Eighth International Workshop on Software Specification and Design, Germany, March 1996

A Survey of Architecture Description Languages

Paul C. Clements

Software Engineering Institute

Carnegie Mellon University

Pittsburgh, PA 1521

Abstract

Architecture Description Languages (ADLs) are emerging as viable tools for formally representing the architectures of systems. While growing in number, they vary widely in terms of the abstractions they support and analysis capabilities they provide. Further, many languages not originally designed as ADLs serve reasonably well at representing and analyzing software architectures. This paper summarizes a taxonomic survey of ADLs that is in progress. The survey characterizes ADLs in terms of (a) the classes of systems they support; (b) the inherent properties of the languages themselves; and (c) the process and technology support they provide to represent, refine, analyze, and build systems from an architecture. Preliminary results allow us to draw conclusions about what constitutes an ADL, and how contemporary ADLs differ from each other.

1. Introduction

Architecture description languages (ADLs) are formal languages that can be used to represent the architecture of a software-intensive system. As architecture becomes a dominating theme in large system development and acquisition, methods for unambiguously specifying an architecture will become indispensable.

By *architecture*, we mean the components that comprise a system, the behavioral specifications for those components, and the patterns and mechanisms for interactions among them. Note that a single system is usually composed of more than one type of component: modules, tasks, functions, etc. An architecture can choose the type of component most appropriate or informative to show, or it can include multiple views of the same system, each illustrating different componentry.

To date, architectures have largely been represented by informal circle-and-line drawings in which the nature of the components, their properties, the semantics of the connections, and the behavior of the system as a whole are poorly (if at all) defined. Even though such figures often give an intuitive picture of the system's construction, they usually fail to answer such questions as:

- What are the components? Are they modules that exist only at design-time, but are compiled together before run time? Are they tasks or processes threaded together from different modules, assembled at compile-time, and form run time units? Or are they something as nebulous as "functional areas," as in data flow diagrams, or something else entirely?
- What do the components do? How do they behave? What other components do they rely on?
- What do the connections mean? Do they mean "sends data to," "sends control to," "calls," "is a part of," some combination of these, or something else? What are the mechanisms used to fulfill these relations?
- ADLs result from a linguistic approach to the formal representation of architectures, and as such they address the shortcomings of informal representations. Further, as will be shown, sophisticated ADLs allow for early analysis and feasibility testing of the design decisions[1].

ADLs trace their roots to module interconnection languages of the 1970s. ADLs today are in a maturing phase but several exist. Current examples include Rapide [12], UniCon [18], ArTek [19], Wright [1], and Meta-H [21].

This paper describes a survey of contemporary ADLs that is currently in progress. Using the techniques of domain analysis, a questionnaire was produced that characterizes an individual ADL in terms

of the systems and architectures it can support, the analysis or automated development it can facilitate or provide, and intrinsic qualities about the ADL itself. The questionnaire has been applied to over a dozen ADLs to date and the resulting data allows ADLs to be compared and contrasted. The data also provides insight into the question of when a language is an ADL as opposed to some other kind of language, such as a requirements, programming, or modelling language.

2. Architecture and ADLs

An architecture plays several roles in project development, all of them important, and all of them facilitated by a formal representation of the architecture, such as with an ADL. A formal architecture representation is more likely to be maintained and followed than an informal one, can more readily be consulted and treated as authoritative, and can more easily be transferred to other projects as a core asset. Roles include:

