A Framework for Classifying and Comparing Architecture Description Languages

Nenad Medvidovic and Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425, U.S.A.
{neno,taylor}@ics.uci.edu

Abstract. Software architectures shift developers' focus from lines-of-code to coarser-grained architectural elements and their interconnection structure. Architecture description languages (ADLs) have been proposed as modeling notations to support architecture-based development. There is, however, little consensus in the research community on what is an ADL, what aspects of an architecture should be modeled in an ADL, and which ADL is best suited for a particular problem. Furthermore, the distinction is rarely made between ADLs on one hand and formal specification, module interconnection, simulation, and programming languages on the other. This paper attempts to provide an answer to these questions. It motivates and presents a definition and a classification framework for ADLs. The utility of the definition is demonstrated by using it to differentiate ADLs from other modeling notations. The framework is used to classify and compare several existing ADLs.[1]

Keywords: software architecture, architecture description languages, definition, classification, comparison

1 Introduction

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family [GS93, PW92]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure. To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. Architecture description languages (ADLs) and their accompanying toolsets have been proposed as the answer. Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [Ves93]. ADLs have recently become an area of intense research in the software architecture community [GPT95, Gar95, Wolf96].

A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages. Examples specifically considered in this paper are Aesop [GAO94], MetaH [Ves96], LILEANNA [Tra93], ArTek [TLPD95], C2 [MTW96, MORT96, Med96], Rapide [LKA+95, LV95], Wright [AG94a, AG94b], UniCon [SDK+95], Darwin [MDEK95, MK96], and SADL [MQR95]. Recently, initial work has been done on an architecture interchange language, ACME [GMW95, GMW97], which is intended to support mapping of archi-

[1] This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under contract number F30602-94-C-0218. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

tectural specifications from one ADL to another, and hence enable integration of support tools across ADLs.[2]

There is, however, still little consensus in the research community on what an ADL is, what aspects of an architecture should be modeled by an ADL, and what should be interchanged in an interchange language [MTW96]. For example, Rapide may be characterized as a general-purpose system description language that allows modeling of component interfaces and their externally visible behavior, while Wright formalizes the semantics of architectural connections. Furthermore, the distinction is rarely made between ADLs on one hand and formal specification, module interconnection (MIL), simulation, and programming languages (PL) on the other. Indeed, for example, Rapide can be viewed as both an ADL and a simulation language, while Clements contends that CODE [NB92], a parallel programming language, is also an ADL [Cle96a].

Several researchers have attempted to shed light on these issues, either by surveying what they consider existing ADLs [KC94, KC95, Cle96a, Ves93] or by listing "essential requirements" for an ADL [LV95, SDK+95, SG94, SG95]. Each of these attempts furthers our understanding of what an ADL is; however, for various reasons, each ultimately falls short in providing a compelling answer to the question.

This paper builds upon the results of these efforts. It is further influenced by insights obtained from studying individual ADLs, relevant elements of languages commonly not considered ADLs (e.g., PLs), and experiences and needs of an ongoing research project, C2. The paper presents a definition and a relatively concise classification framework for ADLs: an ADL must explicitly model *components*, *connectors*, and their *configurations*; furthermore, to be truly usable and useful, it must provide *tool support* for architecture-based development and evolution. These four elements of an ADL are further broken down into their constituent parts.

The remainder of the paper is organized as follows. Section 2 motivates our definition and taxonomy of ADLs. Section 3 demonstrates the utility of the definition by determining whether several existing notations are ADLs. Sections 4-7 describe the elements of components, connectors, configurations, and tool support, respectively, and assess the above ADLs based on these criteria. Conclusions round out the paper.

2 ADL Classification and Comparison Framework

Any effort such as this one must be based on discoveries and conclusions of other researchers in the field. For that reason, we closely examined ADL surveys conducted by Kogut and Clements [KC94, KC95, Cle96a] and Vestal [Ves93]. We also studied several researchers' attempts at identifying essential ADL characteristics and requirements: Luckham and Vera [LV95], Shaw and colleagues [SDK+95], Shaw and Garlan [SG94, SG95], and Tracz [Wolf97]. As a basis for architectural interchange, ACME [GMW95, GMW97] gave us key insights into what needs to remain constant across ADLs. Finally, we built upon our conclusions from an earlier attempt to shed light on the nature and needs of architecture modeling [MTW96].[3]

Individually, none of the above attempts adequately answers the question of what an ADL *is*. Instead, they reflect their authors' views on what an ADL *should have* or

[2] Although ACME is not an ADL, it contains a number of ADL-like features. We include it in the paper in order to highlight the difference between an ADL and an interchange language.

[3] Due to space constraints, details of these approaches are omitted. They are provided in [Med97].

should be able to do. However, a closer study of their various collections of features and requirements shows that there is a common theme among them, which is used as a guide in formulating this framework for ADL classification and comparison. To a large degree, this taxonomy reflects features supported by existing ADLs. In certain cases, also included are those characteristics typically not supported by ADLs, but which have been identified as important for architecture-based development.

