Foundations for the Study of Software Architecture

Dewayne E. Perry
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974
dep@research.att.com

Alexander L. Wolf
Department of Computer Science
University of Colorado
Boulder, Colorado 80309
alw@cs.colorado.edu

ABSTRACT

The purpose of this paper is to build the foundation for software architecture. We first develop an intuition for software architecture by appealing to several well-established architectural disciplines. On the basis of this intuition, we present a model of software architecture that consists of three components: elements, form, and rationale. Elements are either processing, data, or connecting elements. Form is defined in terms of the properties of, and the relationships among, the elements — that is, the constraints on the elements. The rationale provides the underlying basis for the architecture in terms of the system constraints, which most often derive from the system requirements. We discuss the components of the model in the context of both architectures and architectural styles and present an extended example to illustrate some important architecture and style considerations. We conclude by presenting some of the benefits of our approach to software architecture, summarizing our contributions, and relating our approach to other current work.

1 Introduction

Software design received a great deal of attention by researchers in the 1970s. This research arose in response to the unique problems of developing large-scale software systems first recognized in the 1960s [5]. The premise of the research was that design is an activity separate from implementation, requiring special notations, techniques, and tools [3, 9, 17]. The results of this software design research has now begun to make inroads into the marketplace as computer-aided software engineering (CASE) tools [7].

In the 1980s, the focus of software engineering research moved away from software design specifically and more toward integrating designs and the design process into the broader context of the software process and its management. One result of this integration was that many of the notations and techniques developed for software design have been absorbed by implementation languages. Consider, for example, the concept of supporting "programmming-in-the-large". This integration has tended to blur, if not confuse, the distinction between design and implementation.

The 1980s also saw great advances in our ability to describe and analyze software systems. We refer here to such things as formal descriptive techniques and sophisticated notions of typing that enable us to reason more effectively about software systems. For example, we are able to reason about "consistency" and "inconsistency" more effectively and we are able to talk about "type conformance"[1] rather than just "type equivalence".

The 1990s, we believe, will be the decade of *software architecture*. We use the term "architecture", in contrast to "design", to evoke notions of codification, of abstraction, of standards, of formal training (of software architects), and of style. While there has been some work in defining particular software architectures (e.g., [19, 22]), and even some work in developing general support for the process of developing architectures (notably SARA [8]), it is time to reexamine the role of architecture in the broader context of the software process and software process management, as well as to marshal the various new techniques that have been developed.

Some of the benefits we expect to gain from the emergence of software architecture as a major discipline are: 1) architecture as the framework for satisfying requirements; 2) architecture as the technical basis for design

[1] Conformance is used to describe the relationship between types and subtypes.

and as the managerial basis for cost estimation and process managment; 3) architecture as an effective basis for reuse; and 4) architecture as the basis for dependency and consistency analysis.

Thus, the primary object of our research is support for the development and use of software architecture specifications. This paper is intended to build the foundation for future research in software architecture. We begin in Section 2 by developing an intuition about software architecture against the background of well-established disciplines such as hardware, network, and building architecture, establish the context of software architecture, and provide the motivation for our approach. In Section 3, we propose a model for, and a characterization of, software architecture and software architectural styles. Next, in Section 4, we discuss an easily understood example to elicit some important aspects of software architecture and to delineate requirements for a software-architecture notation. In Section 5, we elaborate on two of the major benefits of our approach to software architecture. We conclude, in Section 6, by summarizing the major points made in this paper and considering related work.

2 Intuition, Context, and Motivation

Before presenting our model of software architecture, we lay the philosophical foundations for it by: 1) developing an intuition about software architecture through analogies to existing disciplines; 2) proposing a context for software architecture in a multi-level product paradigm; and 3) providing some motivation for software architecture as a separate discipline.

2.1 Developing an Intuition about Software Architecture

It is interesting to note that we do not have *named* software architectures. We have some intuition that there are different kinds of software architectures, but we have not formalized, or institutionalized, them. It is our claim that it is because there are so many software architectures, not because there are so few, that the present state of affairs exists. In this section we look at several architectural disciplines in order to develop our intuition about software architecture. We look at hardware and network architecture because they have traditionally been considered sources of ideas for software architecture; we look at building architecture because it is the "classical" architectural discipline.

2.1.1 Computing Hardware Architecture

There are several different approaches to hardware architecture that are distinguished by the aspect of the hardware that is emphasized. *RISC* machines are examples of a hardware architecture that emphasizes the instruction set as the important feature. *Pipelined* machines and *multi-processor* machines are examples of hardware architectures that emphasize the configuration of architectural pieces of the hardware.

There are two interesting features of the second approach to hardware architecture that are important in our consideration of software architecture:

- there are a relatively small number of design elements; and
- scale is achieved by replication of these design elements.

This contrasts with software architecture, where there is an exceedingly large number of possible design elements. Further, scale is achieved not by replicating design elements, but by adding more distinct design elements. However, there are some similarities: we often organize and configure software architectures in ways analogous to the hardware architectures mentioned above. For example, we create multi-process software systems and use pipelined processing.

Thus, the important insight from this discussion is that there are fundamental and important differences between the two kinds of architecture. Because of these differences, it is somewhat ironic that we often present software architecture in hardware-like terms.

2.1.2 Network Architecture

Network architectures are achieved by abstracting the design elements of a network into nodes and connections, and by naming the kinds of relationships that these two elements have to each other. Thus we get *star* networks, *ring* networks, and *manhattan street* networks as examples of named network architectures.

The two interesting architectural points about network architecture are:

- there are two components — nodes and connections; and
- there are only a few topologies that are considered.