- Basis for communication: Project team members, managers, and customers all turn to the architecture as the basis for understanding the system, its development, and how it works during execution.
- Project blueprint: The choice of architectural components is institutionalized in the 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 has an extremely long lifetime and survives even outside of the software that it describes.
- Blueprint for product line development. An architecture may be re-used on other systems for which it is appropriate. If managed carefully, an entire product family may be produced using a single architecture. In this case, the importance of an appropriate architecture is magnified across all the projects it will serve.
- Embodiment of earliest design decisions: The architecture represents the first mapping from requirements to computational components. 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. All downstream design decisions must be consistent with the architectural choices. As such, architectural decisions are the hardest to change, and have the most far-reaching consequences.
- First approach to achieving quality attributes: An architecture can either allow or preclude the achievement of most of a system's targeted quality attributes. Modifiability, for example, depends extensively on the system's modularization, which reflects the encapsulation strategies. Reusability of components depends on how strongly coupled they are with other components in the system. Performance depends largely upon the volume and complexity of the inter-component communication and coordination, especially if the components are physically distributed processes. Thus, an architecture embodies decisions about quality priorities and tradeoffs, and represents the earliest opportunity for evaluating those decisions and tradeoffs.

Some ADLs provide an opportunity for architecture-level analysis, such as automatic simulation generation, schedulability analysis, and the like. However, even in the absence of automated analysis capabilities, other evaluative strategies can be applied to the architecture [5]. Thus, these early design decisions and quality attribute tradeoffs can be tested before they are too expensive to change.

3. ADLs and their relationship to other languages

How do ADLs differ from programing languages, requirements languages, modelling languages, and the like? Given a language for expressing properties or behaviors of a system, what are the criteria for deciding if it is an ADL or not? Unfortunately, it isn't clear.

In principle, ADLs differ from requirements languages because the latter describe problem spaces whereas the former are rooted in the solution space. In practice, requirements are often divided into behavioral chunks for ease of presentation, and languages for representing those behaviors are sometimes well-suited to representing architectural components, even though that was not the original

1. The survey draws a distinction between the language, the analysis that *can* be performed on architectural information representable by the language, and the tools that actually exist to perform such analysis. However, just as no one would use Ada without an Ada compiler, it is not likely that anyone would adopt an ADL for a project without also adopting any editing, refinement, analysis, or system-building tools that come with it. Thus, in a survey designed to help practitioners choose an ADL, we view an ADL as the language plus its supporting toolset, the *sum* of which practitioners will use as the basis for selection, usage, and after-the-fact evaluation. Nevertheless, we point out where the language captures information that could be, but perhaps has not yet been, exploited by an analysis tool.

goal of the language. For example, Modechart [10], a requirements language similar to Statechart [7], exhibited stronger analytical capabilities than any other ADL in our survey because of the presence of a model-checking verifier. Modechart was considered to be an ADL because its componentry (state machines) could be *interpreted* as architectural components. But Modechart was not designed to be an ADL, and so it is easy to produce artifacts in Modechart that do not, under any reasonable semantic interpretation, correspond to an architectural view of a system.

In principle, ADLs differ from programming languages because the latter bind all architectural abstractions to specific point solutions whereas ADLs intentionally suppress or vary such binding. In practice, architecture is embodied and recoverable from code, and many languages provide architecture-level views of the system. For example, Ada offers the ability to view a system just in terms of its package specifications, which are the interfaces to components. However, Ada offers little or no architecture-level analytical capabilities, nor does it provide architecture-level insight into how the components are "wired" together.

In principle, ADLs differ from modelling languages because the latter are more concerned with the behaviors of the whole rather than of the parts, whereas ADLs concentrate on representation of components. In practice, many modelling languages allow the representation of cooperating components and can represent architectures reasonably well.

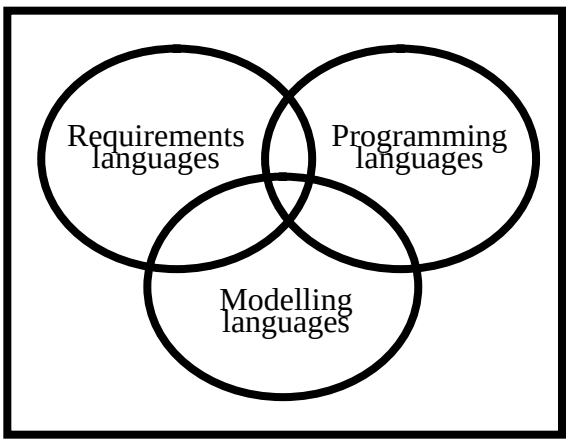

Figure 1: Requirements languages, programming languages, and modelling languages have aspects in common with ADLs.

