

Reasoning about the Composition of Heterogeneous Architectures

Ahmed Abd-Allah, Barry Boehm

(aabdalla@cs.usc.edu, boehm@sunset.usc.edu)

USC Center for Software Engineering

Computer Science Department

University of Southern California

Los Angeles, CA 90089-0781

Abstract: A persistent problem in software engineering is how to put software systems together out of smaller subsystems: the problem of software composition. The emergence of software architectures and architectural styles has focused attention on a new set of abstractions with which we can create and compose software systems. We examine the problem of providing a model for the composition of different architectural styles within software systems, i.e. the problem of composing heterogeneous architectures. We describe a model of pure styles and of their composition. We provide a disciplined approach to the process of architectural composition, and techniques for using the approach to determine architectural constraints and the conditions under which systems will fail to be composed.

Keywords: software architecture, styles, composition, constraints

1.0 Introduction

A persistent problem in software engineering is how to put software systems together out of smaller subsystems, the problem of software composition. There are many levels of granularity at which this problem can be tackled. For example, some of the earliest software engineers dealt with systems at the machine language or assembly language level of granularity. Succeeding engineers addressed the composition problem at a coarser granularity using higher level programming languages. The emergence of *software architectures* and *architectural styles* has introduced a still coarser level of granularity (and higher level of abstraction) at which we can create and compose software systems.

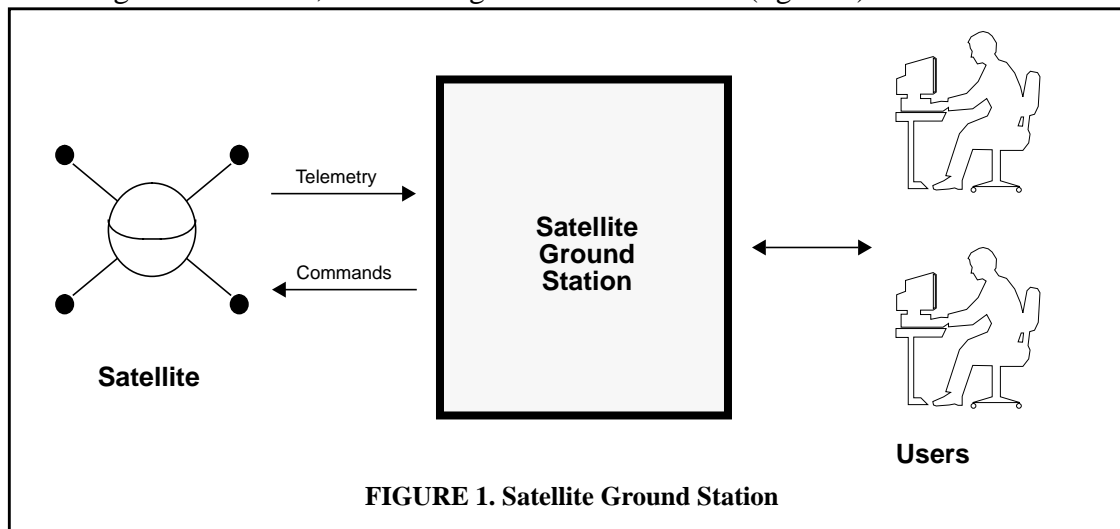
One of the earliest discussions of software composition at the architectural level was first mentioned by Garlan and Shaw in their seminal paper describing some architectural styles [GARL93]. In that paper, the authors state that “most systems typically involve some combination of several styles.” These types of systems are termed to have *heterogeneous* architectures. However, the focus of the paper is more on introducing pure architectural styles rather than on addressing the problem of composing heterogeneous architectures. There are twelve styles explicitly mentioned by the authors:

- layered
- distributed processes and threads

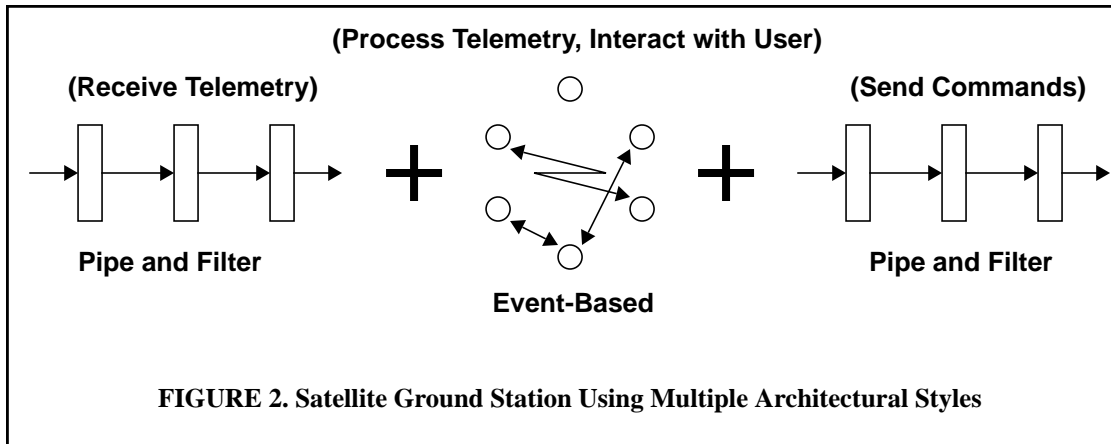
- pipes and filters
- object-oriented
- main program/subroutines
- repositories
- event-based (implicit invocation)
- rule-based
- state transition based
- process control (feedback)
- domain-specific
- heterogeneous

Taken together, the focus of each of these styles does not describe a clear smooth continuum; i.e. there is no clear and common set of attributes that each style makes definitive choices on to distinguish itself from other styles. Some styles focus on certain attributes that are not addressed by other styles. For example, the layered style is concerned with constraining the topology of connections, whereas the repository style merely says that at the heart of the system is a central repository.