It is certainly the case that we can abstract to a similar level in software architecture — for example, processes and inteprocess communication. However, rather than a few topologies to consider, there are an exceedingly large number of possible topologies and those topologies generally go without names. Moreover, we emphasize aspects different from the topology of the nodes and connections. We consider instead such matters as the placement of processes (e.g., *distributed* architectures) or the kinds of interprocess communication (e.g., *message passing* architectures).

Thus, we do not benefit much from using networks as an analogy for software architecture, even though we can look at architectural elements from a similar level of abstraction.

2.1.3 Building Architecture

The classical field of architecture provides some of the more interesting insights for software architecture. While the subject matter for the two is quite different, there are a number of interesting architectural points in building architecture that are suggestive for software architecture:

- multiple views;
- architectural styles;
- style and engineering; and
- style and materials.

A building architect works with the customer by means of a number of different views in which some particular aspect of the building is emphasized. For example, there are elevations and floor plans that give the exterior views and "top-down" views, respectively. The elevation views may be supplemented by contextual drawings or even scale models to provide the customer with the look of the building in its context. For the builder, the architect provides the same floor plans plus additional structural views that provide an immense amount of detail about various explicit design considerations such as electrical wiring, plumbing, heating, and air-conditioning.

Analogously, the software architect needs a number of different views of the software architecture for the various uses and users. At present we make do with only one view: the implementation. In a real sense, the implementation is like a builders detailed view — that is, like a building with no skin in which all the details are visible. It is very difficult to abstract the design and architecture of the system from all the details. (Consider the Pompidou Center in Paris as an example.)

The notion of architectural style is particularly useful from both descriptive and prescriptive points of view. Descriptively, architectural style defines a particular codification of design elements and formal arrangements. Prescriptively, style limits the kinds of design elements and their formal arrangements. That is, an architectural style constrains both the design elements and the formal relationships among the design elements. Analogously, we shall find this a most useful concept in software architecture.

Of extreme importance is the relationship between engineering principles and architectural style (and, of course, architecture itself). For example, one does not get the light, airy feel of the perpendicular style as exemplified in the chapel at King's College, Cambridge, from romanesque engineering. Different engineering principles are needed to move from the massiveness of the romanesque to lightness of the perpendicular. It is not just a matter of aesthetics. This relationship between engineering principles and software architecture is also of fundamental importance.

Finally, the relationship between architectural style and materials is of critical importance. The materials have certain properties that are exploited in providing a particular style. One may combine structural with aesthetic uses of materials, such as that found in the post and beam construction of tudor-style houses. However, one does not build a skyscraper with wooden posts and beams. The material aspects of the design elements provide both aesthetic and engineering bases for an architecture. Again, this relationship is of critical importance in software architecture.

Thus, we find in building architecture some fundamental insights about software architecture: multiple views are needed to emphasize and to understand different aspects of the architecture; styles are a cogent and important form of codification that can be used both descriptively and prescriptively; and, engineering principles and material properties are of fundamental importance in the development and support of a particular architecture and architectural style.

2.2 The Context of Architecture

Before discussing our motivation for software architecture specifications, we posit a characterization of architecture in the context of the entire software product. Note that we do not mean to imply anything about the particular process by which this product is created — though of course there may be implications about the process that are inherent in our view. Our purpose is primarily to provide a context for architecture in what would be considered a fairly standard software product.

We characterize the different parts of the software product by the kinds of things that are important for that part — the kinds of entities, their properties and relationships that are important at that level, and the kinds of decision and evaluation criteria that are relevant at that level:

- *requirements* are concerned with the determination of the information, processing, and the characteristics of that information and processing needed by the user of the system;[2]
- *architecture* is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design;
- *design* is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements; and
- *implementation* is concerned with the representations of the algorithms and data types that satisfy the design, architecture, and requirements.

The different parts of a particular product are by no means as simple as this characterization. There is a continuum of possible choices of models, abstractions, transformations, and representations. We simplify this continuum into four discrete parts primarily to provide an intuition about how architecture is related to the requirements and design of a software system.

It should be noted that there are some development paradigms to which our characterization will not apply — for example, the exploratory programming paradigm often found in AI research. However, our characterization represents a wide variety of development and evolutionary paradigms used in the creation of production software, and delineates an important, and hitherto underconsidered, part of a unified software product [15].

2.3 Motivation for Architectural Specifications

There are a number of factors that contribute to the high cost of software. Two factors that are important to software architecture are evolution and customization. Systems evolve and are adapted to new uses, just as buildings change over time and are adapted to new uses. One frequently accompanying property of evolution is an increasing brittleness of the system — that is, an increasing resistance to change, or at least to changing gracefully [5]. This is due in part to two architectural problems: *architectural erosion* and *architectural drift*. Architectural erosion is due to violations of the architecture. These violations often lead to an increase in problems in the system and contribute to the increasing brittleness of a system — for example, removing load-bearing walls often leads to disastrous results. Architectural drift is due to insensitivity about the architecture. This insensitivity leads more to inadaptability than to disasters and results in a lack of coherence and clarity of form, which in turn makes it much easier to violate the architecture that has now become more obscured.

Customization is an important factor in software architecture, not because of problems that it causes, but because of the lack of architectural maturity that it indicates. In building software systems, we are still at the stage of recreating every design element for each new architecture. We have not yet arrived at the stage where we have a standard set of architectural styles with their accompanying design elements and formal arrangements. Each system is, in essence, a new architecture, a new architectural style. The presense of ubiquitous customization indicates that there is a general need for codification — that is, there is a need for architectural templates for various architectural styles. For the standard parts of a system in a particular style, the architect can select from a set of well-known and understood elements and use them in ways appropriate to the desired architecture. This use of standard templates for architectural elements then frees the architect to concentrate on those elements where customization is crucial.