Two leading ADL researchers offer their desiderata for ADLs. Shaw lists the following as important properties that ADLs should exhibit [18]:

- ability to represent components (primitive or composite) along with property assertions, interfaces and implementations;
- ability to represent connectors, along with protocols, property assertions, and implementations
- abstraction and encapsulation
- types and type checking
- ability to accommodate analysis tools openly

Luckham lists the following as requirements for an ADL [13]:

- component abstraction
- communication abstraction
- communication integrity (limiting communication to those components connected to each other architecturally)
- ability to model dynamic architectures
- ability to reason about causality and time
- hierarchical refinement support
- relativity, the mapping of behaviors to (possibly different) architectures, as a first step towards checking conformance.

These lists illustrate the different points of view about what constitutes an ADL. There is no clear line between ADLs and non-ADLs. Languages can, however, be discriminated from one another according to how much architectural information they represent, and our survey has attempted to capture this. Languages that were born as ADLs show a clear advantage in this area over languages built for some other purpose and later co-opted to represent architectures. In Section 6, we will re-visit this issue in light of the survey results.

4. The ADL survey

This section outlines the purpose, form, content, and methodology of the ADL survey.

4.1: Purpose

Our survey of ADLs was intended to provide information to three communities:

- architects, who must choose an ADL. Our survey is intended to highlight the capabilities and qualities of ADLs presently available. It is not a score card or even an evaluation; ADLs are not better or worse than

each other. Rather, they feature different capabilities and qualities, and the best choice is use specific.

- technology sponsors, who fund development of ADLs and ADL tools. Our survey is intended to allow them to spot and re-direct duplicative work.
- ADL creators. Our survey is intended to allow creators to identify capability areas that have, to date, been largely passed over by ADLs.

4.2: Form

Drawing from previous efforts to survey ADLs [20] as well as other kinds of specification languages [2], we crafted a *feature analysis* of ADLs. Feature analysis is a tool of certain domain analysis methods such as the Feature-Oriented Domain Analysis (FODA) method [11]; it proceeds by cataloguing user-visible system features in a structured fashion. In our survey, the domain consisted of the set of languages that might be considered ADLs; "user-visible" means apparent to a user of the language.

The feature analysis took the form of a survey questionnaire; the questionnaire is the manifestation of a framework of important features that a particular ADL may or may not have. An ADL in the survey is characterized by answering a series of questions about its capabilities, features, and usage history. A set of completed questionnaires thus provides a basis for comparing and contrasting the ADLs with each other, and ADLs with languages that would not be considered ADLs.

Features are structured into the following three categories: system-oriented features, language-oriented features, and process-oriented features.

System-oriented features

System-oriented features are related to the application system derived from the architecture description. For example, certain ADLs may not be able to express real-time constraints about a system's architectural components, while others can. All features in this category are attributes of an end system; however, they reflect on the ability of the ADL to express or describe those attributes at the architectural level.

Specific questions from the survey include[2]:

Applicability: How suitable is the ADL for representing a particular type of application system?

- **Architectural styles: How well does the ADL allow description of architectural styles, such as those enumerated in [6]? Styles include pipe and filter, blackboard, etc.**
- **System class: What broad classes of systems can have their architectures represented with the ADL? Classes include: hard real-time, soft real-time, embedded, distributed, dynamic architectures, imported component systems, other.**
- **Domains: What application domains is the ADL designed specifically to support, if any, and how and to what degree?**

Language-oriented features

Language-oriented features are features 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. An example is how formally specified the ADL's syntax and semantics are and what architectural abstractions are embodied by the ADL.