Architectural styles are a powerful means for grasping the design of large software systems. Though software designers may not precisely articulate the architecture of the systems they design, almost all successful systems will rely on one or more architectural styles. It is likely that more than one style is used, especially if the system is large. Consider the problem of designing a satellite ground station for a representative satellite. The ground station is responsible for receiving satellite telemetry, processing the telemetry, interacting with the users, and sending satellite commands (figure 1).



One possible design for the system might partition the ground station into three distinct parts (figure 2). For receiving telemetry, the architect may choose a streaming pipe and filter style to handle the more or less constant flow of repetitive data. The part of the



station responsible for sending commands up to the satellite might be designed as another pipe and filter subsystem. The remainder of the system which is responsible for interacting with the users and processing the telemetry might be based on the event-based style to achieve high flexibility. This high level description sounds very attractive on paper, yet there are a great deal of important details that are hidden behind the ‘+’ signs!

The hidden details are implicit constraints within each style. Typically, architects may rely on an ad hoc method to compose different styles together, trusting their personal experience to adequately address the implicit constraints. In order to provide architects with a *systematic* approach for composing these styles together, it is useful to construct a theory and/or a tool to aid the architect during composition.

1.1 The Problems of Architectural Composition

Different researchers are beginning to address the problems of architectural composition. Some have attempted to construct a theory of composition which can be used to prove the correctness of architectures in one style being implemented in another style. Others have tried to build tools that allow construction of systems from architectural specifications.

Our aim is to build on the work of Garlan and Shaw, among others, by formally addressing software composition using architectural styles. We are specifically interested in generating a model of the composition of architectural styles within large software systems. In spite of scoping the broad problem of composition by focusing on styles, there are a number of formidable subproblems,

- the number of useful architectural styles is large
- the implicit properties of styles are by their very nature difficult to find
- architectural styles often focus on different properties
- composition can be achieved in many different ways

Each of these subproblems is addressed by our approach.

There are several benefits to be gained by studying this problem of architectural composition using styles. An architectural style is a partial starting point for organizing

the design of a system. It provides a baseline set of entities and constraints that the architect can leverage to solve the problem at hand. By studying the problem of composition, we are forced to examine individual styles in a precise manner, and this helps to clear some of the prevailing confusion on the definition of styles and architectures.

A most significant benefit is the capability of integrating architectural style perspectives with other architecture-level perspectives, such as domain-specific software architectures. In figure 2, for example, analysis of how best to align architectural styles with portions of satellite ground station domain architectures can provide powerful capabilities for reasoning about properties of existing and contemplated satellite ground station architectures. Another benefit involves tools which allow high-level composition of styles within single systems, and which may generate code as well. Finally, we can use our knowledge of styles to evaluate the strength of COTS packages to support specific architectures.

1.2 Approach

Our approach to tackling the problems described above can be summarized in two broad steps:

- create models of a number of pure architectural styles to gain greater insight, and to generate some ‘working material’ (section 2.0)
- create a model of composition based on the styles generated in the prior step (section 3.0)

The purpose of these models is to show the conditions under which styles will or will not be successfully composed with each other, and to establish the basis for a CASE tool which the architect can use to specify systems and generate code from those specifications.

Our approach addresses all of the subproblems mentioned previously. One of the thorniest issues to deal with is the sheer number of styles. There are two possible ways to address this: either look for an underlying common style, or reduce the number of styles and how to compose them with a suitable rationale. We have chosen to take the latter approach, finding the former to be extremely hard at this stage. By focusing initially on a representative subset of styles, and by postulating certain properties about styles and how they are composed, we reduce the complexity associated with the large number of choices to make.

Another subproblem lies in identifying the implicit properties of individual styles. In the process of creating our formal models, we were able to identify a set of properties and constraints per style. Our intent to build a tool based on the models was a powerful incentive to capture the ‘right’ set of properties. After describing a small set of styles, we were able to better identify where the styles differed and where they were similar. We also found out that styles do not discuss many important architectural issues typically associated with systems integration, so we began to address these issues as well.

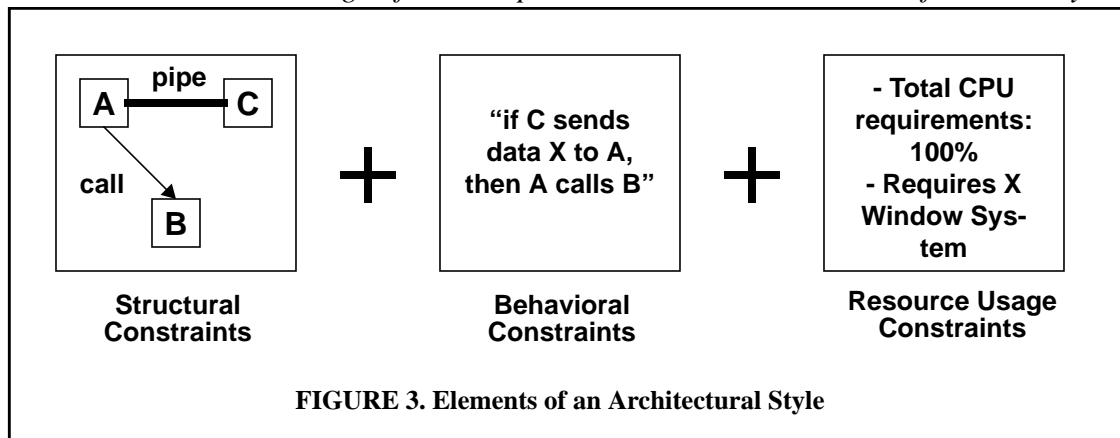
1.3 Previous Work

The definition of software architecture, and of architectural styles, is still in flux. Several similar but not identical definitions have been put forward ([PERR92], [GARL93], [TRAC94], [GACE95]). Architectural composition has been addressed by three different areas: architectural description languages (ADL's), CASE tools, and theoretic approaches. Many different ADL's have been generated for different domains, all with their strengths and weaknesses for supporting composition. Examples include ControlH/MetaH ([BINN]), Rapide ([LUCK94]), and Wright ([ALLE94]). There is currently no ADL whose purpose is to facilitate heterogeneous composition. CASE tools for architectural composition include UniCon ([SHAW94]), Aesop ([GARL94]), and UNAS ([ROYC91]). Although all three support heterogeneous architectures to varying degrees, none of them offers an underlying model for heterogeneous composition (Aesop has a strong model for homogeneous composition however). Finally, a theoretic approach to heterogeneous composition was recently proposed in which the correctness of a composition could be established ([MORI94]). However, the approach assumes that composition never affects an abstract architecture which is transformed into a concrete architecture, a restrictive assumption for practical purposes.