Given our characterization of architecture and motivating problems, there are a number of things that we want to be able to do with an architectural specification:

- *Prescribe the architectural constraints to the desired level* — that is, indicate the desired restrictiveness or permissiveness, determine the desired level of generality or particularity, define what is necessity and what is luxury, and pin-point the degree of relativeness and absoluteness. We want a means of supporting a "principle of least constraint" to be able to to express only those constraints in the architecture that are necessary at the architectural level of the system description. This is an impor-

[2] Note that the notion of requirements presented here is an idealistic one. In practice, requirements are not so "pure"; they often contain constraints on the architecture of a system, constraints on the system design, and even constraints on the implementation.

tant departure from current practice that, instead of specifying the constraints, supplies specific solutions that embody those desired constraints.

- *Separate aesthetics from engineering* — that is, indicate what is "load-bearing" from what is "decoration". This separation enables us to avoid the kinds of changes that result in architectural erosion.

- *Express different aspects of the architecture in an appropriate manner* — that is, describe different parts of the architecture in an appropriate view.

- *Perform dependency and consistency analysis* — that is, determine the interdependencies between architecture, requirements and design; determine interdependencies between various parts of the architecture; and determine the consistency, or lack of it, between architectural styles, between styles and architecture, and between architectural elements.

3 Model of Software Architecture

In Section 2 we use the field of building architecture to provide a number of insights into what software architecture might be. The concept of building architecture that we appeal to is that of the standard definition: *"The art or science of building: especially designing and building habital structures"* [11]. Perhaps more relevant to our needs here is a secondary definition: *"A unifying or coherent form or structure"* [11]. It is this sense of architecture — providing a unifying or coherent form or structure — that infuses our model of software architecture.

We first present our model of software architecture, introduce the notion of software architectural style, and discuss the interdependence of processing, data, and connector views.

3.1 The Model

By analogy to building architecture, we propose the following model of software architecture:

$$\text{Software Architecture} = \{\text{ Elements, Form, Rationale }\}$$

That is, a software architecture is a set of architectural (or, if you will, design) elements that have a particular form.

We distinguish three different classes of architectural *elements*:

- processing elements;
- data elements; and
- connecting elements.

The processing elements are those components that supply the transformation on the data elements; the data elements are those that contain the information that is used and transformed; the connecting elements (which at times may be either processing or data elements, or both) are the *glue* that holds the different pieces of the architecture together. For example, procedure calls, shared data, and messages are different examples of connecting elements that serve to "glue" architectural elements together.

Consider the example of water polo as a metaphor for the different classes of elements: the swimmers are the processing elements, the ball is the data element, and water is the primary connecting element (the "glue"). Consider further the similarities of water polo, polo, and soccer. They all have a similar "architecture" but differ in the "glue" — that is, they have similar elements, shapes and forms, but differ mainly in the context in which they are played and in the way that the elements are connected together. We shall see below that these connecting elements play a fundamental part in distinguishing one architecture from another and may have an important effect on the characteristics of a particular architecture or architectural style.

The architectural *form* consists of weighted properties and relationships. The *weighting* indicates one of two things: either the importance of the property or the relationship, or the necessity of selecting among alternatives, some of which may be preferred over others. The use of weighting to indicate importance enables the architect to distinguish between "load-bearing" and "decorative" formal aspects; the use of weighting to indicate alternatives enables the architect to constrain the choice while giving a degree of latitude to the designers who must satisfy and implement the architecture.

Properties are used to constrain the choice of architectural elements — that is, the properties are used to define constraints on the elements to the degree desired by the architect. Properties define the minimum desired constraints unless otherwise stated — that is, the default on constraints defined by properties is: "what is not constrained by the architect may take any form desired by the designer or implementer".

Relationships are used to constrain the "placement" of architectural elements — that is, they constrain how the different elements may interact and how they are organized with respect to each other in the architecture.

As with properties, relationships define the minimum desired constraints unless otherwise stated.

An underlying, but integral, part of an architecture is the *rationale* for the various choices made in defining an architecture. The rationale captures the motivation for the choice of architectural style, the choice of elements, and the form. In building architecture, the rationale explicates the underlying philosophical aesthetics that motivate the architect. In software architecture, the rationale instead explicates the satisfaction of the system constraints. These constraints are determined by considerations ranging from basic functional aspects to various non-functional aspects such as economics [4], performance [2] and reliability [13].

3.2 Architectural Style

If architecture is a formal arrangement of architectural elements, then architectural style is that which abstracts elements and formal aspects from various specific architectures. An architectural style is less constrained and less complete than a specific architecture. For example, we might talk about a *distributed style* or a *multi-process style*. In these cases, we concentrate on only certain aspects of a specific architecture: relationships between processing elements and hardware processors, and constraints on the elements, respectively.

Given this definition of architecture and architectural style, there is no hard dividing line between where architectural style leaves off and architecture begins. We have a continuum in which one person's architecture may be another's architectural style. Whether it is an architecture or a style depends in some sense on the use. For example, we propose in Section 2.3 that architectural styles be used as constraints on an architecture. Given that we want the architectural specification to be constrained only to the level desired by the architect, it could easily happen that one person's architecture might well be less constrained than another's architectural style.

The important thing about an architectural style is that it encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. The useful thing about style is that we can use it both to constrain the architecture and to coordinate cooperating architects. Moreover, style embodies those decisions that suffer erosion and drift. An emphasis on style as a constraint on the architecture provides a visibility to certain aspects of the architecture so that violations of those aspects and insensitivity to them will be more obvious.

