Presented at Software Technology Conference, Salt Lake City, April 1995

Keywords: software architecture, reuse, design, domain

Features of Architecture Description Languages

Paul Kogut and Paul Clements

1 Introduction

1.1 Background: system architecture for system development?

The characteristic approach in mature engineering disciplines (e.g. civil and chemical engineering) is to build systems (e.g., buildings or chemical plants) from known solutions such as proven designs and existing components [Shaw90,D'Ippolito89]]. Engineers in mature disciplines also put great emphasis on proactively avoiding costly problems by evaluating the system before it is built [Liu87]. Models of the system's architecture are important tools for applying known solutions and doing early evaluation in mature engineering disciplines. In software engineering, the corollary is modelling the software architecture.

By software architecture, we mean the components into which a system is divided at the level of system organization, and the ways in which those components communicate, interact, and coordinate with each other [Garlan93] [Shaw95]. Examples of components include modules, processes or tasks, subsystems, Ada packages, or Unix filters. Examples of coordination mechanisms include procedure call (remote or otherwise), synchronization (e.g., rendezvous), data sharing, message passing, event broadcast and subscription schemes, and Unix pipes. The architectural view of a system separates the concerns of components' functionality (or computation) from the ways in which components interact to cooperatively perform the system's job (or coordination). The architecture represents the first mapping from requirements to computational components and, hence, it becomes possible, at a high level, to evaluate the feasibility of achieving the requirements with the proposed design.

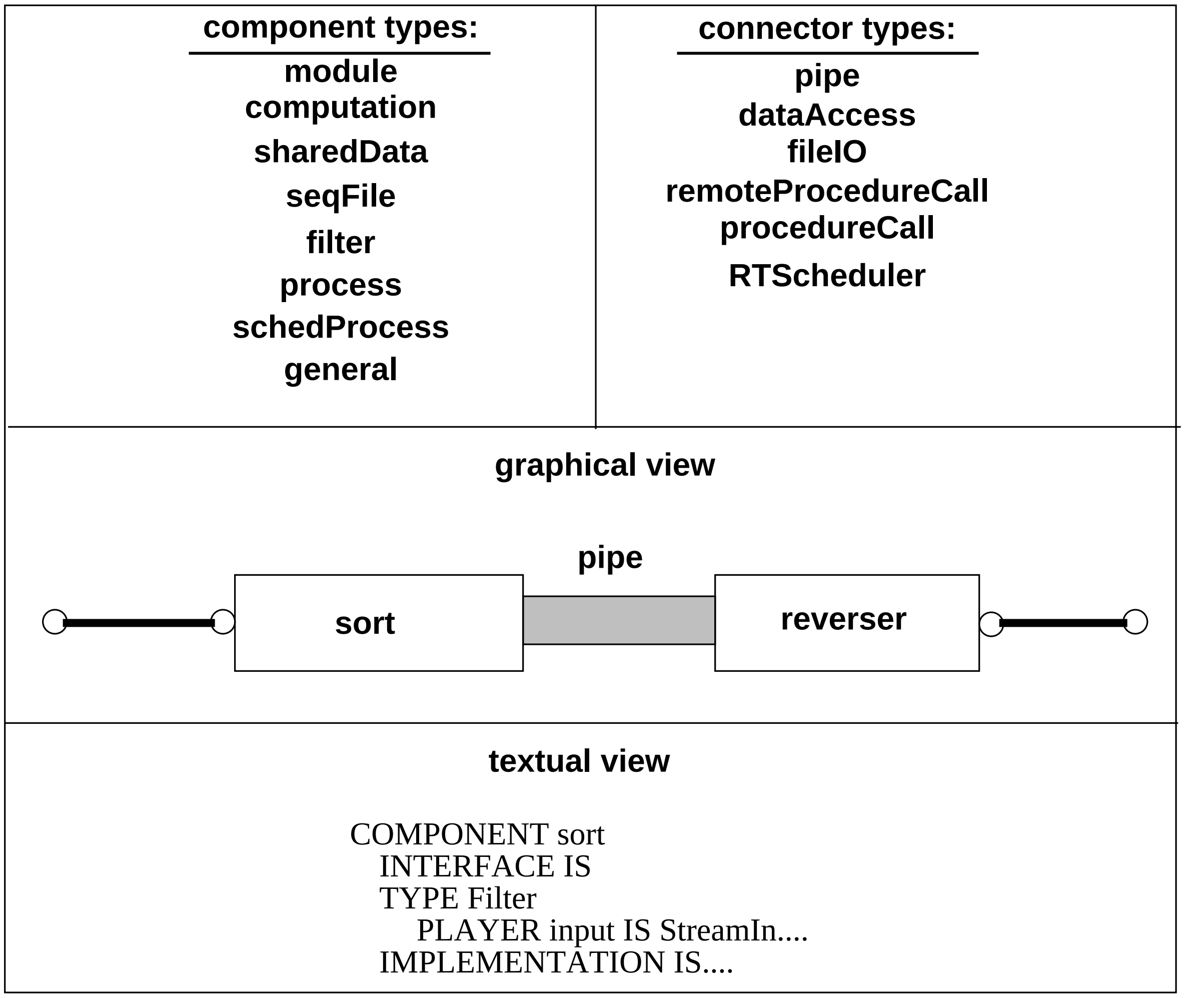
In addition, architectural decisions represent substantial resource commitments. The selection of components and connections, as well as the allocation of functionality to each component, is a codification of the earliest design decisions about a project. Thus these decisions are the hardest to change. The choice of components is institutionalized in developing organization's team structure, work assignments, management units, schedule and work breakdown structures, integration plans, test plans, and maintenance processes. Once it is made, an architectural decision is extremely difficult to retract.