2.0 Framework for Understanding Architectural Styles

2.1 Elements of Architectural Styles

Hardware designers typically model the systems they build in terms of two different types of models: structure and behavior [MILN94]. Software architectures and software architectural styles can be described in a similar but not identical manner. An architectural style is formally represented as *a collection of constraints on the structure, behavior, and resource usage of the components and connectors in a software subsystem*



(figure 3). The structural view of a style describes the components and allowed interconnections between them. All the possible data and control transfers are explicitly identified. The behavioral view places constraints on the types and order of actions which the components and connectors may engage in. The resource usage view places constraints on a variety of systems integration issues such as network protocols, cpu type, etc. In this report, we discuss the structural and resource usage views of styles, however we do not address

behavior (see [ABDA95]).

The formal language used in this work is the Z notation [SPIV92, WORD92, POTT91]. Z was originally created for specifying and designing software, however it has also been used to model other phenomena, including architectural styles ([ABOW93]). It is based on set theory and first order logic, and provides the *schema* construct as the main building block for writing formal specifications. While it is well suited to model state and incremental changes of state (i.e. operations), its primary weaknesses are a lack of direct support for describing sequences of operations as well as object-oriented hierarchies of schemas. However, both weaknesses can be indirectly worked around ([HALL94], [STEP92], [SUFR90]).

In this report, we use a plain text form of Z called ZSL [JIA94]. Different fonts are used to improve readability:

```
this is formal ZSL text  
% this is an embedded comment
```

2.1.1 Modeling the Structural View of a Style

One important difference between hardware and software architectures is that the structure of many software architectures typically *changes* during its operation. The structure of a hardware architecture is assumed to stay constant during the period of its execution. In software, the situation is different. The structural view of a software architectural style describes the vocabulary of components and connectors from which a system may be built using that style (see figure 4 for an example from the distributed processes style). In

```
schema Process  
  name : Name  
  % the name of the process  
  initialthreads : P Thread  
  % the threads initially executing in the process ('P' means 'set of')  
  allowedthreads : P Thread  
  % the set of allowed threads that may execute in the process  
where  
  # allowedthreads >= 1  
  % each process has at least one thread  
  # initialthreads >= 1  
  % each process starts with at least one active thread  
  initialthreads subeq allowedthreads  
  % the initial threads are a subset of the allowed threads  
end schema
```

FIGURE 4. Example of Structural Component

addition to data connectors, there are control connectors which can dynamically change the structure. Components and connectors can be added and removed while the system is running. For example, in the main/subroutine style, a procedure call has the effect of adding a new instance of some procedure to the current structure (the current activation tree). In the distributed processes style, a new circuit may be created then destroyed between two threads as another example. This notion of a dynamic structure is important for some

styles, and not important for others (those which are restricted purely to data transfers). For the remainder of this report, we emphasize the static structure; for a longer discussion of dynamic structures, see [ABDA95].

2.1.2 Modeling the Resource Usage View of a Style

The original description of software architectural styles by Garlan and Shaw only focused on the structural organization of components and connectors. This is the essence of styles, but there are other views that can be considered at the architectural level ([GACE95]). One such view is concerned with *resource usage*, those issues and problems that are typically associated with integration projects. Also, we have found that resource usage considerations are very important in determining the best match of architectural styles to portions of a domain-specific software architecture.

This view covers a wide variety of issues. A classic example of some of these issues can be found in a recent paper describing the experience of integrating four complex software systems together [GARL95]. Researchers in systems integration have developed many different techniques for addressing these issues [NG90].

Many styles are silent regarding constraints on the resource usage of a system. We are proposing, on the other hand, to consider high-level integration problems at the architectural level. Specifically, we are concerned with the following common issues,

- component and system issues: size, cpu usage, memory usage, cpu type, operating system requirements, implementation language
- data connector issues: format, rate, buffered, synchronous, streamed, duplex, multiplicity of senders and receivers, explicitly identified sender or receiver, reliable, duration of transfer, underlying protocols
- control connector issues: complete or partial (e.g. spawn versus call), explicitly identified sender or receiver, returns or not, duration of transfer, underlying protocols

Most of the resource usage issues are currently modeled as simple attributes that can have only a small range of values (figure 5).

```
% data type to represent different types of processors
CUtype ::= sparc | 80x86 | 680x0 | PowerPC

% resource: central processing unit (CPU) requirements
schema CPUResource
  cputype : CUtype
  % the type of CPU (e.g. SPARC, Pentium, 68040, etc)
  cpuload : N1
  % average load as a percentage of the CPU
end schema
```

FIGURE 5. Example of a Resource

2.1.3 Joining the Views Together

We have referred to two independent views of architectural styles: structure and resource usage (there are other views we are ignoring, notably behavior). To tie these two views together, we will describe each architectural entity (component, connector, etc.) with a single schema comprised of its structure and resource attributes (figure 6). The

```

schema SomeStructuralEntity
    % ...structure attributes...
    % ...resource usage attributes...
where
    % ...structure constraints...
    % ...resource usage constraints...
end schema

```

FIGURE 6. Representing Multiple Views: Structure and Resource Usage

schema will also contain constraints on the attributes as part of its property. In this way, the two views are packaged together in a formal manner, ready for composition.