To properly enable further discussion, several definitions are needed. There is no standard definition of architecture, but we will use as our working definition the one provided by Garlan and Shaw [GS93]:

> *[Software architecture is a level of design that] goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.*

An *ADL* is a language that provides features for modeling a software system's *conceptual* architecture. ADLs provide a concrete syntax and a conceptual framework for characterizing architectures [GMW97]. The conceptual framework typically subsumes the ADL's underlying semantic theory (e.g, CSP, Petri nets, finite state machines).

The building blocks of an architectural description are *components*, *connectors*, and *architectural configurations*.[4] An *ADL* must provide the means for their *explicit* specification; this enables us to determine whether or not a particular notation is an ADL. In order to infer any kind of information about an architecture, at a minimum, *interfaces* of constituent components must also be modeled. Without this information, an architectural description becomes but a collection of (interconnected) identifiers.

Several aspects of both components and connectors are desirable, but not essential: their benefits have been acknowledged and possibly demonstrated by some ADL, but their absence does not mean that a given language is not an ADL. These features are *interfaces* (for connectors), and *types*, *semantics*, *constraints*, and *evolution* (for both). Desirable features of configurations are *understandability*, *heterogeneity*, *compositionality*, *constraints*, *refinement and traceability*, *scalability*, *evolution*, and *dynamism*.

Finally, even though the suitability of a given language for modeling software architectures is independent of whether and what kinds of tool support it provides, an accompanying toolset will render an ADL both more usable and useful. The kinds of tools that are desirable in an ADL are those for *active specification*, *multiple views*, *analysis*, *refinement*, *code generation*, and *dynamism*.

This framework is depicted in Fig. 1. It is intended to be extensible and modifiable, which is crucial in a field that is still largely in its infancy. The features of a number of surveyed languages are still changing (e.g., SADL, ACME, C2, ArTek). Moreover, work is being continuously done on extending tool support for all ADLs. Sections 4-7 elaborate further on components, connectors, configurations, and tool support in ADLs. They motivate the taxonomy and compare existing ADLs based on their level of support of the different categories.[5]

---

[4] "Architectural configurations" will, at various times, be referred to as "configurations" or "topologies."

[5] Due to space restrictions, the comparison of ADLs in Sections 4-7 is limited to a representative subset of the languages whenever possible. A complete comparison of existing ADLs is given in [Med97].

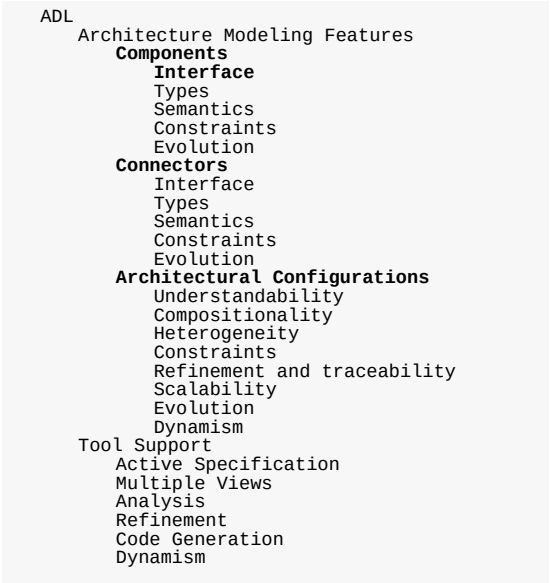

Figure 1. ADL classification and comparison framework. Essential modeling features are bolded.

3 Differentiating ADLs from Other Languages

In order to clarify what *is* an ADL, it may be useful to point out several notations (e.g., high-level design notations, MILs, PLs, OO modeling notations, and formal specification languages) that, though similar, are *not* ADLs according to our definition.

The requirement to model *configurations* explicitly distinguishes ADLs from some high-level design languages. Existing languages that are commonly referred to as ADLs can be grouped into three categories based on how they model configurations:

- *implicit configuration languages* model configurations implicitly through interconnection information that is distributed across definitions of individual components and connectors;
- *in-line configuration languages* model configurations explicitly, but specify component interconnections, along with any interaction protocols, "in-line;"
- *explicit configuration languages* model both components and connectors separately from configurations.

The first category, implicit configuration languages, are, by definition given in this paper, *not* ADLs, although they may serve as useful tools in modeling certain aspects of architectures. Two examples of such languages are LILEANNA and ArTek. In LILEANNA, interconnection information is distributed among *with* clauses of individual packages, package bindings (*view* construct), and compositions (*make*). In ArTek, there is no configuration specification; instead, each connector specifies component ports to which it is attached.

The focus on conceptual architecture and explicit treatment of *connectors* as first-class entities differentiate ADLs from MILs [DK76, PN86], PLs, and OO notations and languages (e.g., Unified Method [BR95]). MILs typically describe the *uses* relationships among modules in an *implemented* system and support only one type of connection [AG94a, SG94]. PLs describe a system's implementation, whose architecture

is typically implicit in subprogram definitions and calls. Explicit treatment of connectors also distinguishes ADLs from OO languages, as demonstrated in [LVM95].

It is important to note, however, that there is less than a firm boundary between ADLs and MILs. Certain ADLs, e.g., Wright and Rapide, model components and connectors at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation. We refer to these languages as *implementation independent*. On the other hand, several ADLs, e.g., UniCon and MetaH, require a much higher degree of fidelity of an architecture to its implementation. Components modeled in these languages are directly related to their implementations, so that a module interconnection specification may be indistinguishable from an architectural description in such a language. These are *implementation constraining* languages.