3.3 Process/Data/Connector Interdependence

As mentioned above, an important insight from building architecture is that of multiple views. Three important views in software architecture are those of processing, data, and connections. We observe that if a process view[3] of an architecture is provided, the resulting emphasis is on the data flow though the processing elements and on some aspects of the connections between the processing elements with respect to the data elements. Conversely, if a data view of an architecture is provided, the resulting emphasis is on the processing flow, but with less an emphasis on the connecting elements than we have in the process view. While the current common wisdom seems to put the emphasis on object-oriented (that is, data-oriented) approaches, we believe that all three views are necessary and useful at the architectural level.

We argue informally, in the following way, that there is a process and data interdependence:

- there are some properties that distinguish one state of the data from another; and
- those properties are the result of some transformation produced by some processing element.

These two views are thus intertwined — each dependent on the other for at least some of the important characteristics of both data and processing. (For a more general discussion of process and data interdependence, see [10].)

The interdependence of processing and data upon the connections is more obvious: the connecting elements are the mechanisms for moving data around from processor to processor. Because of this activity upon the data, the connecting elements will necessarily have some of the properties required by the data elements in precisely the same way that processing elements have some of the properties required by the data elements.

At the architectural level, we need all three views and the ability to move freely and easily among them. Our example in the next section provides illustrations of this interdependence and how we might provide three different, but overlapping, views.

[3] We use the dichotomy of *process* and *data* instead of *function* and *object* because these terms seem to be more neutral. The latter terms seem to suggest something more specific in terms of programming than the former.

4 Examples

One of the few software architectural styles that has achieved widespread acceptance is that of the multi-phase compiler. It is practically the only style in which every software engineer is expected to have been trained. We rely on this familiarity to illustrate some of the insights that we have gained into software architectures and their descriptions.

In this section we look at two compiler architectures of the multi-phase style:

- a compiler that is organized sequentially; and
- a compiler that is organized as a set of parallel processes connected by means of a shared internal representation.

Because of space limitations and for presentation purposes, our examples are somewhat simplified and idealized, with many details ignored. Moreover, we use existing notations because they are convenient for illustrative purposes; proposals for specific architectural notations are beyond the scope of this paper. In each case we focus on the architectural considerations that seem to be the most interesting to derive from that particular example. (Of course, other examples of multi-phase compiler architectures exist and we make no claims of exhaustive coverage of this architectural landscape.) Before exploring these examples, we provide a brief review of their common architectural style.

4.1 The Multi-phase Architectural Style

Our simplified model of a compiler distinguishes five phases: lexical analysis, syntactic analysis, semantic analysis, optimization, and code generation. Lexical analysis acts on *characters* in a source text to produce *tokens* for syntactic analysis. Syntactic analysis produces *phrases* that are either definition phrases or use phrases. Semantic analysis correlates use phrases with definition phrases — i.e., each use of a program element such as an identifier is associated with the definition for that element, resulting in *correlated phrases*. Optimization produces *annotations* on correlated phrases that are hints used during generation of *object code*. The optimization phase is considered a preferred, but not necessary, aspect of this style. Thus, the multi-phase style recognizes the following architectural elements:

processing elements: lexer, parser, semantor, optimizer, and code generator; and

data elements: characters, tokens, phrases, correlated phrases, annotated phrases, and object code.

Notice that we have not specified connecting elements. It is simply the case that this style does not dictate what connecting elements are to be used. Of course, the choice of connecting elements has a profound impact on the resulting architecture, as shown below.

The form of the architectural style is expressed by weighted properties and relationships among the architectural elements. For example, the optimizer and annotated phrases must be found together, but they are both only preferred elements, not necessary. As another example, there are linear relationships between the characters constituting the text of the program, the tokens produced by the lexer, and the phrases produced by the parser. In particular, tokens consist of a sequence of characters, and phrases consist of a sequence of tokens. However, there exists a non-linear relationship between phrases and correlated phrases. These relationships are depicted in Figure 1. As a final example, each of the processing elements has a set of properties that defines the constraints on those elements. The lexer, for instance, takes as input the characters that represent the program text and produces as output a sequence of tokens. Moreover, there is an ordering correspondence between the tokens and the characters that must be preserved by the lexer. A good architectural description would capture these and other such properties and relationships.

Let us illustrate this by formally describing the relationship between characters and tokens and describing the order-preserving property of the lexer. We begin the description with a data view stated in terms of sequences and disjoint subsequences.

> Let $C = \{c_1, c_2, \ldots, c_m\}$ be a sequence of characters representing a source text, C_j^i $i \leq j$ be a subsequence of C whose elements are all the elements in C between c_i and c_j inclusive, $T = \{t_1, t_2, \ldots, t_n\}$ be a sequence of tokens, and "$\cong$" indicate the correspondence between a token in T and a subsequence of C. T is said to *preserve* C if there exists an i, j, k, q, r, and s such that $1 \leq i < j \leq m$, $1 < k < n$, $1 < q \leq r < m$, and for all $t \in T$ there exists a C_y^x such that:

$$t \cong \begin{cases} C_i^1 & \text{if } t = t_1 \\ C_m^j & \text{if } t = t_n \\ C_r^q & \text{if } t = t_k, \text{ where } \exists\ u,\, v \left| \begin{array}{l} 1 \leq u \leq q-1 \\ r+1 \leq v \leq m \\ t_{k-1} \cong C_{q-1}^u \\ t_{k+1} \cong C_v^{r+1} \end{array} \right. \end{cases}$$

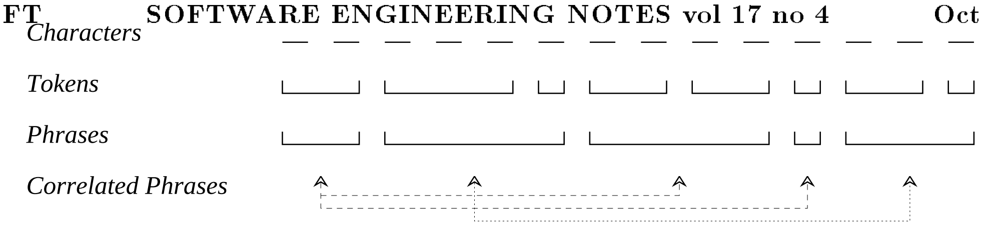

Figure 1: Data Element Relationships.