2.1.4 Example: Event-Based Style

A popular example of the event-based style is the class of applications built for the X Window System, a graphical interface for Unix based workstations [SCHE86, NYE92]. It is the basis for our representation of this style. One important thing to note about X and therefore the semantics of our representation is that it is also object-oriented. Thus, it is not as subjectively ‘pure’ as the others. Informally, this style is characterized by many event-based objects producing events which are sent to a centralized event manager which in turn forwards the events to other objects. The receiving objects may invoke certain procedures in response to the input events - and thereby produce a new set of events. Events are transferred between objects via the system over an implicit globally available network.

The set of components and connectors available in an event-based system is given in table 1. We also show the typical operations associated with each component and con-

Entity	Type	Operations
Data Structure	Component	read, write
Procedure	Component	activate, suspend, compute
Event	Component	
Event queue	Component	add event, remove event, peek
Object	Component	produce event, act on event
System	Component/ Connector	instantiate object, destroy object, call procedure, procedure return, invoke callback, callback return, forward event to object, receive event from object
Procedure Call	Connector	call, return

TABLE 1. Components, Connectors, and Operations of the Event-Based Style

connector. The operations are useful for specifying behavioral constraints. The formal representation of the system component is given in figure 7.

```

schema EventBasedSystem
  name : Name
    % the name of the system
  allowedobjects : P EventBasedObject
    % a set of event based objects
  initialobjects : P EventBasedObject
    % the initial set of running objects
  recognizedevents : P EventType
    % the set of events the manager can respond to
  registry : EventType <-> EventBasedObject
    % a relation showing which events are acted on by which objects
  CPUResource
    % other resource usage factors not listed
where
  initialobjects subeq allowedobjects
    % the initial set of objects is a subset of the allowed set
  dom registry subeq recognizedevents
    % the events acted on by objects are events which the manager can
      recognize
  ran registry subeq allowedobjects
    % registered objects are in the set of allowed objects
end schema

```

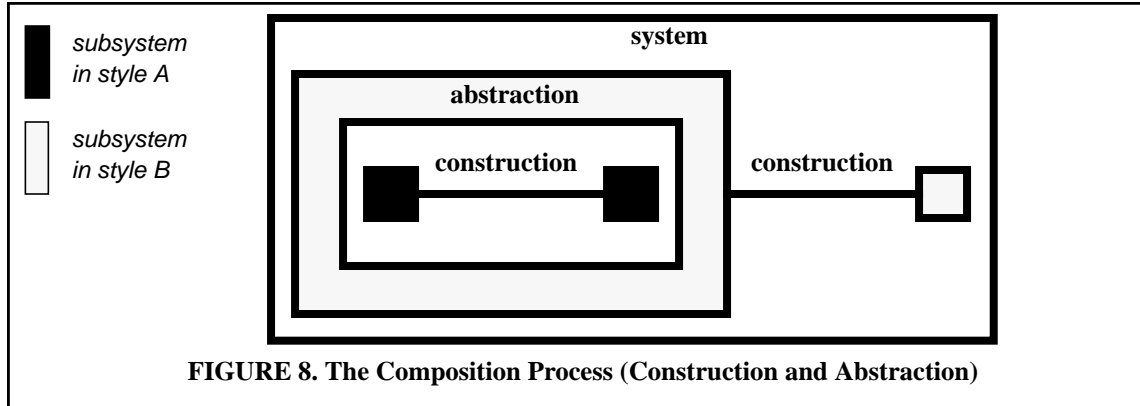
FIGURE 7. Event-Based System

3.0 Composition of Architectural Styles

There are few large software systems that adhere purely to a single architectural style. Most systems exhibit a mixture of styles in their composition. Garlan and Shaw describe these systems as having *heterogeneous* architectures [GARL93]. Creating systems with heterogeneous architectures is not an easy task. One simplifying assumption we make here is that the architectures of the subsystems to be composed are entirely visible and available (for example, in an architecture composition CASE tool in which the entire system is designed). We assume that descriptions of the structural, behavioral, and resource usage views of the subsystem architectures are given. This is different from another type of composition problem, the problem of integrating already existing software having only the executables and no architectural descriptions [NILS90].

3.1 Operations Involved in Composition

It is important to establish a disciplined approach to the process of architecture composition. Taking our cue from hardware [MILN94], we can define this process in terms of essentially two operations. Specifically, a software system architecture is composed by applying a sequence of construction and abstraction operations on some initial subsystems (figure 8). These are the only two mechanisms that we use to compose styles together in a single architecture. Informally, we may think of construction as bringing together elements into a larger whole, and abstraction as hiding some properties of a group of elements - and perhaps replacing them with other properties. This process of interleav-



ing construction with abstraction is not new to software of course, having widespread use at other stages of software design. Here, we are formally specifying the systematic application of these operations on systems described in terms of their architectural styles.

Since we have described architectural styles using different views, the operations of construction and abstraction need to address these views. If two subsystems are constructed, their construction must be specified structurally, behaviorally, and with respect to their resource usage. A similar situation exists when a subsystem is abstracted. As our understanding of software architectures and architectural styles gets broader and deeper, the operations will need to be extended to accommodate any additional types of views.

3.2 Architectural Constraints under Composition

The central problem with architectural composition is resolving constraint mismatches ([GARL94]). Constraint mismatches can arise as a result of two general cases during composition:

- one or more subsystem's constraints conflict with an imposed global constraint
- two or more subsystem's constraints conflict with each other

For example, in the first case, an architect may find that a library of components does not meet certain imposed performance requirements. In the second case, two libraries may use different network protocols for data transfers.

In both cases of constraint mismatches, a major obstacle is presented by the task of identifying and representing constraints. Since architectural styles have many different views, and the styles themselves are not tightly related, it is useful to begin categorizing the types of interesting constraint mismatches that might occur. After examining four styles in depth (main/subroutine, distributed processes, event-based, pipe and filter), we have identified the following categories (presented with brief examples):

1. A name is shared between two systems that are to be composed.
 - example: two separate main/subroutine systems that are to be constructed together have two different procedures having identical names.
2. A layering constraint is violated.
 - example: a hidden layer is accessed by another system during construction.