In addition to its organizational implications, an architecture can either permit or preclude the achievement of most of a system's targeted quality attributes. Modifiability, for example, depends extensively on the system's modularization, which in turn, reflects the encapsulation strategies. Likewise reusability of components depends on how strongly coupled they are with other components in the system. In addition performance depends largely upon the volume and complexity of intercomponent communication and coordination, especially if the components are physically distributed processes.

The use of reliable, analyzable architecture models for software will help to

- capture design and component knowledge and rationale, for use and guidance throughout the development life cycle
- facilitate early analysis and simulation to support feasibility and resource allocation decisions

In software engineering, an architecture model may apply to a single system as well as to a family of systems in a domain; the latter is referred to as a *generic architecture* or a *domain specific software architecture* (DSSA). Architecture models for single systems support both development and maintenance. DSSAs support the development of various systems in a domain. A DSSA provides a context for components. This context forms a basis for judging a component's fit to the architecture and for developing of components that conform to the architecture.



```
COMPONENT sort
   INTERFACE IS
   TYPE Filter
       PLAYER input IS StreamIn....
   IMPLEMENTATION IS....
```

Figure 1: Example ADL - UniCon

1.2 Architecture description languages

Architecture description languages (ADLs) are emerging as the notation for architecture models. ADLs use graphics and text to express architectural information as shown in Figure 1. ADLs are often supported by tools for creation, modification, browsing, simulation, and analysis.

There are a variety of ADLs emerging from various industrial and academic research groups. For example, UniCon (language for Universal Connector support) is being developed at Carnegie Mellon University to explore issues of abstractions for architecture and composition of systems [Shaw94]; see Figure 1. Some ADLs are commercial products like UNAS/SALE (Universal Network Architectural Services/Software Architects Life-cycle Environment), which was developed by TRW and marketed by Rational [Krutchen94]. Other ARPA-sponsored ADLs include LILEANNA [Tracz93], Rapide [Luckham93], MetaH [Binns93], and ArTek/DADSE [Terry94].

ADLs vary widely in the architecture styles they support and that forms of analyses they permit. Like other tools, there is no one ADL that best fits all possible situations.

A variety of groups use ADLs. Each group has a different perspective of what a good ADL should be. People involved in the acquisition of software specify ADLs for procurements and evaluate architecture models (encoded in an ADL) that are included in proposals and presented at design reviews. ADLs as an important tools for designing, building, or re-engineering application systems.For domain engineers, ADLs are even more impratnt; they can be used for developing a DSSA.