An ADL typically subsumes a formal semantic theory. That theory is part of an ADL's underlying framework for characterizing architectures; it influences the ADL's suitability for modeling particular kinds of systems (e.g., highly concurrent systems) or particular aspects of a given system (e.g., its static properties). Examples of formal specification theories are Petri nets [Pet62], Statecharts [Har87], partially-ordered event sets [LVB+93], communicating sequential processes (CSP) [Hoa85], model-based formalisms (e.g., *CH*emical *A*bstract *M*achine [IW95], Z [Spi89]), algebraic formalisms (e.g., Obj [GW88]), and axiomatic formalisms (e.g., Anna [Luc87]).

Of the above-mentioned formal notations, Z has been demonstrated appropriate for modeling only certain aspects of architectures, such as architectural style rules [AAG93, MTW96]. Partially-ordered event sets, CSP, Obj, and Anna have already been successfully used by existing modeling languages (Rapide, Wright, and LILEANNA, respectively). Modeling capabilities of the remaining three, Petri nets, Statecharts, and CHAM are somewhat similar to those of ADLs. Although they do not express systems in terms of components, connectors, and configurations per se, their features may be cast in that mold and they may be considered ADLs in their existing forms. In the remainder of this section we will discuss why it would be inappropriate to do so.[6]

3.1 Petri Nets

Petri net places can be viewed as components maintaining state, transitions as components performing operations, arrows between places and transitions as simple connectors, and their overall interconnection structure as a configuration. Petri nets mandate that processing components may only be connected to state components and vice-versa. This may be an unreasonable restriction. Overcoming it may require some creative and potentially counterintuitive architecting. A bigger problem is that Petri nets do not model component interfaces, i.e., they do not distinguish between different types of tokens. If we think of tokens as messages exchanged among components, this is a crucial shortcoming. Colored Petri nets [Jen92, Jen94] attempt to remedy this problem by allowing different types of tokens. However, even they explicitly model

[6] These three notations represent only a small subset of ADL-like formal specification languages. They are used here to draw attention to what differentiates them from ADLs and to outline heuristics for determining whether a potential candidate is indeed an ADL according to our definition. These heuristics can then be used in the future to evaluate other notations as possible ADLs.

only the interfaces of state components (places), but not of processing components (transitions). Therefore, Petri nets violate the definition of ADLs.

3.2 Statecharts

Statecharts is a modeling formalism based on finite state machines (FSM), which provides a state encapsulation construct, support for concurrency, and broadcast communication. To compare Statecharts to an ADL, the states would be viewed as components, transitions among them as simple connectors, and their interconnections as configurations. However, Statecharts does not model architectural configurations explicitly: interconnections and interactions among a set of concurrently executing components are implicit in *intra*-component transition labels. In other words, as was the case with LILEANNA and ArTek, the topology of an "architecture" described as a Statechart can only be ascertained by studying its constituent components. Therefore, Statecharts is not an ADL.

3.3 CHAM

In the CHAM approach, an architecture is modeled as an abstract machine fashioned after chemicals and chemical reactions. A CHAM is specified by defining molecules, their solutions, and transformation rules that specify how solutions evolve. An architecture is then specified with processing, data, and connecting elements. The interfaces of processing and connecting elements are implied by (1) their topology and (2) the data elements their current configuration allows them to exchange. The topology is, in turn, implicit in a solution and transformation rules. Therefore, even though CHAM can be used effectively to prove certain properties of architectures, without additional syntactic constructs it does not fulfill the requirements to be an ADL.

4 Components

A component is a unit of computation or a data store. Therefore, components are loci of computation and state [SDK+95]. A component in an architecture may be as small as a single procedure (e.g., MetaH *procedures*) or as large as an entire application (e.g., hierarchical components in C2 and Rapide or *macros* in MetaH). It may require its own data and/or execution space, or it may share them with other components.

Each surveyed ADL models components in one form or another. In this section, we present the aspects of components that need to be modeled in an ADL and assess existing ADLs with respect to them.

4.1 Interface

A component's interface is a set of interaction points between it and the external world. An interface specifies the services (messages, operations, and variables) a component provides. In order to be able to adequately reason about a component and its encompassing architecture, ADLs also typically provide facilities for specifying component needs, i.e., services required of other components in the architecture. Interfaces also enable a certain, though limited, degree of reasoning about component semantics.

All surveyed ADLs support specification of component interfaces. They differ in the terminology and the kinds of information they specify. For example, each interface point in MetaH, ACME, Aesop, and Wright is a *port*. On the other hand, in C2, the entire interface is provided through a single port; individual interface elements are *messages*. In UniCon, an interface point is a *player*, and in Rapide a *constituent*.

4.2 Types

Software reuse is one of the primary goals of architecture-based development [BS92, GAO95, MOT96]. Since architectural decomposition is performed at a level of abstraction above source code, ADLs can support reuse by modeling abstract components as types and instantiating them multiple times in an architectural specification. Abstract component types can also be parameterized, further facilitating reuse.[7]

All of the surveyed ADLs distinguish component types from instances. MetaH and UniCon support only a predefined set of types. Three ADLs make explicit use of parameterization: ACME, Darwin, and Rapide.