Language-oriented questions include:

Language definition quality

- **Formality: How formally are the ADL's syntax and semantics defined?**
- **Completeness: How well is completeness defined for an architecture descriptions? How does the ADL treat an incomplete architecture description?**
- **Consistency: Is self-consistency defined for an architecture description? Is it defined between two different architecture description, or between an architecture description and some other rendering of the system such as a requirements specification or coded implementation? Are the consistency rules built-in or user-defined?**

Scope of language: How much non-architecture information can the ADL represent?

Design History: How well does the ADL provide for recording architectural design information?

Views: How well does the ADL support different views that highlight different aspects/perspectives of the architecture?

- **Syntactic view list: Which syntactic views are supported? Graphical, textual, etc.**
- **Semantic view list: Which semantic views are supported? Data flow, control flow, process view, etc.**
- **Inter-view cross reference: Does the ADL provide for translating among views?**

---

2. For brevity, a great deal of clarifying elaboration has been omitted, as has the explanation of the response scales. For instance, the question, "How does the ADL treat an incomplete architecture description?" is instantiated in the actual questionnaire as several detailed questions dealing with the operations that can be performed on an incomplete description, whether built-in or user-defined completeness rules exist or prevail, whether the language features a wildcard or incompleteness token, etc. We ask the reader's indulgence to believe that the subjective-sounding questions presented in this paper are backed up in the actual questionnaire with detailed sub-questions that address each issue much more objectively.

- **Architectural content of views: Does the ADL provide views that show mostly architectural information?**

Readability

- **Embedded comments**
- **Presentation control**

Characteristics of intended users

- **Target users: domain engineer, application engineer, systems analyst, software manager?**
- **Expertise required: domain expertise, software design expertise, programming language expertise?**

Modifiability of software architecture description

- **Ease of change: How well does the ADL support ease of change of the architecture and its representation?**
- **Scalability: the degree to which the ADL can represent large and/or complex systems. Hierarchies? Cross-referencing? Subset capability? Composition? Multiple instantiation of templates?**

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

Expressive power of the ADL.

- **Powerful primitives featured**
- **Extensibility**
- **What abstractions does the ADL support or provide? Are the abstractions architectural, behavioral, or implementation based?**

Process-oriented features

Process-oriented features are features of a process related to using the ADL to create, validate, analyze, and refine an architecture description, and build an application system from it. Included are attributes that measure or describe how or to what extent an ADL allows predictive evaluation of the application system based on architecture-level information. These attributes measure whether or not the ADL contains enough information to make an architecture analyzable, independently of whether or not tools actually exist that exploit that capability. In addition, the questionnaire provides a place for existing tools to be described.

For example, an ADL may allow enough timing information to be given to support schedulability analysis. A rate monotonic analysis schedule analyzer (if it exists for the ADL) would be an example of a tool that exploits such information.

The analysis areas are primarily drawn from IEEE Std 1061, "Software Quality Metrics Methodology" [9]. Many of these attributes are not addressed by any existing ADLs.

Process-oriented questions are:

Architecture creation support: textual editor, graphical editor, import tool?

Architecture validation support: syntax checker, semantics checker, completeness checker, consistency checker?

Architecture refinement support: browser, search tool, incremental refinement tool, version control, architecture comparison?

Architecture analysis support: What support is provided by the ADL for analyzing architecture-level information in order to predict or project qualities of the end system?

- **Analyzing for time and resource economy: schedulability, throughput, other time economies, memory utilization, other resource economics?**
- **Analyzing for functionality: completeness, correctness, security, interoperability?**
- **Analyzing for maintainability: correctability, expandability, testability?**
- **Analyzing for portability: hardware independence, software independence?**
- **Analyzing for reliability: error tolerance, degraded operation capability, availability?**
- **Analyzing for usability: understandability, ease of learning, operability?**

Application building: building a compilable (or executable) software system from a specific system design.

- **System composition: the composition or integration of components' bodies, in order to produce a compilable or executable software system for: single processor target, distributed homogeneous system, distributed heterogeneous system, components written in more than one language?**
- **Application generation support: component code generation, wrapper code generation, test case generation, documentation generation?**

Tool Maturity, availability, and support.