1.3 Characterizing ADLS

The Comprehensive Approach for Reusable Defense Software (CARDS) Program and the Software Engineering Institute are cooperatively attempting to characterize the features of ADLs. The purpose of this effort is to

- provide guidance to government and industry organizations for choosing and tailoring existing ADLs for a particular domain or project.
- improve thearchitecture model for the CARDS Command Center Library iIdentify weak areas in existing ADLs that require further research

The goal of this research is to examine a representative sample of ADLs and answer these questions:

- What features do ADLs have?
- What features help describe the differences between ADLs?

Figure 2 shows the relation of ADLs to other, more familiar software engineering notations and tools. Parts of traditional programming languages (e.g., Ada package specifications) represent module interconnection, which is also important for ADLs. In addition, requirements specification languages share some notations with ADLs. Furthermore, Computer Aided Software Engineering (CASE) environments significantly overlap with ADLs, but there is much Computer Aided Software Engineering (CASE) functionality that is not part of architecture (e.g. test tools). After a preliminary analysis of over 10 example ADLs, we identified with 2 key characteristics:

- ADLs support the routine use of existing designs and components in new application systems.
- ADLs support the evaluation of an application system before it is built.

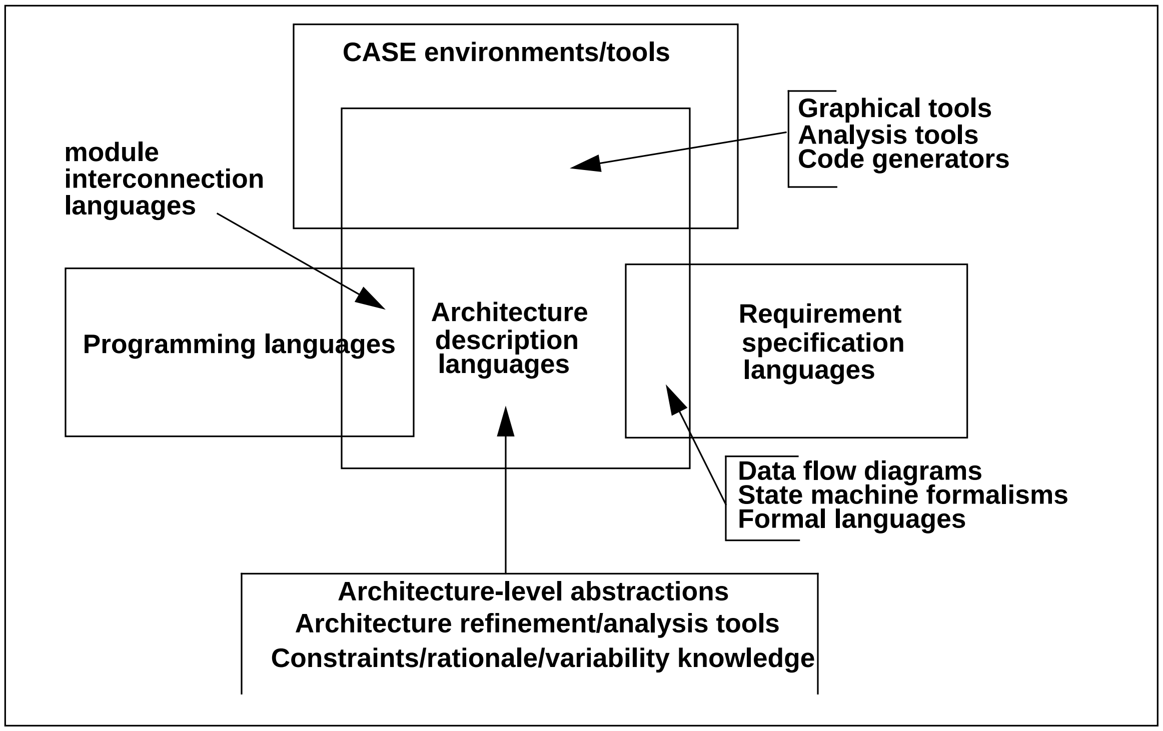

CASE environments/tools
Graphical tools
Analysis tools
Code generators
module interconnection languages
Programming languages
Architecture description languages
Requirement specification languages
Data flow diagrams
State machine formalisms
Formal languages
Architecture-level abstractions
Architecture refinement/analysis tools
Constraints/rationale/variability knowledge

Figure 2: Relation of ADLs to other notations and tools

Representations and notations used in many structured and object-oriented design methods are borderline cases of ADLs because they provide weak support for the reuse of designs (e.g. no constraints and rationale) and little support for early evaluation (e.g. rate monotonic analysis [Klein93]). However, some ADLs are based on object-oriented notations.

