that the resulting set has as few elements as possible. This means that we choose a solution that has as much overlap between the two unificands as possible, in hopes of minimizing the new constraints that are imposed on the result.
2. It does not unify any elements from the same set. When designers distinguish elements in the same design fragment, it is likely that they intend to have two elements those distinctions preserved in the final design.

We use a backtracking search through the possible pairings to find the best pairing, unify all the elements in that pairing, and then add the leftovers to the unification set, according to the following algorithm:*

```
UnifySets(set1, set2, mapping)
    Let Pairings = FindMaximalPairing(set1, set2, mapping)

    for each pairing (u1, u2) in Pairings
        Unify(u1, u2, mapping)
    for each u1 in set1 - (dom Pairings)
        Unify(u1, I, mapping)
    for each u2 in set2 - (ran Pairings)
        Unify(I, u2, mapping)
```

All the complication of UnifySets is in FindMaximalPairing. FindMaximalPairing takes the two lists, the mapping, and the context, and returns a (possibly empty) list of pairs of elements such that:

1. For each pair (*u1, u2*) in the list, *u1* is in *set1*, *u2* is in *set2*, and *u1* and *u2* can be unified.
2. No element occurs in more than one pair.
3. There is no longer list of pairs that satisfies requirements 1. and 2.

FindMaximalPairing does a backtracking search. Backtracking is necessary, because simpler algorithms cannot be guaranteed to find a maximal pairing. For example, consider unifying two sets, each of which contains one real object

*. The complexity of this algorithm is clearly exponential. Fortunately, the sets encountered in most architectural descriptions are small.

and one placeholder object. If an algorithm tries to unify the two placeholders first, it will not be able to unify the two real objects.

```
FindMaximalPairing(set1, set2, mapping)
    let AllPairs = set1 × set2
    return TestPairs(AllPairs, mapping)

testPairs(pairs, mapping)
    // pairs is a set of pairs of elements to try to unify.
    if pairs = {} return {}

    let p be an element of pairs.
    let u1 be (first p); let u2 be (second p)

    try Unify(u1, u2, mapping)
        if unification succeeds
            let L1 be list( p, testPairs( {(u1',u2') ∈ pairs | u1' != u1 ∧ u2' != u2},
mapping)
            undo unification
            let L2 be testPairs( pairs - {p}, mapping)
            if (length(L1) >= length(L2))
                return L1
            else return L2
        else return testPairs( pairs -{p}, mapping}
```

7.3. Feature-Specific Unification

The third problem is that we would like to use feature-specific information to unify elements that could not otherwise be unified. The naive algorithm fails to unify elements that have the same feature but with different values for that feature. Often, however, we can use knowledge of the semantics of the features to combine such information.

In general, there are four possible responses to an attempt to unify two different features:

- Allow one unificand to override the other according to some policy;
- Attempt to merge the two attributes; the exact nature of the merge would depend on the type of the attribute; if the merge fails, we would fail to unify the objects;
- Nondeterministically choose one of the attributes;
- Fail.

In general, no one of these policies is right all of the time. In fact, the best policy is generally feature-specific. Therefore, we introduce feature-specific unification methods as extensions to the basic algorithm.

For example, consider an attribute *specification* that partially specifies the behavior of a component. Two specification attributes can be combined to create a specification attribute for a unification that captures both of those specifications. The simplest way to do that combination is through logical conjunction. As another example consider merging two elements that contain the visualization information (for example, describing how an element should be displayed on the screen). One reasonable policy is to place the result in the place formerly occupied by one of the unificands. Again, this is difficult to express as an assignment of values to variables.

To handle the feature-specific unification, the default policy is to fail when the values of two features of the same name differ. However, the party that defines the attribute type may also define the mechanism for resolving differences between those attributes in unification. To enable this, we add a hook to *Unify_feature_structures*; when it encounters different values for an attribute, it calls a routine determined by the name of the attribute and the type of the object on which it is being created to determine how to resolve those differences.

7.4. Design Restrictions

The fourth problem with the standard unification algorithm is that it must be modified to support additional constraints on the results of unification. This arises because many architectural designs are developed in the context of a set of restrictions on how design elements can be used and modified. For example, in the Aesop system [GAO94] the Pipe-and-Filter style supplies a component of type *filter.* A filter is subject to the restriction that it may only have reader ports and writer ports; no other type of port may be added to a filter. If one were to naively attempt to unify a filter with some other element, the algorithm presented thus far would lead to an illegal component.

Such “auxiliary” restrictions are themselves partial information about that elements in a fragment. Like other sorts of partial information, the restrictions that apply must be maintained and preserved through unification. That is, the restrictions that apply to either unificand must apply to the result of unification.

To handle this issue, we must add a check to the algorithm to make sure that any substitution satisfies the additional constraints. In practice, this capability is generally provided by tools that are knowledgeable about the contextual style and design constraints.

8. Implementation

We have implemented an extension to the Aesop architectural development environment that supports unification as defined above. The Aesop system is a family of architectural development environments that share a common graphical interface, a database in which architectural designs are stored, and a repository of reusable design elements.