Process support: Does a user's manual exist? Does a training course exist?

5. Gathering data

The questionnaire has been circulated to the proprietors of over a dozen ADLs to date. Table 1 contains the list of those who have responded and had their results validated to date. Other surveys in progress include those for Rapide [12], Gestalt [17], ACME [4], and FR [8].

In order to gain confidence in the questionnaire, we applied it to some languages, such as Modechart, that would not be considered mainstream ADLs. The intent was to observe how these "cusp" languages scored in order to try to understand qualities that distinguish ADLs from non-ADLs.

Table 1: ADLs surveyed (alphabetical order)

ADL	Source	Citation
ArTek	Teknowledge	[19]
CODE	Univ. of Texas at Austin	[15]
Demeter	Northeastern Univ.	[16]
Modechart	Univ. of Texas at Austin	[10]
PSDL/ CAPS	Naval Postgrad. School	[14]
Resolve	Ohio State Univ.	[3]
UniCon	Carnegie Mellon Univ.	[18]
Wright	Carnegie Mellon Univ.	[1]

The intent of the questionnaire was to make as many of the questions as possible objective, in the sense that two independent observers would be highly likely to respond the same way to the question. Where subjectivity was called for (e.g., "how well does the language support architecture representations that are easy to change?") we asked the respondent to provide specific language features and usage scenarios that justified the answer. Nevertheless, it has been necessary to conduct a validation interview for each questionnaire in order to add credibility to the responses.

6. Conclusions

This section will present the results of the survey to date and postulate some conclusions about ADLs in general.

6.1: Survey results

Table 3 distills many of the detailed questions down into summary statements about the languages' capabilities. Table 2 explains the answer symbols in Table 3. Table 3 serves as a quick-reference guide for readers interested in finding a language for a particular application. The intent is that the quick-reference guide would provide a reader with a small set of candidate languages mostly likely to suit his or her purpose. For complete details and final selection, the full surveys of the languages (not reproduced in this paper) should be consulted.

To create the quick-reference guide, the full surveys were distilled using the following heuristics:

- If it was possible to combine a set of questions into a general one about the language's capability without loss of significant detail, this was done. The rating of a language's general capability is either the arithmetic average of its ratings on the component questions (if the rating scale is ordered, such as High/Medium/Low), or the answer that appeared most often in the component questions (if the rating scale is unordered). If the component questions were Yes/No, then the summary rating reflects the ratio of Yes answers.
- Questions on which all languages scored the same or nearly the same tended to be suppressed, since they did not serve as useful discriminators among the languages.

Shading information is used in the cells to augment the textual content. The lighter the shading, the more capable the language rated on the particular issue.

6.2: ADLs versus other languages

Languages that are generally considered "mainstream" ADLs shared the following aspects:

- The abstractions they provided to the user were architectural in nature. All represented components and connections (although Rapide represents connections only by specifying communication behaviors among components).
- Most of the views provided by the ADLs contained predominantly architectural information. This is in contrast to a programming language or a requirements language that tends to show other kinds of information.
- Analysis provided by the language relies on architecture-level information. Rapide's discrete event simulator, for instance, uses behavioral information about each component to generate partially ordered event sets. UniCon's interface to the rate monotonic analyzer uses black-box performance information about each component in order to compute schedulability. This is in contrast to a performance analyzer that identifies bottlenecks based on implementation information (i.e., code).

6.3: Discriminators

The following areas uncovered interesting differences among the ADLs we surveyed:

- ADLs differed markedly in their ability to handle real-time constructs at the architectural level. Rough-

ly half claimed to deal with hard-real-time constructs such as deadlines; only a small number dealt with soft-real-time constructs such as tasking priorities.