2 ADL Descriptive Model Framework

2.1 Organizing the framework

Since the list of ADL descriptive attributes is quite long, it was important to establish an organizing frameworkthat would serve two purposes. First, it would help searchers locate particular attribute in the framework quickly by limiting attention to relevant sections only. Second, during creation of the feature list, it would suggest inclusion of other attributes that might otherwise be forgotten.

We borrowed the high-level organizing framework of a previous language taxonomy study [Clements92]. As a result, our attributes each reside in one of three sections:

- System-riented attributes are attributes related to the application system derived from the software architecture that was encoded in the ADL. While all are attributes of the end system, they reflect the ability of the ADL to produce such a system.
- Language oriented attributes are attributes of the ADL itself, independent of the system(s) it is being used to develop. These attributes include the kind of information usually found in a language reference manual, if one exists.
- Process-oriented attributes are attributes of a process for using an ADL to create, validate, analyze, and refine an architecture description, and build an application system. Included are attributes that measure or describe how or to what extent an ADL allows predictive evaluation of the application system with respect to that attribute. They are related to the semantics of the language that support analysis, evaluation, and verification of the architecture [Vestal93]. These attributes assert that the ADL contains enough information to allow analysis of an architecture, whether or not tools actually exist that exploit the capability.
- In addition, the framework provides a place for existing tools to be described. For example, a language may allow enough timing information to be given to support schedulability analysis; a rate monotonic analysis schedule analyzer (if it exists for the language) would be an example of a tool that exploits such information.The analysis areas are drawn primarily from IEEE Std 1061, “Software Quality Metrics Methodology.” Many of these attributes are not addressed by any existing ADLs.

Some overlap between these categories is unavoidable. For example; a particular ADL might have the *linguistic* feature of expressing timing deadlines, which facilitates the development of real-time *systems* and supports analysis of schedulesas part of the development *process*. However, we have found that the overall framework is beneficial and that some of the inherent ambiguity between attribute categories actually resolved itself at lower levels.

2.2 Using the framework to characterize an ADL

The descriptive model is a framework for characterizing an individual ADL. It describes a hierarchy of attributes that define important features of an ADL. A set of ADLs characterized in this framework can then be compared.

The descriptive model is given in the form of a questionnaire to be filled out a particular ADL. Each question comes with a specific answer scale. The questionnaire provides space for the answer to be given, justification for it to be provided, plus a free-form notes section where the respondent may elaborate his or her answers. Justification and notes sections are not shown in the examples (e.g., citing specific language freatures) for space considerations.

At this point, many framework questions are highly subjective. Some questions are of the form, “Does this ADL provide a high, medium, or minimal amount of support for a particular capability.” The answer, of course, is in the eye of the beholder, and we hope that our preliminary round of ADL characterizations will provide insight into ways to make these questions less ambiguous.

2.3 System-oriented attributes

System-oriented attributes characterize an ADL by enumerating the types of systems for which they are especially applicable, or, which they were intended to support. We characterize a system by its predominant architectural style (in the sense of [Garlan93]), its broad taxonomic category such as real-time or distributed, and its application domain. Figure 3 illustrates the questions used to elicit system-oriented attributes with respect to the architecture style(s) supported by an ADL.

Architecture style: How well does the ADL allow description of architectural styles, such as those enumerated in [Garlan93]? An architectural style is a family of architectures constrained by component/ connector vocabulary, topology, and semantic constraints.