In our implementation of unification, placeholder elements are displayed in a lighter color than normal elements. To attempt to unify two elements, the user drags one element over another element of the same architectural type, and the system automatically attempts to unify them, or reports that they cannot be unified. Fragments can be stored in the reuse repository [MG96], where they may be retrieved and unified with other fragments or an existing design.

The Aesop system also provides by default many of the feature-specific unification hooks that were discussed in Section 7.3. In particular, it provides built in resolution for element names, ports, roles and representations. Other feature-specific unification hooks can be provided through environment customizations.

9. Open Issues

The application of unification to software architecture raises a number of issues that are ripe for further investigation and future research.

9.1. Tools and Environments

While having a well-specified algorithm for unifying two design fragments is a necessary starting point, there are many questions about how best to exploit these ideas in the construction of practical system development tools.

First is the question of interface. How should design fragments be visually presented? Are placeholders distinguished, and if so how? How should a user indicate that two fragments are to be unified? (As noted above, in our own Aesop environment we use a drag and drop interface. But there are many other alternatives.)

Second, is the question of persistence: should a design fragment persist after it has been unified into some larger design? For example, should it be possible to determine what design fragments were used to create a given design?

Should it be the case that the user can later directly modify a design even if it violates some aspects of the design fragment? Should an environment provide an “undo-unification” operation?

9.2. Other Applications of Architectural Unification

We have focused on the use of unification for combining existing design fragments. But there are other applications that one might make of the same idea. One is the use of unification for search. Suppose you are attempting to find the part of a design that has a certain architectural form Unification could be used to match a partially filled out architectural skeleton against an existing design to locate all occurrences of it.

A related application would be to search for problem areas. Frequently it is known that certain kinds of architectural structures are a bad idea. (For example, you might know that there are typically performance bottlenecks whenever three specific types of components are interconnected.) Again a query could be phrased in terms of an architectural fragment that captures the bad design.

A third application of unification could be in terms of classification. Since unification determines when one structure specializes another, unification could be used to organize and index repositories of design fragments. That is, fragments can be related by the fact that there exist unifications of them.

9.3. Algorithm Extensions

As presented, unification assumes that the different features of an architectural element are unrelated. In practice this is often not the case. For example certain attribute values may depend on others. It would be useful if the algorithm could be extended to support these dependencies. In the case where the dependencies are known and functional, it is possible to structure the algorithm so that it calculates the non-derived values, and then uses recomputed derived values to complete the unification. More general cases of dependency, however, may require more sophisticated modifications.

10. Conclusion

Based on the observation that architectural design fragments are often combined into larger architectures we have presented a way to represent incomplete architectural designs and an algorithm for combining two or more such fragments. To combine design fragments that may overlap, we represent design fragments as pieces of architecture that may contain placeholder elements. The technique of unification from predicate logic provides an effective way to think about the combination of architectural placeholders with other design elements. However, several enhancements to the basic unification algorithm are necessary to use unification in the domain of software architecture. With these enhancements, architectural unification promises to be an effective tool for the use of design fragments in architectural design.

References

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In *Proceedings of ECOOP’94*, 1994.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[BO92] Don Batory and Sean O’Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[Bus93] Frank Buschmann. Rational architectures for object-oriented systems. *Journal of Object-Oriented Programming*, September 1993.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[GMW95] David Garlan, Bob Monroe, and David Wile. ACME: An interchange language for software architecture. Technical Report CMU-CS-95-219, Carnegie Mellon University, 1995. In preparation.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

[Kay79] Martin Kay. Functional grammar. In *5th Annual Meeting of the Berkeley Linguistic Society*, 1979.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[MG96] Robert T. Monroe and David Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, April 1996.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering*. ACM Press, October 1996.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

Appendix A: ACME Description of Design Fragment

This demonstrates the representation of the design fragment of Figure 1a in the architectural description language ACME.

```
Component UI1 = {
    Property name = "User Interface";
    Ports {
        Error_handler = {
            Property name = "Error Handler"
            Property type = file_handle
        }
        data_export = {
            Property Name = "Data export"
            Property type = ACME_output
            }
        data_import = {
            Property name = "Data import"
            Property type = ACME_input
            }
        }
    Property Placeholder = true;
    }


Component Repository = {
    Property name = "Reuse Repository";
    Ports {
        data_export = {
            Property Name = "Data export"
            Property type = ACME_output
            }
        data_import = {
            Property name = "Data import"
            Property type = ACME_input
            }
        }
    }
Connector exporting_data = {
    Property name = "Exporting Data Connector";
    Roles {
        acme_sink = {
            Property name = "Sink"
            Property type = ACME_sink
            }
        acme_source = {
            Property name = "Source"
            Property type = ACME_sink
            }
        }
    }
Connector importing_data = {
    Property name = "Importing Data Connector";
    Roles {
        acme_sink = {
            Property name = "Sink"
            Property type = ACME_sink
            }
        acme_source = {
            Property name = "Source"
            Property type = ACME_sink
            }
        }
    }
Attachments {
    UI1.ACME_output to exporting_data.acme_source
    UI1.ACME_input to importing_data.acme_sink
    repository.ACME_output to importing_data.acme_source
    repository.ACME_input to importing_data.acme_sink
    }
```