- ADLs varied in their ability to support the specification of particular architectural styles. All ADLs could represent pipe-and-filter architectures, either directly or indirectly. Other styles did not fare as well. All provided hierarchical structuring of components, and could represent objects, but only a few could handle object-oriented class inheritance. Only two handled dynamic architectures.
- The set of surveyed ADLs split evenly on their ability to let a user define new types of components and connectors, define new statements in the ADL, and represent non-architectural information (such as requirements, or test cases) using the ADL. Extensibility and scope were good discriminators.
- While all languages provided built-in internal consistency and completeness rules for artifacts rendered in those languages, only a few allowed the user to define what was meant by consistency, and only a few dealt with consistency between different artifacts (e.g., between an architecture and component designs).
- ADLs varied widely in their ability to support analysis. Rapide features a discrete-event simulator based on partially-ordered sets of event behaviors. Modechart features a model-checking verifier that takes as input a logical assertion and reports whether the system description guarantees, prohibits, or is merely compatible with that assertion. UniCon interfaces directly to a rate monotonic schedule analyzer. A more general analysis capability (such as in the case of a verifier) showed up in the questionnaire as the ability to analyze for many different quality attributes. For instance, a simulator could be used to analyze for usability by letting users observe the simulated behavior to see if it meets their expectations.
- ADLs differed in their ability to handle variability, or different instantiations of the same architecture. All supported component variability through simple rewrite capability, but few supported maintaining different instantiations of the same architecture simultaneously.
- When offering more than one architectural view, ADLs varied widely in their ability to translate among those views. View interchangeability is a strong discriminator among the ADLs we surveyed.

6.4: Commonalities

The following commonalities were noticed:

- All of the ADLs surveyed had a graphical syntax; all but one also had a textual form. Every language but one featured a formal syntax, and the vast majority featured formally-defined semantics.
- Every language claimed to be able to model distributed systems.
- ADLs tended not to provide much support for capturing design rationale and/or history, other than through generic or general-purpose annotation mechanisms.
- All of the ADLs claimed to handle data flow and control flow as interconnection mechanisms. Modechart was the weakest in this regard, because of its ability to deal only with state predicates (such as “data sent”) instead of actual data values.
- All ADLs provided help with creation and validation an refinement of architectures, even if validation was only done in the context of the language’s own rules for completeness or legality.
- All featured the ability to represent hierarchical levels of detail, and handle multiple instantiations of a template as a quick way to perform copying substructures during creation.
- All ADLs support the application engineer, and most support the domain engineer, even if only indirectly. Most support the systems analyst by providing upfront analytical capabilities. None claimed to directly support project management.
- Finally, a glaring commonality was the lack of indepth experience and real-world application that ADLs currently offer. It is to be hoped that as the benefits of architecture become better understood, that the benefits of formal representations will be equally prized, and ADLs will come into their own as viable technologies for complex system development.

6.5: What constitutes an ADL?

After surveying a broad selection of languages, all of which have some claim to being ADLs, what can we conclude about what makes a language an ADL? The following seems to be a minimal set of requirements for a language to be an ADL:

- An ADL must support the tasks of architecture creation, refinement, and validation. It must embody rules about what constitutes a complete or consistent architecture.

- An ADL must provide the ability to represent (even if indirectly) most of the common architectural styles enumerated in [6].
- An ADL must have the ability to provide views of the system that express architectural information, but at the same time suppress implementation or non-architectural information.
- If the language can express implementation-level information, then it must contain capabilities for matching more than one implementation to the architecture-level views of the system. That is, it must support specification of families of implementations that all satisfy a common architecture.
- An ADL must support either an analytical capability, based on architecture-level information, or a capability for quickly generating prototype implementations.