4.3 Semantics

Modeling of component semantics enables analysis, constraint enforcement, and mappings of architectures across levels of abstraction. Several languages do not model component semantics beyond interfaces. SADL and Wright focus on other aspects of architectural description (connectors and refinement), although, in principle, Wright allows specification of component functionality in CSP.

Underlying semantic models vary across those ADLs that do support specification of component semantics. For example, Rapide uses partially ordered event sets (posets), while Darwin uses π-calculus [MPW92]. MetaH and UniCon supply certain kinds of semantic information in property lists, e.g., specification of event traces in UniCon to describe component semantics. MetaH also uses an accompanying language, ControlH, for modeling algorithms in the guidance, navigation, and control domain [BEJV94].

4.4 Constraints

A constraint is a property of or assertion about a system or one of its parts, the violation of which will render the system unacceptable to one or more stakeholders [Cle96b]. In order to ensure adherence to intended component uses, enforce usage boundaries, and establish intra-component dependencies, constraints on them must be specified. Constraints may be defined in a separate constraint language or using the notation of the given ADL and its underlying semantic model.

All surveyed languages restrict component usage via interfaces. Specification of semantics further constrains internal elements of a component. Several ADLs provide additional means for specifying constraints on components: Wright specifies protocols of interaction with a component for each port; Rapide uses an algebraic language to specify constraints on the abstract state of a component; MetaH and UniCon constrain specification, implementation, and usage of components by specifying their non-functional attributes; SADL and Aesop provide and enforce stylistic invariants.

4.5 Evolution

As design elements, components evolve. ADLs should support component evolution through subtyping and refinement. Only a subset of existing ADLs provide this support. Even within those ADLs, evolution support is limited and often relies on the chosen implementation language. The remainder of the ADLs view and model components as inherently static.

---

[7] A detailed discussion of the role of parameterization in reuse is given in [Kru92].

Rapide supports inheritance. MetaH and UniCon define component types by enumeration, allowing no subtyping, and hence no evolution. ACME supports structural subtyping via its *extends* feature, while Aesop supports behavior preserving subtyping to create substyles. C2 provides a more advanced subtyping and type checking mechanism, described in [MORT96].

5 Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems. They may be separately compilable message routing devices, or shared variables, table entries, buffers, instructions to a linker, dynamic data structures, procedure calls, initialization parameters, client-server protocols, pipes, etc. [GMW95, SDK+95].

Surveyed ADLs model connectors in many forms. Languages such as C2, Wright, UniCon, ACME, and SADL model them explicitly and refer to them as *connectors*. In Rapide and MetaH they are *connections*, modeled in-line, and cannot be named, subtyped, or reused. Rapide does allow abstracting away complex connection behavior into a "connector component." Connectors in Darwin are *bindings* and are also specified in-line. In this section, we present the aspects of connectors that we believe need to be modeled in an ADL and compare existing ADLs with respect to them.

5.1 Interface

In order to enable proper connectivity of components and their communication in an architecture, a connector should export as its interface those services it expects. Therefore, a connector's interface is a set of interaction points between it and the components attached to it. It enables reasoning about the well-formedness of a configuration.

Only those ADLs that model connectors explicitly support specification of connector interfaces. Wright, Aesop, ACME, and UniCon refer to connector interface points as *roles*. Explicit connection of component ports (players in UniCon) and connector roles is then required in an architectural configuration. In C2, on the other hand, a connector's interface is modeled with *ports* and is determined by (potentially dynamic) interfaces of its attached components. This added flexibility may prove a liability when analyzing for interface mismatches between communicating components.

5.2 Types

Architecture-level communication is often expressed with complex protocols. To abstract away these protocols and make them reusable, ADLs should model connectors as types. This is typically done in two ways: as extensible type systems defined in terms of communication protocols and independent of implementation, or as enumerated types based on their implementation mechanisms.

Only ADLs that model connectors as first-class entities distinguish connector types from instances. This excludes MetaH, Rapide, and Darwin. Wright, ACME, C2, and Aesop base connector types on protocols. SADL and UniCon, on the other hand, only allow prespecified, though extensible, sets of connector types.

5.3 Semantics

To perform analyses of component interactions, consistent refinements across levels of abstraction, and enforcement of interconnection and communication constraints, archi-

tectural descriptions should provide connector protocol and transaction semantics. It is interesting to note that languages that do not model connectors as first-class objects, e.g., Rapide, may model connector semantics, while some ADLs that do model connectors explicitly, such as C2, do not provide means for defining their semantics.

ADLs generally use a single semantic model for both components and connectors. For example, Rapide uses posets, and Wright models connector *glue* with CSP. As was the case with its components, UniCon allows specification of semantic information for connectors in property lists.

5.4 Constraints

In order to ensure adherence to interaction protocols, establish intra-connector dependencies, and enforce usage boundaries, connector constraints must be specified. With the exception of C2, ADLs that model connectors as first-class objects constrain their usage via interfaces. Wright further constrains connectors by specifying protocols for each role, while UniCon restricts the types of players that can serve in a given role using the *Accept* attribute. C2 restricts the number of component ports that may be attached to each connector port (one). ADLs that specify connections in-line (e.g., Rapide, MetaH, and Darwin) place no such constraints on them.