- example: encapsulated data is accessed directly during abstraction from an object to a process.
 - example: a resource assumed to be exclusively accessible by a system isn't after construction with another system.
3. An input is used as an output (or vice-versa) during composition.
 - example: a pipe going from one input socket to another input socket bridges two pipe and filter subsystems that are constructed with each other.
 4. A connection is made during composition to an entity that can only support a single connection, and which has already been connected.
 - example: a circuit is added to an already connected socket while constructing two distributed processes systems together.
 5. A component that should be connected is not.
 - example: a socket which is neither a system input nor a system output is left dangling with no pipe to it.
 6. The methods/protocols used by two systems for data and/or control transfers are incompatible during composition.
 - example: the construction of two distributed processes systems is hindered by the fact that the first uses message-passing and the second uses distributed shared memory for data transfers.
 7. The data formats (or value ranges) used by two systems are incompatible during composition.
 - example: the sets of recognized events for two event-based systems to be composed are not identical.
 - example: the rules of two rule-based systems to be composed have different rule formats and allowed actions.
 8. A resource independent of behavior is overused after composition.
 - example: the sum of two pipe and filter subsystem's static code size exceeds available space after composition.
 - example: a distributed process subsystem abstracted to a filter fails to meet an imposed average cpu load constraint.
 9. A resource dependent on behavior is overused in a particular scenario.
 - example: a sequence of operations exceeds a maximum allowed time requirement during composition.
 10. A resource that must be identical over two systems to be composed isn't.
 - example: the underlying platforms required for two systems to be composed are incompatible.

These constraint mismatches can occur during both construction and abstraction.

3.3 Construction Operation

The construction operation takes as input the architectures of two separate subsystems and some bridging connectors, and produces as output the interconnected archi-

ecture of a single system (i.e. there is at least one connection bridging the two input subsystems). All of the original components and connectors of the input subsystems are present in the output system. Thus the constructed output system describes the possibility of a connection between dynamic instances of the two input subsystems. All other connections are not allowed.

In many cases, we will want to construct a system from two subsystems of differing styles. In our approach, we accommodate this as two-step process. The construction operation is limited to pairs of operands (subsystems) of the *same* style. In order to compose a system that is built from subsystems of different styles, the abstraction operation must first be used on at least one of the subsystems to bring both into a common style (figure 8). For each style, the construction operation is similar but not identical. A simplifying assumption we are making is to further limit the construction operation to be between subsystems (e.g. constructing a pipe with a socket is not considered, but constructing a single pipe and filter system from two other pipe and filter subsystems is considered).

3.3.1 Example: Event-Based Construction

For event-based systems, the construction operation joins two subsystems together simply by bringing them together. The implicit global connectivity of this style unites the two subsystems together. The inputs to the operation are the two subsystems and the name of the new system to construct. The output is the new interconnected system. An example of a constraint that must be satisfied is that the two subsystems must recognize the same set of events (e.g. both are X Window based as opposed to trying to construct an X Window application with a Microsoft Windows application).

```

% structural construction for event-based style
global
    % a function mapping pairs of subsystems to systems
    constructEB : (EventBasedSystem cross
        EventBasedSystem cross
        % the two input subsystems
        Name)
        % the new name of the system
        --> EventBasedSystem
        % the output system

axiom
    % constrain the new system to obtain its properties from the sub-
    % systems and the new interconnections
    forall sys1, sys2, newsys : EventBasedSystem;
        newname : Name
        @
        (constructEB (sys1, sys2, newname) = newsys
        <=>
        newsys.name = newname and
        % the new system has the new input name
        newsys.allowedobjects = sys1.allowedobjects union
        sys2.allowedobjects and
        % the system's set of allowed objects is the union of the sub-
        % systems' allowed objects

```

```

newsys.initialobjects = sys1.initialobjects union
sys2.initialobjects and
% the system's set of initial objects is the union of the sub-
  systems' initial objects
sys1.recognizedevents = sys2.recognizedevents and
% the subsystems' recognized events are identical
newsys.registry = sys1.registry union sys2.registry and
% the system's set of registered events is the union of the sub-
  systems' registered events
sys1.cpload + sys2.cpload <= 100 and
% the cpu load of the two systems together must be less than
  100%
newsys.cpload = sys1.cpload + sys2.cpload)
% the new system cpu load is the sum of the old loads

end axiom

```

3.4 Abstraction Operation

The abstraction operation provides systems in one style with an interface which is of a different style (i.e. it provides heterogeneity). For example, we might use it to provide a set of processes and threads with a single filter interface. Before we can precisely describe the inputs and outputs of the abstraction operation, however, we need to examine it carefully because it is more complicated than construction. In fact, the abstraction operation for just *one* pair of styles is a difficult problem by itself if we consider that reengineering from legacy systems to object-oriented systems is a form of abstraction.

Whereas construction is limited to between subsystems having the same style, abstraction is used to provide a subsystem in one style with an interface of a possibly *different* style. Having no restriction on the type of operands, the abstraction operation is inherently more difficult to capture than construction. There are at least two specific reasons for this:

- the construction operation's set of inputs grows linearly with the number of styles but the abstraction operation's set grows polynomially (figure 9)
- construction only considers pairs of systems, however abstraction is often done not only from system to system, but also from system to component or system to connector

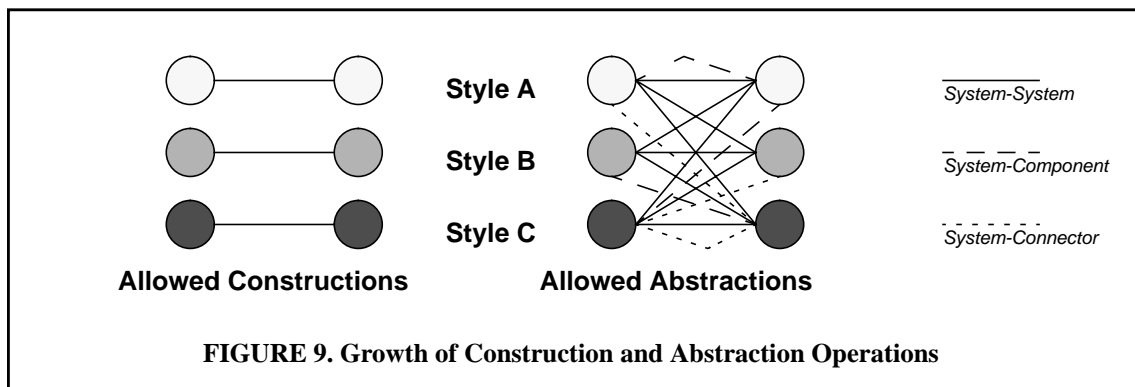


FIGURE 9. Growth of Construction and Abstraction Operations

Any practical approach to software composition at the architectural level will have to

address these two problems associated with the abstraction operation.