6.6: Future trends

Communicating an architecture to a stakeholder becomes a matter of representing it in an unambiguous, readable form that contains the information appropriate to that stakeholder. Current trends in ADL development seems to be focusing on enhancing the analysis and system-generation capabilities of the languages. Architecture is, after all, only a means to an end, and information that developers can infer about the end system is more valuable than information about just the architecture. Being able to quickly develop a system or manage qualities of the final product are the real payoff of ADLs. While development of architecture languages is proceeding apace, there is less attention being paid to the following areas:

- **Infrastructures to support ADL development.** Most ADLs share a set of common concepts. Building tools to support an ADL involves solving a common set of problems. Development of an ADL development environment would facilitate the rapid production of ADLs and supporting tools, thus allowing good ideas to come to market faster. Garlan's Aesop/ACME work represents an early and important contribution to this area [4].
- **Integration of ADL information with other life-cycle products.** As ADLs mature, they will take a more prominent role in the litany of life-cycle products (such as detailed design documents, test cases, etc.). Encouragement should be given to early consideration of the relationship that an architecture description will bear to these other documents. For example, what test cases might be generated for a system based on a description of its components and interconnection mechanisms? What kind of and how much executable code can be automatically generated? How can traceability of architecture to requirements be established? This work could culminate in the complete integration of architecture descriptions into the development environment, giving rise to a sort of "architectorium." This can be thought of as an exploration environment in which architectures are drafted, validated via mapping to requirements, their implications explored via analysis or rapid prototyping, alternatives suggested in an expert-system-like fashion, and project infrastructures necessary for development (e.g., work schedule templates, component-based configuration control libraries, test plans, etc.) are generated.

It is to be hoped researchers will be spurred to provide ADLs that address gaps in current capability, particularly in analysis and program family support.

7. Acknowledgments

Paul Kogut of Loral is a co-creator of the ADL feature analysis and a co-author of the SEI report describing it. Many of the insights in this paper are his. Thanks are due to the many ADL creators who patiently filled out our questionnaire, and allowed us to come visit and discuss the survey results in depth. This work would not have been possible without them. Special thanks go to David Garlan, David Luckham, Bruce Weide, John Hartman, Mary Shaw, J.C. Browne, and their respective, usually overworked, ADL-building teams.

The SEI is sponsored by the U.S. Department of Defense.

Table 2: Answer key

Symbol	Meaning
Y	Yes
N	No
H	High capability: language provides explicit features to support this capability
M	Medium: language provides generic features through which this capability may be indirectly achieved
L	Low: language provides little support
T	Tool specifically developed for the ADL supports this capability
E	External tool provides this capability
P	Language provides enough information to support the capability, but no tool support currently exists.

Table 3: ADL survey results

Attributes	ARTEK	CODE	DEMETER	MODECHART	PSDL/CAPS	RESOLVE	UNICON	WRIGHT
Applicability								
Ability to represent styles	M	M	M	M	H	M	M	H
Ability to handle real-time issues	M		M	H	H	L	H	L
Ability to handle distributed system issues	H	H	H	H	H	M	M	M
Ability to handle dynamic architectures	L	H	H	L	M	L	?	L
Language definition quality								
Attention to completeness of arch. spec.	M	L	H	M	M	L	M	H
Attention to consistency of arch. spec.	L	L	M	L	M	H	L	H
Scope of language; intended users								
Requirements	H	L	?	H	H	M	L	L
Detailed design/algorithms	L	M	H	M	H	M	L	L
Code	M	L	H	L	L	H	H	L
Domain engineer	Y	Y	Y	N	Y	N	Y	Y
Application engineer	Y	Y	Y	Y	Y	Y	Y	Y
Systems analyst	Y	N	Y	Y	Y	Y	N	Y
Capturing design history	?	L	?	L	?	?	?	?
Views								
Textual	Y	N	Y	Y	Y	Y	Y	Y
Graphical	Y	Y	Y	Y	Y	N	Y	N
Semantic view richness	L	M	H	H	H	L	?	L
Inter-view cross reference	M	L	L	M	M	L	M	L
Support for Variability	M	L	H	L	H	H	L	H
Expressive power, extensibility	H	L	M	M	H	H	M	M
Support for architecture creation	T	T	P	T	T	E	T	
Support for architecture validation	T	T	E	T	T	P	T	E
Support for architecture refinement	T	P	P	T	T	P	P	P
Support for architecture analysis	E		P	T	T	P		
Support for application building		T	P		P			
Tool Maturity								
Available as COTS?	N	N	?	N	N	N	N	N
Age (years)	3	?	10	7	5	?	2	3
Number of sites in use	11	?	?	4	12	4	1	1
Customer support available	Y	Y	N	N	N	Y	N	N

8. References

1. Allen, R., Garlan, D. “Beyond Definition / Use: Architectural Interconnection,” *Proceedings, Workshop on Interface Definition Languages,* Portland Oregon, 20 January 1994l.