5.5 Evolution

Component interactions are governed by complex and changing protocols. Maximizing connector reuse is achieved by modifying or refining existing connectors. ADLs can support connector evolution with subtyping and refinement.

ADLs that do not model connectors as first-class objects also provide no facilities for their evolution. Others currently only focus on component evolution (C2) or provide a predefined set of connector types with no evolution support (UniCon). Wright does not facilitate connector subtyping, but supports type conformance, where a role and its attached port may have behaviorally related, but not necessarily identical, protocols. Aesop and SADL provide more extensive support for connector evolution. Aesop supports behavior preserving subtyping, while SADL supports refinements of connectors across styles and levels of abstraction.

6 Configurations

Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether: appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior. In concert with models of components and connectors, descriptions of configurations enable assessment of concurrent and distributed aspects of an architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, etc. Configurations also enable analyses for adherence to design heuristics and style constraints.

6.1 Understandable Specifications

A major role of architectures is to facilitate understanding of systems at a high level of abstraction. Therefore, ADLs must model structural information with simple and understandable syntax, where system structure is clear from a configuration specification alone.

Configuration descriptions in *in-line configuration ADLs*, such as Rapide, Darwin, and MetaH, tend to be encumbered with connector details, while *explicit configuration ADLs*, such as UniCon, ACME, Wright, SADL, and C2 have the best potential to facilitate understandability of architectural structure.

Several languages provide both a graphical and textual notation. Graphical specification provides another way of achieving understandability. However, this is only the case if there is a precise relationship between the graphical description and the underlying model. UniCon, MetaH, Aesop, C2, Rapide, and Darwin, support such "semantically sound" graphical notations, while ACME, SADL, and Wright do not.

6.2 Compositionality

Architectures may be required to describe software systems at different levels of detail, where complex behaviors are either explicitly represented or abstracted away into single components and connectors. An ADL may also need to support situations in which an entire architecture becomes a single component in another, larger architecture. Therefore, support for compositionality, or hierarchical composition, is crucial.

Several ADLs provide explicit features to support hierarchical composition: MetaH macros, ACME templates and rep-maps, composite elements in Darwin and UniCon, internal component architecture in C2, and Rapide and SADL maps. Other ADLs, such as Wright, allow hierarchical composition in principle, but provide no specific constructs to support it.

6.3 Heterogeneity

A goal of architectures is to facilitate development of large systems, with components and connectors of varying granularity, implemented by different developers, in different programming languages, and with varying OS requirements. It is therefore important that ADLs provide facilities for architectural specification and development with heterogeneous components and connectors.

ADLs may be tightly tied to a particular formal modeling or implementation language, or they may support multiple such languages. Some ADLs fail to maximize reuse by supporting only certain types of components and connectors. For example, UniCon can use existing filters and sequential files, but not spreadsheets or constraint solvers. Most surveyed ADLs support modeling of both fine- and coarse-grain components. At one extreme, *computations* in UniCon or *procedures* in MetaH describe a single operation, while the other can be achieved by hierarchical composition, discussed above. Finally, MetaH requires that each component contain a loop with a call to a predeclared procedure to periodically dispatch a process. Any existing components have to be modified to include this construct.

6.4 Constraints

Constraints that depict desired dependencies among components and connectors in a configuration are as important as those specific to individual components and connectors. Many global constraints are derived from or directly dependent upon local constraints. For example, performance of a system will depend upon the performance of each individual element.

Only a handful of ADLs provide facilities for global constraint specification. Aesop, SADL, and C2 specify stylistic invariants. Aesop and SADL allow specification of invariants corresponding to different styles, while in C2 they refer to a single

(C2) style and are therefore fixed. Refinement maps in SADL and Rapide constrain valid configuration refinements. Rapide's timed poset constraint language can be used to constrain configurations. Finally, MetaH allows explicit constraint of *applications*[8] with non-functional attributes.

6.5 Refinement and Traceability

ADLs provide expressive and semantically elaborate facilities for specifying architectures. However, an ADL must also enable correct and consistent refinement of architectures to executable systems and traceability of changes across levels of refinement. This may very well be the area in which existing ADLs are most lacking.

Several languages enable system generation directly from architectural specifications; these are typically the *implementation constraining languages* (see Section 3). Both UniCon and MetaH allow specification of source files that corresponds to given architectural elements. There are several problems with this approach: the assumption that the relationship between architectural elements and those of the resulting implementation is 1-to-1 may be unreasonable; there is also no guarantee that specified source modules will correctly implement the desired behavior or that future changes to those modules will be traced back to the architecture and vice versa.

Only SADL and Rapide support refinement and traceability. Both provide refinement maps for architectures at different abstraction levels. SADL uses the maps to correctly refine architectures across styles, while Rapide generates comparative simulations of architectures at different abstraction levels. Both languages thus provide the means for tracing decisions across levels of architectural specification.[9]

6.6 Scalability

Architectures are intended to support large-scale systems. For that reason, ADLs must support specification and development of systems that may grow in size. Objectively evaluating an ADL's support for scalability is difficult, but certain heuristics can be of help.