3.4.1 Reducing the Difficulty of Abstraction

To reduce the difficulty of abstraction, we first make an observation: not all compositions are attempted by architects. In other words, just because it is theoretically possible to abstract a system in one style into another system of a different style does not mean it should be done (however, if COTS is being used to implement the system, then the architect may be required to composed certain styles, and may want information on the possibility or difficulty of abstracting a COTS product of one style into another).

As mentioned above, it is theoretically possible to abstract whole systems into a single component or connector (and we are ignoring other important cases like component to component), and this creates a potentially overwhelming number of cases to consider. We can reduce the number by limiting the allowed abstractions in a way that does not limit the architect unnecessarily. The reduced set eliminates relatively ‘useless abstractions’ (e.g. a main/subroutine system to a thread spawn), but it keeps the common and useful cases.

The problem now becomes to specify a limited set of abstractions that are useful and which will allow us to express *most* architectures. To resolve this problem, we examined four styles in depth, and *generalized* their components and connectors (table 2).

Main/Sub.	Dist. Processes	Pipe & Filter	Event-Based	Generalized
Data Structure	Data Structure	Data	Data Structure Event	<i>Data Component</i>
	Socket	Socket	Event Queue	<i>Data Socket</i>
	Circuit	Pipe		<i>Data Connector</i>
Procedure	Thread Process	Filter	Procedure Event Object	<i>Control Component</i>
				<i>Controldata Socket</i>
Call	Call Spawn		Call Invoke Callback	<i>Controldata Connector</i>
M/S System	D/P System	P/F System	E/B System	<i>System</i>

TABLE 2. Generalizing components and connectors

There are seven generalized entities that we can recognize from the four styles, informally defined here:

- A *data socket* is typically associated with a control component, and represents the latter’s entry and exit ports for data components during data transfers.
- A *controldata socket* is typically associated with a control component, and represents the latter’s ability to participate in a transfer of control along that socket, possibly transferring data as well.
- A *data connector* models the potential for two or more control components to engage in data transfers amongst themselves.

- A *controldata connector* models the potential for two or more control components to engage in control transfers (possibly with data) amongst themselves.
- A *data component* models data that is used to store state and/or is passed around in data transfers (communication).
- A *control component* is executed by the underlying machine and which can initiate (and respond to) data and control transfers. It is assumed to have a single thread of control.
- A *system* is any collection of control components which can potentially engage in a control or data transfer - i.e. there is at least one control or data connector joining any two components.

It may be that with the examination of additional styles we would discover other generalized entities.

These generalized entities can be used to constrain the inputs of the abstraction operation. We will allow abstraction to occur between the following pairs of operands *only*:

- a style-specific component/connector to another style-specific component/connector which shares the *same generalized* component/connector. Example: a procedure to a filter.
- a controldata connector to a data connector for any two styles. Example: a call to a circuit.
- a system to a control component for any two styles. Example: a pipe and filter system to a thread.
- a system to a data connector for any two styles. Example: a main/subroutine system to a pipe.

This restricted set of abstractions targets the most common and useful cases of abstraction from one style entity to another.

Just as with the construction operation, there is no immediately obvious *single operation* for abstracting any style to any other style (even with all the restrictions described above). In fact, the number of ways that a particular abstraction could be implemented is also large, an issue we are still grappling with. The abstraction operation is therefore different for every pair of input types it is used on. In the future, we would like to look for commonalities between different abstractions.

3.4.2 Example: Event-based System Abstracted to a Filter

One possible abstraction would be to provide an event-based system with a filter interface to construct it with other filters in a larger pipe and filter system. The abstraction is implemented by adding a *hybrid* component to the event-based system which acts as both an event-based object and a filter. This hybrid component acts as the filter interface to the rest of the system. The intent is to provide a component which can translate filter inputs to output events for the system to process, and which can also translate event inputs

to filter outputs for the external world.

```
schema HybridEventBasedObjectFilter  
  object : EventBasedObject  
  filter : Filter  
  CPUResource  
end schema
```

Like construction, the abstraction operation is modeled as a function. The inputs in this particular case are the system to be abstracted, an object, and a filter - the latter two are used to describe the necessary hybrid component. The output is the new event-based system containing a new object which can also act as a filter.

```
% structural abstraction of event-based system to filter  
global  
  abstractEBSystemtoPFFilter :  
    (EventBasedSystem cross  
      % the input subsystem  
      EventBasedObject cross Filter)  
      % the basis for the hybrid component  
      --> EventBasedSystem  
      % the output system  
  
  axiom  
    % constrain the new system to obtain its properties from the old sub-  
      system and the desired interface  
    forall sys, newsys : EventBasedSystem;  
    newobject : EventBasedObject; filterinterface : Filter  
    @  
    (sys, newobject, filterinterface) abstractEBSystemtoPFFilter  
    newsys  
    <=>  
    (exists hybrid : HybridEventBasedObjectFilter |  
      hybrid.object = newobject and hybrid.filter = filterinterface  
    @  
    newsys.name = sys.name and  
    % the new system has the same name as the old  
    newsys.allowedobjects = sys.allowedobjects union {newobject}  
    % the new system's allowed objects are the old's plus the hybrid's object  
    and  
    newsys.initialobjects = sys.initialobjects union {newobject} and  
    % the new system's initial objects are the old's plus the hybrid's object  
    newsys.recognizedevents = sys.recognizedevents union  
      newobject.inpotevents union newobject.outpotevents and  
    % new recognized events incorporates those of the hybrid  
    newsys.registry = sys.registry union  
      {evtype : EventType; o : EventBasedObject | evtype in  
        newobject.inpotevents and o = newobject} and  
    % new registry includes new object's input events  
    sys.cpuload + hybrid.cpuload <= 100 and  
    % the cpu load of the input system and the new component together  
      must be less than 100%  
    newsys.cpuload = sys.cpuload + hybrid.cpuload)  
    % the new system cpu load is the sum of the old loads  
  end axiom
```