- **Pipes and filters: linked stream transformers**
- **Main program and subroutines: traditional functional decomposition**
- **Layered: structured layers with well-defined interfaces, and restrictions on cross-layer invocation**
- **Object oriented: abstract data types with inheritance**
- **Communicating processes: synchronous or asynchronous message-passing, including client-server and peer-peer**
- **Event system: implicit invocation**
- **Transactional database system: central data repository, query driven**
- **Blackboard: central shared representation, opportunistic execution**
- **Interpreter: input driven state machine**
- **Rule based**
- **Heterogeneous styles**
- **Other styles (specify which)**

Figure 3: Example of system-oriented attributes. This attribute assesses the ADL's ability to represent specific architectural styles.

2.4 Language-oriented attributes

The choice of language-oriented attributes was influenced by earlier related work [Shaw93] [Webster88]. Language-oriented attributes are divided into the following categories:

Language definition quality: How formally are the syntax and semantics of the language defined? Are there built-in (or user-defined) rules for completeness and consistency of an architecture description rendered in the language? Is the notion of consistency defined between two separate architecture descriptions? Does the language tolerate (allow useful operations on) incomplete descriptions?

Expressive power of language: Does the language offer powerful architecture-level primitives? Is it extensible, in the sense of adding new statements to the language? How does it describe architectural components and connectors? What abstractions does the language support or provide? How much nonarchitecture information (requirements, low-level design, code, test plans, etc.) can be expressed in the language? How well does the ADL support different views of the architecture (e.g., syntactic and semantic) which highlight different aspects or perspectives? (Examples of syntactic views are text and graphical diagrams. Semantic views include data flow, control flow, or state transition diagrams.) Does the language support automatic translation between views?

Readability: To what extent does the ADL support embedding comments? To what extent does the architect have control over the presentation (e.g., layout) of the architectural information?

Characteristics of intended users: Can a domain engineer use the ADL? An application engineer? A software manager? What level of knowledge is required?

Modifiability of software architecture description: Does the ADL support modularity? Are effects of change localized or highly distributed? Can descriptions rendered in the ADL be scaled up to represent large, complex systems? Does the language support multiple instantiation of components or connectors or subsystems? If so, is the instantiation static or parameterized? Can descriptions be scaled down to represent coherent subsets of the system being described? What language features exist to support these abilities?

Variability: How well does the ADL represent the variations in the application systems that can be derived from an architecture? (This attribute is illustrated in Figure 4.) The respondent would be asked to cite specific language features to justify his or her response.

Variability: How well does the ADL represent the variations in the application systems that can be derived from an architecture? Answer:

H = the variability is supported without having to change any but the replaced entities themselves
M = the variability is supported without having to change any but the replaced entities and those with which they directly interact
L = the user has to make many non-local changes to express variability

Supports structural variability: adding or deleting components, or adding or deleting connections between existing components, or replacing components or connections with new ones of a different type. (H M L)

Supports component variability: replacing a component with one that has a different functional interface or connection protocol. (H M L)

Supports component instance variability: replacing a component with one that has the same functional interface and connection protocol. (H M L)

Supports component inclusion variability: representing variations in the exclusion or inclusion of components in a single architecture description, such as explicitly marking them as optional, or allowing the specification of alternative components. (H M L)

Figure 4: Sample question eliciting an ADL’s language-oriented attribute dealing with support of

architecture variability

2.5 Process-oriented attributes

We have up to now ignored the distinction between a language and any tools or environmental infrastructure to support it. The likelihood that anyone would adopt a language but eschew all tool support for it, presumably preferring to write his own or proceed manually, is quite small. Process-oriented attributes exist to explicitly describe the reasoning ability provided by the language, and the tool support available to exploit that ability.

For each process step, an ADL is characterized by whether or not the language carries enough information in it to execute that step and, if so, whether tools exist (that are specific to the ADL, as opposed to general-purpose operating system utilities) that automate (completely or partially) that step. Respondents are asked to describe the tool and, where appropriate asked to tell whether the tool relies on simulation or analysis.

Process-oriented attributes assess whether or not the language and its associated infrastructure provide support for

Architecture creation: Is there a textual editor and graphical editor? Is importation of an architecture description allowed? If so, what native forms are supported?

Architecture validation: Is there a syntax checker (parser)? Is there a semantics checker? Is there a completeness and consistency checker?

Architecture refinement: Is there a browser or a search tool? Is there interactive support for incrementally constraining design alternatives (refinement)? Is version control supported? Can two representations be compared to see if they represent the same architecture?