One way of supporting scalability is through hierarchical composition, discussed in Section 6.2. Furthermore, it is generally easier to expand architectures described in *explicit configuration ADLs* (e.g., C2, UniCon) than *in-line configuration ADLs* (e.g., Rapide): connectors in the latter are described solely in terms of the components they connect; adding new components may require modification of existing connectors.

ADLs, such as C2, that allow a variable number of components to be attached to a connector are better suited to scaling up than those, such as Wright or ACME, which specify the exact number of components a connector can handle. UniCon allows architects to either specify the maximum number roles in a connector or leave it unbounded.

Note that these are heuristics and should not be used as the only criteria in excluding a candidate ADL from consideration. For example, both Wright and Rapide have been highlighted as examples of ADLs lacking scalability features, yet they have both been used to specify architectures of large, "real world" systems [All96, LKA+95].

---

[8] MetaH applications specify architectures containing both hardware and software elements of a system.

[9] [Med97] discusses in detail the drawbacks of each approach and motivates a hybrid approach.

6.7 Evolution

Architectures evolve to reflect evolution of a single software system; they also evolve into families of related systems. ADLs need to augment evolution support at the level of components and connectors with features for incremental development and support for system families.

Incrementality of an architectural configuration can be viewed from two different perspectives. One is its ability to accommodate addition of new components. Issues inherent in doing so were discussed above and the arguments applied to scalability also largely apply to incrementality.

Another view of incrementality is an ADL's support for incomplete architectural descriptions. Most existing ADLs and their supporting toolsets have been built to prevent precisely these kinds of situations. For example, UniCon, and Rapide compilers and constraint checkers raise exceptions if such situation arise. Thus, an ADL, such as Wright, which focuses its analyses on information local to a connector is better suited to accommodate incremental specification than, e.g., SADL, which is very rigorous in its refinement of *entire* architectures.

Another aspect of evolution is support for application families. In [MT96], we showed that the number of possible architectures in a component-based style grows exponentially as a result of a linear expansion of a collection of components. All such architectures may not belong to the same logical family. Therefore, relying on component and connector evolution mechanisms is insufficient. Aesop, and, more recently, ACME and Wright have provided support for system families.

6.8 Dynamism

Explicit modeling of architectures is intended to support development and evolution of large and potentially long-running systems. It may be necessary to evolve such systems during execution. Configurations exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or subarchitectures.

The majority of existing ADLs view configurations statically. The exceptions are Darwin, Rapide, and C2. Darwin allows runtime replication of components via *dyn*amic instantiation, as well as deletion and rebinding of components by interpreting Darwin scripts. Rapide's *where* clause supports a form of architectural rewiring at runtime. Finally, C2's architecture construction notation supports insertion, removal, and rewiring of elements in an architecture at runtime [Med96].

7 Tool Support for ADLs

A major impetus behind developing languages for architectural description is that their formality renders them suitable for manipulation by software tools. The usefulness of an ADL is directly related to the kinds of tools it provides to support architectural design, evolution, refinement, constraints, analysis, and executable system generation.

The need for tool support in architectures is well recognized. However, there is a definite gap between what is identified as desirable by the research community and the state of the practice. While every surveyed ADL provides some tool support, they tend to focus on a single area, such as analysis or refinement, and direct their attention to a particular technique (e.g., Wright's analysis for deadlocks), leaving other facets unexplored. This is the very reason ACME has been proposed as an architecture interchange language: to enable interaction and cooperation among different ADLs'

toolsets and thus fill in these gaps. This section surveys the tools provided by different ADLs, attempting to highlight the biggest shortcomings.

7.1 Active Specification

Only a handful of existing ADLs provide tools that support active specification of architectures. In general, such tools can be proactive or reactive. UniCon's graphical editor is proactive; it invokes the language processor's checking facilities to *prevent* errors during design. Reactive specification tools detect *existing* errors. They may either only inform the architect of the error or also force him to correct it before moving on. An example of the former is C2's Argo design environment [RR96], and of the latter MetaH's graphical editor.

7.2 Multiple Views

When defining an architecture, different stakeholders may require different views of the architecture. Customers may be satisfied with a "boxes-and-lines" description; developers may want detailed component and connector models; managers may require a view of the corresponding development process.

Several ADLs (e.g., Rapide, UniCon, Aesop, MetaH, Darwin, and C2) support two basic views of an architecture: textual and graphical. UniCon, MetaH, and Aesop further distinguish different types of components and connectors iconically, while Darwin and C2, for example, do not. Each of these ADLs allows both top-level and detailed views of composite elements.

Support for other views is sparse. C2 provides a view of the development process that corresponds to the architecture [RR96]. Rapide and C2 allow visualization of an architecture's execution behavior by building a simulation and providing tools for viewing and filtering events generated by the simulation.

7.3 Analysis

Architectural descriptions are often intended to model large, distributed, concurrent systems. Evaluating properties of such systems upstream, at architectural level, can substantially lessen the costs of any errors.