4.0 Applications of Model

We are currently beginning implementation of a tool which sits upon our model of architectural styles. This tool will allow architects to analyze their composed architectures for constraint mismatches, and also to determine the ability of a COTS package to satisfy the constraints of a given architecture. Since we are just now beginning implementation of this tool, we present instead two other applications of our model: an evaluation of an existing COTS package, and a description of the hierarchy of architectural styles.

4.1 COTS Evaluation

UNAS (Universal Network Architecture Services) is an architectural composition tool ([ROYC91]). It facilitates the development of large, distributed networks in either Ada or C++. There are two ways to use UNAS to build applications: either manually access the provided libraries in one's own code, or use a graphical interface for laying out an architecture (a 'skeleton') and subsequently fill in the user code. We only discuss the latter approach here. Much of our experience with UNAS is derived from using it to design the software architecture of a satellite ground station used as an example in the United States Air Force Academy ([LARS92]).

UNAS supports a mix of architectural styles. The main primitives are messages, sockets, circuits, tasks, and processes. Using the graphical interface, the architect lays out a set of tasks (threads) within processes, and connects the tasks together using sockets and circuits. This reflects the distributed processes style, however there is one major difference in that there is no connector which models control transfers. In other words, the graphical interface does not allow the architect to model a task spawning or calling another task. Only data transfers are modeled. This is in effect a simplifying assumption for the tool: the dynamic structure of the system is *always* the same as the static structure.

In addition to the distributed processes style, UNAS also supports the event-based style. The graphical interface allows the architect to specify actions for tasks to take on receipt of messages. Actions include calling user-defined procedures and sending out more messages. If we ignore this capability, we see that UNAS can be used to create systems in the pipe and filter style.

Composition of systems in UNAS using the graphical interface is limited to construction. There are no instantiations of the abstraction operation supported by the tool. This limits the tool to creating monolithic distributed-processes systems that cannot be 'hidden' by a single filter or other entity.

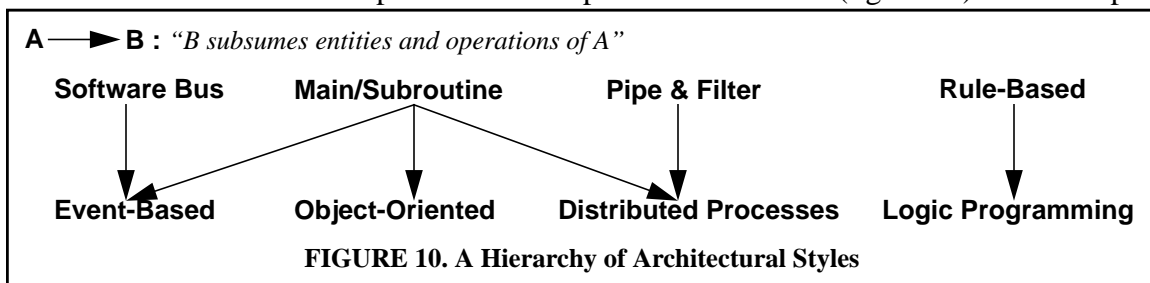
Thus, our formalisms enable one to determine the underlying reasons which have made a COTS tool such as UNAS a powerful composition capability for its chosen class of compositions, and also to determine classes of composition that the tool does not directly support.

4.2 A Hierarchy of Architectural Styles

Successfully composing heterogeneous systems will often depend on the relationships between the styles which are used. The original list of styles described by Garlan and Shaw was presented as a flat list, with no clear relationship between individual styles. It is possible however to hierarchically arrange the styles based on many different criteria. One possible hierarchy could be based on the entities and operations offered by a style. Styles that are linked in the hierarchy should be easier to compose with each other. In the course of investigating architectural composition, we enumerated the following high-level entities and operations of several styles:

- main/subroutine: data structure (read, write), procedure (activate, suspend, compute), system (call procedure, procedure return)
- pipe & filter: data, socket (add data, remove data), pipe (transfer data, transfer message), filter (send data, receive data, send message, receive message, compute), system
- software bus: message, communicant (send message, receive message), bus (transfer message)
- event-based: data structure (read, write), procedure (activate, suspend, compute), event, event queue (add event, remove event), object (produce event, act on event), system (instantiate object, destroy object, call procedure, procedure return, invoke callback, callback return, forward event, receive event)
- object-oriented: data structure (read, write), procedure (activate, suspend, compute), object, system (instantiate object, destroy object, invoke method, method return)
- distributed processes: data structure (read, write), socket (send message, receive message), circuit (transfer message), thread (activate, suspend, compute, send message, receive message), process (add thread, remove thread, activate, suspend), system (call thread, thread return, spawn thread, thread exit, call process, process return, spawn process, process exit, create circuit, destroy circuit)
- rule-based: fact, rule, interpreter (match condition, add fact, modify fact, remove fact), system (select rule, execute rule)
- logic programming: fact, rule, inference engine (unify term, add fact, modify fact, remove fact), system (prove goal, generate subgoals)

This information was enough to create a tree of styles, where the children of a node subsume the set of entities and operations of the parent and add to it (figure 10). For example,



the software bus style can be mapped into the event-based style by mapping messages into events, communicants into objects, and the bus into the system. A communicant's send-

message and receive-message operations are mapped into an object's produce-event and act-on-event operations respectively, and the bus's transfer-message operation is mapped into the system's receive-event/forward-event pair of operations.