Architecture analysis: Can the architecture be analyzed for time and resource economy (e.g., schedulability, throughput, memory utilization)? Can it be analyzed for functionality (e.g. completeness, correctness, security, interoperability)? Can it be analyzed for maintainability (e.g., expandability, correctability)? Can it be analyzed for portability (e.g., independence from hardware or software environments)? Can it be analyzed for reliability or usability?

Application building: What support is provided for building a compilable (or executable) software system from a specific system design?

What support is provided for building a compilable (or executable) software system from a specific system design?

System composition: the composition or integration of components for:

- **Single processor target**
- **Distributed system with homogeneous processors and operating systems**
- **Components written in more than one programming language**
- **Distributed system with more than one variety of processors and/or operating systems**

Application generation support

- **Component code generation**
- **Wrapper code generation**
- **Test case generation**
- **Documentation generation**

Figure 5: Sample question eliciting process-oriented attribute dealing with an ADL's support for application building

3 Conclusions and Issues

The descriptive model framework appears to be feasible and reasonably comprehensive based on applying it to a few ADLs so far. Significant refinement of the fine-grained attributes is expected based on descriptive modeling of more ADLs. The descriptive model appears as though it will provide an adequate basis for choosing and tailoring existing ADLs for a particular domain or project and identifying weak areas in existing ADLs that require further research.

One main issue that has been encountered is subjectivity in determining a value for an attribute. The goal is to reduce subjectivity as much as possible but retain subjective information when it is deemed useful. Many attributes allow values of high, medium, low. The meaning of these values are explained in the context of the specific attribute. Free text notes are permitted to record useful subjective information.

Fine-grained attributes that measure or describe how or to what extent an ADL allows predictive evaluation of the application system with respect to that attribute are included under the process-oriented attributes dealing with refinement and analysis. In some cases, such as "time economy/schedulability," it is clear that the value is yes when a tool such as rate monotonic analysis is supported by the ADL. In other cases, such as "reliability," it is not so clear how an ADL could explicitly support this predictive evaluation capability. This area is still an open issue although the investigation may lead to new proposed predictive evaluation capabilities for ADL tools.

Constraint, rationale, and variability information are important for software architecture.-However, the classification and description of this type of information is immature. Therefore, enhancements to the framework are likely in this area.

Another issue is the interrelationships between these attributes. For example, will more formal semantics make the ADL more applicable to a real-time systems domain? To make the descriptive model more complete and less subjective, there are plans to add important relations between attributes, as well as to apply the languages to predefined scenarios involving a variety of application areas and development activities.

Acknowledgments

The CARDS Program is sponsored by the U. S. Air Force. The Software Engineering Institute is sponsored the U. S. Department of Defense.

References

[Binns93] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. “Domain-Specific Software Architectures for Guidance, Navigation, and Control,” Honeywell Technology Center, 1993.

[Clements 92] Clements, P.,Gasarch, C., Jeffords, R. *Evaluation Criteria for Real-Time Specification Languages* Naval Research Laboratory Memorandum Report 6935 February 1992

[D’Ippolito89] D’Ippolito, R.“Using Models in Software Engineering” *Proceedings of Tri-Ada 89* ACM

[Garlan93] Garlan, David; and Shaw, Mary. *An Introduction to Software Architecture*. (CMU/SEI-93-TR-33). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, December 1993. Also in Ambriola, V.; and Tortora, G. (eds.), Advances in Software Engineering and Knowledge Engineering, Volume I. Singapore: World Scientific Publishing, 1993.