The types of analyses for which an ADL is well suited depend on its underlying semantic model. For example, Wright, which is based on CSP, analyzes individual connectors for deadlocks. MetaH and UniCon both support schedulability analysis by specifying non-functional properties, such as criticality and priority. SADL can establish relative correctness of two architectures with respect to a refinement map. Rapide's and C2's event monitoring and filtering tools also facilitate analysis of architectures. C2 uses critics to establish adherence to style rules and design guidelines.

Language parsers and compilers are another kind of analysis tools. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All of the surveyed languages have parsers. UniCon, MetaH, and Rapide also have compilers, which enable these languages to generate executable systems from architectural descriptions. Wright uses FDR [For92], a model checker, to establish type conformance.

Another aspect of analysis is enforcement of constraints. Parsers and compilers enforce constraints implicit in types, non-functional attributes, component and connector interfaces, and semantic models. Rapide's constraint checker also analyzes the conformance of a Rapide simulation to formal constraints defined in the architecture.

7.4 Refinement

The importance of supporting refinement of architectures across styles and levels of detail was argued in Section 6.5 and, more extensively, in [MQR95] and [Gar96]. Refining architectural descriptions is a complex task whose correctness cannot always be guaranteed by formal proof, but adequate tool support can give us increased confidence in this respect.

Only SADL and Rapide provide tool support for refinement of architectures. SADL requires manual proofs of mappings of constructs between an abstract and a concrete style. Such a proof need be performed only once, after which SADL provides a tool that checks whether two architectural descriptions adhere to the mapping.

Rapide's event pattern mappings ensure behavioral consistency between architectures. Maps are used to verify that the events generated by simulating an architecture satisfy constraints in the architecture to which it is mapped.

7.5 Code Generation

The ultimate goal of software design and modeling is to produce the executable system. An elegant and effective architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. It is, therefore, desirable, if not imperative, for an ADL to provide code generation tools.

A large number of ADLs, but not all, do so. Aesop generates C++ code, MetaH - Ada, UniCon - C, and C2 - Java, C++, and Ada. Rapide can construct executable systems in C, C++, Ada, VHDL, and Rapide. On the other hand, SADL, ACME, and Wright are used strictly as modeling notations and provide no code generation support.

7.6 Dynamism

Given that the support for modeling dynamism in existing ADLs is limited, it is of no surprise that tool support for dynamism is not very prevalent. Rapide can model only planned modifications at runtime; its compilation tools ensure that all possible configuration alternatives are enabled. C2's *ArchShell* tool [Ore96, MOT97], on the other hand, currently enables arbitrary interactive construction, execution, and runtime-modification of C2-style architectures implemented in Java. Darwin supports both planned (the *dyn* construct) and unplanned runtime changes (interpretation of Darwin scripts)

8 Conclusions and Future Work

Classifying and comparing any two languages objectively is a difficult task. For example, a PL, such as Ada, contains MIL-like features and debates rage over whether Java is "better" than C++ and why. On the other hand, there exist both an exact litmus test (Turing completeness) and a way to distinguish different kinds of PLs (imperative vs. declarative vs. functional, procedural vs. OO). Similarly, formal specification languages have been grouped into model-based, state-based, algebraic, axiomatic, etc. Until now, however, no such definition or classification existed for ADLs.

The main contribution of this paper is just such a definition and classification framework. The definition provides a simple litmus test for ADLs that largely reflects community consensus on what is essential in modeling an architecture: an architectural description differs from other notations by its *explicit* focus on connectors and architectural configurations. We have demonstrated how the definition and the accompany-

ing framework can be used to determine whether a given notation is an ADL and, in the process, discarded several notations as potential ADLs. Some (LILEANNA and ArTek) may be more surprising than others (Petri nets and Statecharts), but the same criteria were applied to all.

Of those languages that passed the litmus test, several straddled the boundary by either modeling their connectors in-line (*in-line configuration ADLs*) or assuming a bijective relationship between architecture and implementation (*implementation constraining ADLs*). We have discussed the drawbacks of both categories. Nevertheless, it should be noted that, by simplifying the relationship between architecture and implementation, *implementation constraining ADLs* have been more successful in generating code than "mainstream" (*implementation independent*) ADLs. Thus, for example, although C2 is implementation independent, we assumed this 1-to-1 relationship in building the initial prototype of our code generation tools [MOT97].

Finally, neither the definition nor the accompanying framework have been proposed as immutable laws on ADLs. Quite the contrary, we expect both to be modified and extended in the future. We are currently considering several issues: providing a clearer distinction between descriptive languages (e.g., ACME) and those that primarily provide semantic modeling (e.g., Wright); distinguishing style- or domain-specific ADLs from "general purpose" ADLs; and expanding the "top" level of the framework to include criteria such as support for system families, openness, and extensibility. We have also had to resort to heuristics and subjective criteria in comparing ADLs at times, indicating areas where future work should be concentrated. However, what this taxonomy provides is an important first attempt at answering the question of what an ADL is and why, and how it compares to other ADLs. Such information is needed both for evaluating new and improving existing ADLs, and for targeting architecture interchange efforts more precisely.

9 References

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 9-20, Los Angeles, CA, December 1993.

[AG94a] R. Allen and G. Garlan. Formal Connectors. Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994.

[AG94b] R. Allen and G. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.

[All96] R. Allen. HLA: A Standards Effort as Architectural Style. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 130-133, San Francisco, CA, October 1996.