The lexer is now constrained from a processing perspective to accept a sequence of characters C, produce a sequence of tokens T, and to preserve the ordering correspondence between characters and tokens:

lexer: $C \rightarrow T$, where T *preserves* C

Finally, it is interesting to note that the connector view reveals additional constraints that should be placed on the architectural style. These constraints are illustrated by the connection between the lexer and the parser. In particular, connecting elements must ensure that the tokens produced by the lexer are preserved for the parser, such that the order remains intact and that there are no losses, duplicates, or spurious additions.

4.2 Sequential Architecture

If there is a "classical" multi-phase compiler architecture, then it is the sequential one, in which each phase performs its function to completion before the next phase begins and in which data elements are passed directly from one processing element to the other. Thus, we add the following architectural elements to those characterizing the overall style:

connecting elements: procedure call and parameters.

Furthermore, we refine tokens to include the structuring of the identifier tokens into a *name table* (NT), and refine phrases to be organized into an *abstract syntax tree* (AST). Correlation of phrases results in an *abstract syntax graph* (ASG) and optimization in an *annotated abstract syntax graph* (AASG). Figure 2 gives a processing view of the sequential architecture, showing the flow of data through the system. Notice that there are two paths from the semantor to the code generator, only one of which passes through the optimizer. This reflects the fact that a separate optimization phase is not necessary in this architecture. That is, a design satisfying this architecture need not provide an optimizer.

To illustrate the interdependence of processing and data views, let us consider the data as a whole being

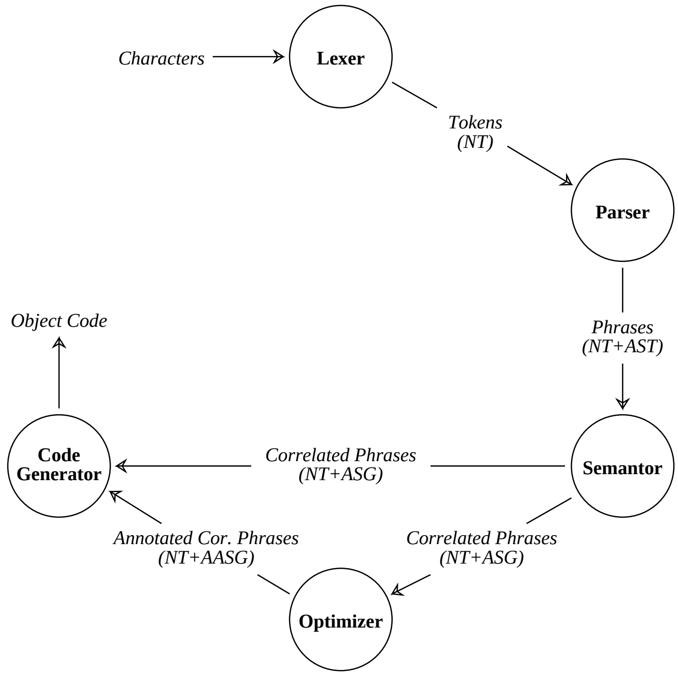

Figure 2: Processing View of Sequential Compiler Architecture.

created and transformed as they flow through the system. We have found that the data view is best captured by a notion that we call *application-oriented properties*. Application-oriented properties describe the states of a data structure that are of significance to the processing elements manipulating that structure. They can be used for such things as controlling the order of processing, helping to define the effects of a processing element on a data structure, and even helping to define the operations needed by the processing elements to achieve those effects.

For this example, the (simplified) application-oriented properties are as follows:

has-all-tokens: a state produced as a result of lexically analyzing the program text, necessary for the parser to begin processing;

has-all-phrases: a state produced by the parser, necessary for the semantor to begin processing;

has-all-correlated-phrases: a state produced by the semantor, necessary for the optimizer and code generator to begin processing; and

has-all-optimization-annotations: a state produced by the optimizer, preferred for the code generator to begin processing.

Notice again that the last property is only preferred. While in this example the application-oriented properties may appear obvious and almost trivial, in the next example they are crucial to the description of the architecture and in guaranteeing the compliance of designs and implementations with that architecture.

An interesting question to consider is why we evidently chose to use a property-based scheme for describing architectural elements rather than a type-based scheme. The reason is that type models, as they currently exist, are essentially only able to characterize elements and element types in terms of the relationship of one element type to another (e.g., subtyping and inheritance [12]), in terms of the relationships that particular elements have with other elements (e.g., as in Oros [18]), and in terms of the operations that can be performed on the elements. They are not suited to descriptions of characteristics of elements such as the application-oriented properties mentioned above. For example, simply knowing that there is an operation associated with abstract syntax graphs to connect one phrase to another does not lead to an understanding that the abstract syntax graph must have all phrases correlated before the code generator can access the graph.

Property-based schemes, on the other hand, can be used to capture easily all these characteristics; one property of an element could be the set of operations with which it is associated. It seems reasonable to consider enhancing type models in this regard and we see this as a potentially interesting area of future work. We note, however, that type-based schemes are already appropriately used at the design level, as mentioned in Section 2. Further, we note that application-oriented properties provide a good vehicle with which to drive the design, or justify the suitability, of a set of operations for an element type.