[Liu87] Liu, McGee, Epperley “Recent Developments in Chemical Process and Plant Design” Wiley 1987

[Klein93] Klein, Ralya, Pollak, Obenza, González, Harbour “Practitioner’s Handbook for Real-Time Analysis, A Guide to Rate Monotonic Analysis for Real-Time Systems” Kluwer 1993

[Krutchen94] Krutchen, Thompson “An Object-Oriented, Distributed Architecture for Large Scale Ada Systems” *Proceedings of Tri-Ada 94* ACM November 1994

[Luckham93] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann. “Specification and Analysis of System Architecture Using Rapide” Stanford technical report 1993

[Shaw90] Shaw, M. “Prospects for an Engineering Discipline of Software” *IEEE Software* Nov. 1990

[Shaw93]	Shaw, M., Garlan, D. “Characteristics of Higher-Level Languages for Software Architecture” unpublished manuscript 1993
[Shaw94]	Shaw, DeLine, Klein, Ross, Young, Zelesnik “Abstractions for Software Architectures and Tools to Support Them” unpublished report Feb. 1994
[Shaw95]	Shaw, Mary; and Garlan, David “Software Architecture: Perspectives on an Emerging Discipline” Prentice Hall. 1995
[Terry94]	Terry, Hayes-Roth, Erman, Coleman, Devito, Papanagopoulos, Hayes-Roth, “Overview of Teknowledge’s DSSA Program” ACM SIGSOFT Software Engineering Notes October 1994
[Tracz93]	Will Tracz, “LILEANNA: A Parameterized Programming Language”, Proceedings of the 2nd International Workshop on Software Reuse, March 1993
[Vestal93]	Vestal, S. *A Cursory Overview and Comparison of Four Architectural Description Languages* informal technical report Feb. 1993
[Webster88]	Webster, D. “Mapping the Design Information Representation Terrain” *IEEE Computer* December 1988

Presentation Summary

Presenters: Paul Kogut, Paul Clements

Title: Feature Analysis of Software Architecture Description Languages

Software architecture description languages (ADLs) represent a keystone technology in the paradigm shift towards domain engineering and product-line development, and away from programming and one-at-a-time development. There are many existing and emerging ADLs, and many specification languages that have ADL-like properties. A domain analysis was performed on the realm of ADLs. Features were identified that describe each ADL, the products produced by using the ADL, and the processes used to employ the ADL to develop systems. The purpose of the work is to aid a developer who may be considering using a ADL in his or her work in selecting a ADL appropriate to the architecting task at hand. An additional intent is to provide a framework that ADL designers may reference to understand which areas are important and may not be currently well supported by any existing ADL.

Paul Kogut

Paul Kogut works in Unisys Government Systems Advanced Programs at the Valley Forge Engineering Center. Kogut has worked on the Comprehensive Approach to Reusable Defense Software (CARDS) Program, which is supported by the Air Force Electronic Systems Command, since 1992. CARDS is a “virtual company” made up of several companies based in Fairmont, West Virginia and managed by Unisys. He has done domain modeling for the CARDS Command Center Library, a model-based, architecture-centric reuse library. Kogut also codeveloped a half-day CARDS tutorial on software architecture and reuse. As an SEI resident affiliate, he was a member of the Application of Software Models Project. His primary responsibility was to work on the SEI Software Architecture Technical Initiative (SATI), which draws on staff from several SEI projects. From 1983 to 1989, Paul worked for the Army Communications and Electronics Command at Fort Monmouth on a variety of software development and software engineering research projects.

Kogut has a BS in chemical engineering from Drexel University and an MS in computer science from Fairleigh Dickinson University. He is a PhD candidate in computer science at Lehigh University. His dissertation is in the area of acquiring lexical semantic knowledge for natural language processing systems.

Paul Kogut
Unisys Government Systems
70 E. Swedesford Road
Paoli, PA 19301 USA
Phone: (610) 648-2015
FAX: (610) 648-2288
kogut@vfl.paramax.com

Paul Clements

Paul Clements is head of the Software Architecture Technology Project the Software Engineering Institute. This project is investigating best practices in the representation, evaluation, analysis, and creation of software architectures. Prior to coming to the SEI, he was with the U. S. Naval Research Laboratory in Washington for 14 years where he conducted research in software engineering of real-time embedded computer systems. He received his BS and MS degrees from the University of North Carolina at Chapel Hill, and his PhD from the University of Texas at Austin. He has authored many papers in the field, perhaps the best known of which is “A Rational Design Process: How and Why to Fake It,” with David Parnas.

Paul Clements
Software Engineering Institute / Carnegie Mellon University
Pittsburgh, PA 15213 USA
Phone: (412) 268-8243
FAX: (412) 268-5758
clements@sei.cmu.edu