[BEJV94] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. To appear in *International Journal of Software Engineering and Knowledge Engineering*, January 1994, revised February 1995.

[BR95] G. Booch and J. Rumbaugh. *Unified Method for Object-Oriented Development*. Rational Software Corporation, 1995.

[BS92] B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the Software Technology Conference 1992, pages 63-82*, Los Angeles, April 1992. DARPA.

[Cle96a] P. C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

[Cle96b] P. C. Clements. Succeedings of the Constraints Subgroup of the EDCS Architecture and Generation Cluster, October 1996.

[DK76] F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-

small. *IEEE Transactions on Software Engineering*, pages 80-86, June 1976.

[For92] *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems (Europe) Ltd., Oxford, England, October 1992.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[Gar95] D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.

[Gar96] D. Garlan. Style-Based Refinement for Software Architecture. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 72-75, San Francisco, CA, October 1996.

[GMW95] D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219, Carnegie Mellon University, November 1995.

[GMW97] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. Submitted for publication, January 1997.

[GPT95] D. Garlan, F. N. Paulisch, and W. F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, February 1995. Reprinted in ACM Software Engineering Notes, pages 63-83, July 1995.

[GS93] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-99. SRI International, 1988

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IW95] P. Inverardi and A. L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, pages 373-386, April 1995.

[Jan92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*. Volume 1: Basic Concepts. *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1992.

[Jen94] K. Jensen. *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. In J. W. de Bakker, W. P. De Roever, and G. Rozenberg, eds., volume 803 of *A Decade of Concurrency, Lecture Notes in Computer Science*, pages 230-272, Springer-Verlag, 1994.

[KC94] P. Kogut and P. Clements. Features of Architecture Description Languages. Draft of a CMU/SEI Technical Report, December 1994.

[KC95] P. Kogut and P. Clements. Feature Analysis of Architecture Description Languages. In *Proceedings of the Software Technology Conference (STC'95)*, Salt Lake City, April 1995.

[Kru92] C. W. Krueger. *Software reuse*. Computing Surveys, pages 131-184, June 1992.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, pages 336-355, April 1995.

[Luc87] D. Luckham. *ANNA, a language for annotating Ada programs: reference manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[LV95] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.

[LVB+93] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Journal of Systems and Software*, pages 253-265, June 1993.

[LVM95] D. C. Luckham, J. Vera, and S. Meldal. Three Concepts of System Architecture. Unpublished Manuscript, July 1995.

[Med96] N. Medvidovic. ADLs and Dynamic Architecture Changes. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24-27, San Francisco, CA, October 1996.

[Med97] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report, UCI-ICS-97-02, University of California, Irvine, January 1997.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3-14, San Francisco, CA, October 1996.

[MOT96] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 17-23, 1997.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. *A Calculus of Mobile Processes, Parts I and II*. Volume 100 of *Journal of Information and Computation*, pages 1-40 and 41-77, 1992.

[MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pages 356-372, April 1995.

[MT96] N. Medvidovic and R. N. Taylor. Reusing Off-the-Shelf Components to Develop a Family of Applications in the C2 Architectural Style. In *Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families*, Las Navas del Marqués, Ávila, Spain, November 1996.

[MTW96] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28-40, Los Angeles, CA, April 1996.

[NB92] P. Newton and J. C. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.

[Ore96] Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report, UCI-ICS-96-35, University of California, Irvine, August 1996.

[PC94] P. Kogut and P. Clements. Features of Architecture Description Languages. Draft of a CMU/SEI Technical Report, December 1994.

[Pet62] C. A. Petri. Kommunikationen Mit Automaten. PhD Thesis, University of Bonn, 1962. English translation: Technical Report RADC-TR-65-377, Vol.1, Suppl 1, Applied Data Research, Princeton, N.J.

[PN86] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, pages 307-334, October 1989.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.

[RR96] J. E. Robbins and D. Redmiles. Software architecture design from the perspective of human cognitive needs. In *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pages 314-335, April 1995.

[SG94] M. Shaw and D. Garlan. Characteristics of Higher-Level Languages for Software Architecture. Technical Report, CMU-CS-94-210, Carnegie Mellon University, December 1994.

[SG95] M. Shaw and D. Garlan. *Formulations and Formalisms in Software Architecture*. Springer-Verlag Lecture Notes in Computer Science, Volume 1000, 1995.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall, New York, 1989.

[TLPD95] A. Terry, R. London, G. Papanagopoulos, and M. Devito. The ARDEC/Teknowledge Architecture Description Language (ArTek), Version 4.0. Technical Report, Teknowledge Federal Systems, Inc. and U.S. Army Armament Research, Development, and Engineering Center, July 1995.

[Tra93] W. Tracz. LILEANNA: A Parameterized Programming Language. In *Proceedings of the Second International Workshop on Software Reuse*, pages 66-78, Lucca, Italy, March 1993.

[Ves93] S. Vestal. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center, February 1993.

[Ves96] S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.

[Wolf96] A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

[Wolf97] A. L. Wolf. Succeedings of the Second International Software Architecture Workshop (ISAW-2). *ACM SIGSOFT Software Engineering Notes*, pages 42-56, January 1997.