Returning to the interdependence between the processing and data views, we can see that the data

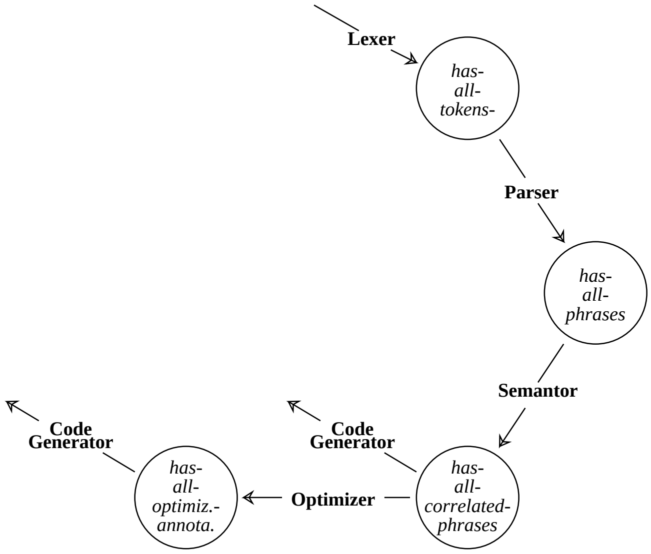

Figure 3: Data View of Sequential Compiler Architecture.

view concentrates on the particular application-oriented properties that are of importance in describing each data structure, while the processing view concentrates on the functional properties of each processing element. These views are actually complementary. In fact, if we depict the data view, as is done in Figure 3, and compare it to the processing view, shown in Figure 2, then the correspondence becomes fairly obvious.

The important architectural considerations that derive from this example can be summarized as follows:

- the form descriptions must include the relationships and constraints among the elements, including relative weightings and preferences;
- current type-based schemes for characterizing elements are insufficient; and
- there is a natural interdependence between the processing and data views that can provide complementary descriptions of an architecture.

4.3 Parallel Process, Shared Data Structure Architecture

Suppose that performance is of paramount importance, such that one wants to optimize the speed of the compiler as much as possible. One possible solution is to adopt an architecture that treats the processing elements as independent processes driven by a shared internal representation — that is, the connecting element is the shared representation and each processing element performs eager evaluation. Figure 4 depicts a

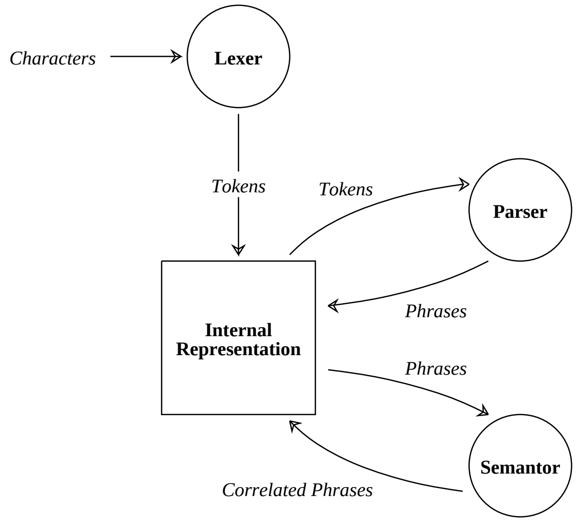

Figure 4: Partial Process View of Parallel Process, Shared Data Structure Compiler Architecture.

simplified and partial process view of this architecture, showing the relationships between the internal representation and the lexer, the parser, and the semantor. (We only consider these three processing elements in the remainder of this example.)

The application-oriented properties of the shared internal representation in this architecture are much more complicated, and interesting, than those given in the previous example. In particular, a number of processing elements are affecting the state of the internal representation, and doing so concurrently. This implies that the application-oriented properties must provide for coordination and synchronization among the processing elements. We begin by giving the basic properties that the internal representation may exhibit:

no-tokens
has-token
will-be-no-more-tokens
no-phrases
has-phrase
will-be-no-more-phrases
no-correlated-phrases
have-correlated-phrases
all-phrases-correlated

Notice that these properties imply that tokens and phrases are *consumed*, but that correlated phrases are *accumulated* (consider "has-phrase" versus "have-correlated-phrases").

Because of the parallel behavior of the processing elements, the interrelationships among the various basic properties must be explicitly described. A number of notations exist that are suitable for making such descriptions, including *parallel path expressions* [6], *constrained expressions* [1], and *petri nets* [16]. In this example we use parallel path expressions, where a comma indicates sequence, a plus sign indicates one or more repetitions, an asterisk indicates zero or more repetitions, and subexpressions are enclosed in parentheses. Synchronization points occur where names of application-oriented properties are the same in different parallel path expressions. First, the path expressions for each of the data elements — tokens, phrases, and correlated phrases — are given:

(1) (no-tokens, has-token^{+})*,
will-be-no-more-tokens, has-token*, no-tokens
(2) (no-phrases, has-phrase^{+})*,
will-be-no-more-phrases, has-phrase*, no-phrases
(3) no-correlated-phrases, (have-correlated-phrases)*,
all-phrases-correlated

Next, the path expressions relating the application-oriented properties are given:

(4) will-be-no-more-tokens, will-be-no-more-phrases,
all-phrases-correlated
(5) has-token^{+}, has-phrase
(6) has-phrase^{+}, has-correlated-phrase

Thus, tokens are consumed to produce phrases, and phrases are correlated until they are all processed.