The layered style mentioned by Garlan and Shaw cannot really be called a style as the others since it makes no definitive choices of what are the entities and operations. Hence, we prefer to consider layering as a type of constraint which can be applied to any style.

Note, however, that there is a large amount of information missing from the list above that is essential to the identity of some styles. For example, the notion of encapsulation is not captured as an entity or an operation for the object-oriented style, and neither is the notion of backtracking for logic programming. It is clear that if we look at different views of a style, different hierarchies may emerge.

5.0 Summary

We have presented several capabilities for reasoning about the composition of heterogeneous architectures. A framework for modeling many different styles is provided, dependent on three different views: structural, resource usage, and ultimately behavioral. A model of heterogeneous composition is also presented, which hinges on the two operations of construction and abstraction. For both operations, the emphasis was to identify the conditions under which composition will not work by looking for constraint mismatches between systems. A categorization of constraints based on the three views was presented. Finally, we showed how we used the model to evaluate a COTS package, and also to demonstrate a hierarchy of styles based on the generality of their entities and operations.

6.0 Acknowledgments

We would like to thank the following people for their comments on this work: Robert Balzer, Prasanta Bose, Ellis Horowitz, Eberhardt Rehtin, and David Wile. We would also like to acknowledge the Affiliates of the Center for Software Engineering for their support: AT&T, Bellcore, EDS, HP, IDE, Motorola, Rational, TI, Xerox, Sun, E-Systems, Hughes, Litton, Lockheed, Loral, Northrop Grumman, Rockwell, SAIC, TRW, DISA, USAF Rome Lab, US Army Research Labs, Aerospace, IDA, JPL, SEI, and SPC.

7.0 References

- [ABDA95] A. Abd-Allah. "Composing Heterogeneous Software Architectures", Technical Report, USC-CSE-95-502, University of Southern California, 1995, (URL - <http://sunset.usc.edu/TechRpts/heterogeneous.ps>)
- [ABOW93] G. Abowd, R. Allen, D. Garlan. "Using Style to Understand Descriptions of Software Architecture", *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Software Engineering Notes, ACM Press, December 1993
- [ALLE94] Allen R. and Garlan D. "Formal Connectors", Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994
- [BINN] Binns P., Englehart M., Jackson M., and Vestal S. *Domain-Specific Software Architectures for Guid-*

- ance, Navigation, and Control*. Honeywell Technology Center, Minneapolis, MN.
- [GACE95] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm. "On the Definition of Software Architecture", *ICSE 17 Software Architecture Workshop*, April 1995
- [GARL93] D. Garlan and M. Shaw. "An Introduction to Software Architecture" *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 1993
- [GARL94] D. Garlan, R. Allen, and J. Ockerbloom. "Exploiting Style in Architectural Design Environments", *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, Software Engineering Notes, ACM Press, December 1994
- [GARL95] D. Garlan, R. Allen, J. Ockerbloom. "Architectural Mismatch or Why it's hard to build systems out of existing parts", *ICSE 17 Software Architecture Workshop*, April 1995
- [HALL94] A. Hall. "Specifying and Interpreting Class Hierarchies in Z", *Z User Workshop 1994*, Workshops in Computing, Springer Verlag, 1994
- [JIA94] X. Jia. "ZTC: A Type Checker for Z User's Guide", Department of Computer Science, DePaul University, 1994
- [LARS92] W. Larson and J. Wertz (eds.). *Space Mission Analysis and Design*, Microcosm Inc., 1992
- [LUCK94] Luckham D., Augustin L., Kenney J., Vera J., Bryan D., and Mann W. *Specification and Analysis of System Architecture Using Rapide*. 1994
- [MILN94] G. Milne. *Formal Specification and Verification of Digital Systems*, McGraw-Hill Book Co., 1994
- [MORI94] M. Moriconi and X. Qian. "Correctness and Composition of Software Architectures", *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, Software Engineering Notes, ACM Press, December 1994
- [NG90] P. Ng, C. V. Ramamoorthy, L. Seifert, R. Yeh (editors). *Proceedings of the First International Conference on Systems Integration*, IEEE Computer Society Press, 1990
- [NILS90] E. Nilsson, E. Nordhagen, G. Oftedal. "Aspects of Systems Integration", *Proceedings of the First International Conference on Systems Integration*, IEEE Computer Society Press, 1990
- [NYE92] A. Nye (ed.). *X Protocol Reference Manual*, O'Reilly & Associates, 1992
- [PERR92] Perry D. and Wolf A. "Foundations for the Study of Software Architecture". *ACM SIGSOFT Software Engineering Notes*, Vol. 17, 4, October 1992
- [POTT91] B. Potter, J. Sinclair, D. Till. *An Introduction to Formal Specification and Z*, Prentice Hall International, 1991
- [ROYC91] Walker Royce and Winston Royce. "Software Architecture: Integrating Process and Technology", *TRW Space and Defense*, Summer 1991
- [SCHE86] R. Scheifler and J. Gettys. "The X Window System", *ACM Transactions on Graphics*, vol. 5, April 1986
- [SHAW94] Shaw M., DeLine R., Klein D., Ross T., Young D., and Zelesnik G. *Abstractions for Software Architecture and Tools to Support Them*. Carnegie Mellon University, Pittsburgh, February 1994
- [SPIV92] J. Spivey. *The Z Notation*, Prentice Hall International, 1992
- [STEP92] S. Stepney, R. Barden, D. Cooper (eds.). *Object Orientation in Z*, Workshops in Computing, Springer Verlag, 1992
- [SUFR90] B. Sufrin and J. He. "Specification, analysis and refinement of interactive processes", *Formal Methods in Human-Computer Interaction*, Cambridge University Press, 1990
- [TRAC94] Tracz W. ed. "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)". *ACM SIGSOFT Software Engineering Notes*, Vol. 19, 2, April 1994
- [WORD92] J. Wordsworth. *Software Development with Z*, Addison-Wesley Publishing Company, 1994