2. Clements, P.,Gasarch, C., Jeffords, R. “Evaluation Criteria for Real-Time Specification Languages,” Naval Research Laboratory Memorandum Report 6935, February 1992.

3. Edwards, S., Heym, W., Long, T., Sitarman, M., and Weide, B.; “Specifying Components in RESOLVE,” Software Engineering Notes, vol. 19, no. 4, October 1994.

4. Garlan, D., Allen, R., Ockerbloom, J. “Exploiting Style in Architectural Design Environments,” *Proceedings of SIGSOFT ‘94: Foundations of Software Engineering,* December 1994, ACM Press.

5. R. Kazman, L. Bass, G. Abowd, and M. Webb, “SAAM: A Method for Analyzing the Properties Software Architectures”. In *Proceedings of the 16th International Conference on Software Engineering*, (Sorrento, Italy), May 1994, pp. 81-90.

6. 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.

7. Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; Politi, M.; Sherman, R.; Shtul-Trauring, A.; “STATEMATE: a working environment for the development of complex reactive systems” Proceedings of the 10th International Conference on Software Engineering Singapore April 1988;

8. Hartman, J., Chandrasekaran, B., “Functional Representation and Understanding of Software: Technology and Application,” *Proceedings, 1995 Dual-Use Technologies and Applications Conference,* 1995.

9. IEEE Std 610.12, *IEEE Standard Glossary of Software Engineering Terminology*, September 1990.

10. Jahanian, F., and Mok, A. “Modechart: A Specification Language for Real-Time Systems,” *IEEE Transactions on Software Engineering*, vol. 20, no. 12, December 1994, pp. 933-947.

11. Kang, Kyo C.; Cohen, Sholom G.; Hess, James A.; Novak, William E.; & Peterson, A. Spencer. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU / SEI-90-TR-21, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

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

13. Luckham, David, Vera, James. “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering,* to appear.

14. Luqi, Shing, M., Barnes, P., and Hughes, G. “Prototyping Hard Real-Time Ada Systems in a Classroom Environment,” *Proceedings of the Seventh Annual Ada Software Engineering Education and Training (ASEET) Symposium*, Monterey, 12-14 January 1993.

15. Newton, P., Browne, J. “The CODE 2.0 Graphical Parallel Programming Language,” *Proceedings, ACM International Conference on Supercomputing,* July 1992.

16. Palsberg, J., Xiao, C., Lieberherr, K. “Efficient Implementation of Adaptive Software (Summary of Demeter Theory)”, Northeastern University, Boxton, 10 January 1995.

17. Schwanke, R., “Industrial Software Architecture with Gestalt,” technical report, Siemens Corporate Research, Princeton NJ.

18. Shaw, DeLine, Klein, Ross, Young, Zelesnik “Abstractions for Software Architectures and Tools to Support Them,” Carnegie Mellon University, unpublished report Feb. 1994

19. Terry, Hayes-Roth, Erman, Coleman, Devito, Papanagopoulos, Hayes-Roth “Overview of Teknowledge’s DSSA Program,” *ACM SIGSOFT Software Engineering Notes*, October 1994.

20. Vestal, S. *A Cursory Overview and Comparison of Four Architectural Description Languages*, technical report, Honeywell, Feb. 1993

21. Vestal, S. “Mode Changes in a Real-Time Architecture Description Language,” Proceedings, Proc. International Workshop on Configurable Distributed Systems: Honeywell Technology Center and the University of Maryland.