What we have given so far is essentially a connector view (and, in this case, effectively a data view as well). Concentrating instead on the processing view allows us to describe how each processing element transforms the internal representation as well as how those processing elements are synchronized:

lexer: *(no-tokens, has-token^{+})*,*
will-be-no-more-tokens

parser: *no-phrases, (has-token^{+}, has-phrase)*,*
will-be-no-more-tokens, (has-token^{+},
has-phrase), no-tokens,*
will-be-no-more-phrases

semantor: *no-correlated-phrases, (has-phrase^{+},*
has-correlated-phrase),*
will-be-no-more-phrases, (has-phrase^{+},
has-correlated-phrase), no-phrases,*
all-phrases-correlated

An interesting question to ask is how this architecture relates to the previous one. In fact, the ability to relate similar architectures is an important aspect of the software process; an example is the evaluation of "competing" architectures. Certainly, the architectures both being of a common style captures some portion of the relationship. More can be said, however, given the use of application-oriented properties. In particular, we can draw correlations among the properties of the different architectures. The table below shows some of these correlations.

Sequential Architecture	**Parallel Architecture**
has-all-tokens	*will-be-no-more-tokens*
has-all-phrases	*will-be-no-more-pharses*
has-all-correlated-phrases	*all-phrases-correlated*

In this case, the correlations indicate common points of processing, leading, for instance, to a better understanding of the possible reusability of the processing elements.

The important points of this example can be summarized as follows:

- the processing elements are much the same as in the previous architecture, but with different "locus of control" properties;
- the form of this architecture is more complex than that of the previous one — there are more application-oriented properties and those properties require a richer notation to express them and their interrelationships;
- we still benefit from the processing/data/connector view interdependence, albeit with more complexity; and
- application-oriented properties are useful in relating similar architectures.

5 Some Benefits Derived from Software Architecture

We have previously mentioned the use of software architecture in the context of requirements and design. Software architecture provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system. There are two further benefits that we wish to discuss in detail: the kinds of analysis that software architecture specifications will enable us to perform and the kinds of reuse that we gain from our approach to software architecture.

5.1 Software Architecture and Analysis

Aside from providing clear and precise documentation, the primary purpose of specifications is to provide automated analysis of the documents and to expose various kinds of problems that would otherwise go undetected. There are two primary categories of analysis that we wish to perform: consistency and dependency analysis. Consistency occurs in several dimensions: consistency within the architecture and architectural styles, consistency of the architecture with the requirements, and consistency of the design with the architecture. In the same way that Inscape [14] formally and automatically manages the interdependencies between interface specifications and implementations, we also want to be able to manage the interdependencies between requirements, architecture, and design.

Therefore, we want to provide and support at least the following kinds of analysis:

- consistency of architectural style constraints;
- satisfaction of architectural styles by an architecture;
- consistency of architectural constraints;
- satisfaction of the architecture by the design;
- establishment of dependencies between architecture and design, and architecture and requirements; and
- determination of the implications of changes in architecture or architectural style on design and requirements, and vice versa.

5.2 Architecture and the Problems of Use and Reuse

An important aspect in improving the productivity of the designers and the programmers is that of being able to build on the efforts of others — that is, using and reusing components whether they come as part of another system or as parts from standard components catalogs.

There has been much attention given to the problem of finding components to reuse. Finding components may be important in reducing the duplication of effort and code within a system, but it is not the primary issue in attaining effective use of standardized components. For example, finding the components in a math library is not an issue. The primary issue is understanding the concepts embodied in the library. If they are understood, then there is usually no problem finding the appropriate component in the library to use. If they are not understood, then browsing will help only in conjunction with the acquisition of the appropriate concepts.

The primary focus in architecture is the identification of important properties and relationships — constraints on the kinds of components that are necessary for the architecture, design, and implementation of a system. Given this as the basis for use and reuse, the architect, designer, or implementer may be able to select the appropriate architectural element, design element, or implemented code to satisfy the specified constraints at the appropriate level.

Moreover, the various parts of previously implemented software may be teased apart to select that which is useful from that which is not. For example, the design of a component from another system may have just the right architectural constraints to satisfy a particular architectural element, but the implementation is constrained such that it conflicts with other system constraints. The solution is to use the design but not the implementation. This becomes possible only by indentifying the architectural, design, and implementation constraints and use them to satisfy the desired goals of the architecture, design, and implementation.

The important lesson in reusing components is that the possibilities for reuse are the greatest where specifications for the components are constrained the least — at the architectural level. Component reuse at the implementation level is usually too late because the implementation elements often embody too many constraints. Moreover, consideration of reuse at the architectural level may lead development down a different, equally valid solution path, but one that results in greater reuse.

6 Conclusions

Our efforts over the past few years have been directed toward improving the software process associated with large, complex software systems. We have come to believe that software architecture can play a vital role in this process, but that it has been both underutilized and underdeveloped. We have begun to address this situation by establishing an intuition about and context for software architecture and architectural style. We have formulated a model of software architecture that emphasizes the architectural elements of data, processing, and connection, highlights their relationships and properties, and captures constraints on their realization or satisfaction. Moreover, we have begun to delineate the necessary features of architectural description techniques and their supporting infrastructure. In so doing, we have set a direction for future research that should establish the primacy of software architecture.

Others have begun to look at software architecture. Three that are most relevant are Schwanke, et al., Zachman, and Shaw.

Schwanke, et al., [20] define architecture as the permitted or allowed set of connections among components. We agree that that aspect of architecture is important, but feel that there is much more to architecture than simply components and connections, as we demonstrate in this paper.

Zachman [23] uses the metaphor of building architecture to advantage in constructing an architecture for information systems. He exploits the notion of different architectural documents to provide a vision of what the various documents ought to be in the building of an information system. The architect is the mediator between the user and the builders of the system. The various documents provide the various views of different parts of the product — the users view, the contractors views, etc. His work differs from ours in that he is proposing a specific architecture for a specific application domain while we are attempting to define the philosophical underpinnings of the discipline, to determine a notation for expressing the specification of the various architectural documents, and determine how these documents can be used in automated ways.

Shaw [21] comes the closest in approach to ours. She takes the view of a programming language designer and abstracts classes of components, methods of composition, and schemas from a wide variety of systems. These correspond somewhat to our notions of processing and data elements, connecting elements, and architectural style, respectively. One major difference between our work and Shaw's is that she is taking a traditional type-based approach to describing architecture, while we are taking a more expressive property-based approach. Our approach appears better able to more directly capture notions of weighted properties and relationships. Shaw's approach of abstracting the various properties and relationships of existing architectures and embodying them in a small set of component and composition types appears rather restrictive. Indeed, she is seeking to provide a fixed set of useful architectural elements that one can mix and match to create an architecture. Shaw's approach is clearly an important and useful one and does much to promote the understanding of basic and important architectural concepts. Our approach, on the other hand, emphasizes the importance of the underlying properties and relationships as a more general mechanism that can be used to describe the particular types of elements and compositions in such a way that provides a latitude of choice.

In conclusion, we offer the following conjecture: perhaps the reason for such slow progress in the development and evolution of software systems is that *we have trained carpenters and contractors, but no architects.*

REFERENCES

[1] G.S. Avrunin, L.K. Dillon., J.C. Wileden, and W.E. Riddle, *Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Systems,* **IEEE Trans. on Software Engineering,** Vol. SE-12, No. 2, Feb. 1986, pp. 278–292.

[2] J.L. Bentley, **Writing Efficient Programs,** Addison-Wesley, Reading, MA, 1982.

[3] G.D. Bergland, *A Guided Tour of Program Design Methodologies,* **IEEE Computer,** Vol. 14, No. 10, Oct. 1981, pp. 13–37.

[4] B.W. Boehm, **Software Engineering Economics,** Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] F.P. Brooks, Jr., **The Mythical Man-Month,** Addison-Wesley, Reading, MA, 1972.

[6] R.H. Campbell and A.N. Habermann, *The Specification of Process Synchronization by Path Expressions,* **Lecture Notes in Computer Science,** No. 16, Apr. 1974, pp. 89–102.

[7] E.J. Chikofsky (ed.), **Software Development — Computer-aided Software Engineering,** Technology Series, IEEE Computer Society Press, 1988.

[8] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon, *SARA (System ARchitects Apprentice),* **IEEE Trans. on Software Engineering,** Vol. SE-12, No. 2, Feb. 1986, pp. 293–277.

[9] P. Freeman and A.I. Wasserman, **Tutorial on Software Design Techniques,** IEEE Computer Society Press, 1976.

[10] D. Jackson, *Composing Data and Process Descriptions in the Design of Software Systems,* **LCS Tech. Report 419,** Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[11] F.C. Mish, **Webster's Ninth New Collegiate Dictionary,** Merriam Webster, Springfield, MA, 1983.

[12] J.E.B. Moss and A.L. Wolf, *Toward Principles of Inheritance and Subtyping for Programming Languages,* **COINS Tech. Report 88–95,** COINS Dept., Univ. of Mass., Amherst, MA, Nov. 1988.

[13] J.D. Musa, **Software Reliability: Measurement, Prediction, Application,** McGraw-Hill, New York, NY, 1990.

[14] D.E. Perry, *The Inscape Environment,* **Proc. Eleventh Inter. Conf. on Software Engineering,** Pittsburgh, PA, IEEE Computer Society Press, May 1989, pp. 2–12.

[15] D.E. Perry, *Industrial Strength Software Development Environments,* **Proc. IFIP Congress '89, The 11th World Computer Congress,** San Francisco, CA, Aug. 1989.

[16] J.L. Peterson, *Petri Nets,* **ACM Computing Surveys,** Vol. 9, No. 3, Sept. 1977, pp. 223-252.

[17] W.E. Riddle and J.C. Wileden, **Tutorial on Software System Design: Description and Analysis,** Computer Society Press, 1980.

[18] W.R. Rosenblatt, J.C. Wileden, and A.L. Wolf, *OROS: Towards a Type Model for Software Development Environments,* **Proc. OOPSLA '89,** New Orleans, Louisiana, October 1989.

[19] E. Sandewall, C. Strömberg, and H. Sörensen, *Software Architecture Based on Communicating Residential Environments,* **Proc. Fifth Inter. Conf. on Software Engineering,** San Diego, CA, IEEE Computer Society Press, Mar. 1981, pp. 144–152.

[20] R.W. Schwanke, R.Z. Altucher, and M.A. Platoff, *Discovering, Visualizing and Controlling Software Structure,* **Proc. Fifth Inter. Workshop on Software Specification and Design,** Pittsburgh, PA, May 1989, appearing in **ACM SIGSOFT Notes,** Vol. 14, No. 3, May 1989, pp. 147–150.

[21] M. Shaw, **Larger Scale Systems Require Higher-Level Abstractions,** *Proc. Fifth Inter. Workshop on Software Specification and Design,* Pittsburgh, PA, May 1989, appearing in **ACM SIGSOFT Notes,** Vol. 14, No. 3, May 1989, pp. 143–146.

[22] A.Z. Spector, *Modular Architectures for Distributed and Database Systems,* **Proc. Eighth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems,** Philadelphia, PA, ACM Press, Mar. 1989, pp. 217–224.

[23] J.A. Zachman, *A Framework for Information Systems Architecture,* **IBM Systems Journal,** Vol. 26, No. 3, 1987.