A Classification and Comparison Framework for Software Architecture Description Languages

Neno Medvidovic
Technical Report UCI-ICS-97-02
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425

neno@ics.uci.edu

February 1996

Abstract

Software architectures shift the focus of developers from lines-of-code to coarser-grained architectural elements and their overall 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 of several possible ADLs 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 also used to classify and compare several existing ADLs. One conclusion is that, although much research has been done in this area, no single ADL fulfills all of the identified needs.

I. 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, Gar95a, Wolf96].

A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages. In this paper, we specifically consider those languages most commonly referred to as ADLs: Aesop [GAO94, Gar95b, GKMM96], ArTek [TLPD95], C2 [MTW96, MORT96, Med96], Darwin [MDK93, MDEK95, MK95, MK96], LILEANNA [Tra93a, Tra93b], MetaH [BEJV94, Ves96], Rapide [LKA+95, LV95, Rap96], SADL [MQR95], UniCon [SDK+95, SDZ96], and Wright [AG94a, AG94b].[1] Recently, initial work has been done on an architecture interchange language, ACME [GMW95, GMW96], which is intended to support mapping of architectural specifications from one ADL to another, and hence enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains a number of ADL-like features. Furthermore, it is useful to compare and differentiate it from other ADLs to highlight the difference between an ADL and an interchange language. It is, therefore, included in this paper.

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 on the other. Indeed, for

1. The full name of the ADL for C2-style architectures is "C2 SADL." In order to distinguish it from SADL, which resulted from an unrelated project at SRI, it will be referred to simply as "C2" in the remainder of the paper.

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].

Another source of discord is the level of support an ADL should provide to developers. At one end of the spectrum, it can be argued that the primary role of architectural descriptions is to aid understanding and communication about a software system. As such, an ADL must have simple, understandable, and possibly graphical syntax, well understood, but not necessarily formally defined, semantics, and the kinds of tools that aid visualization, understanding, and simple analyses of architectural descriptions (e.g., Argo [RR96, RHR96]). At the other end of the spectrum, the tendency has been to provide formal syntax and semantics of ADLs, powerful analysis tools, model checkers, parsers, compilers, code synthesis tools, runtime support tools, and so on (e.g., SADL's architecture refinement patterns [MQR95], Darwin's use of π-calculus to formalize architectural semantics [MK96], or UniCon's parser and compiler [SDK+95]). While both perspectives have merit, ADL researchers have generally adopted one or the other extreme view. It is our contention that both are important and should be reflected in an ADL to a certain degree.

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 definitive 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., programming languages), 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 II discusses contributions and shortcomings of other attempts at surveying and classifying ADLs. Section III motivates our definition and taxonomy of ADLs. Section IV demonstrates the utility of the definition by determining whether several existing notations are ADLs. Sections V-VIII describe the elements of components, connectors, configurations, and tool support, respectively, and assess the above ADLs based on these criteria. Discussion and conclusions round out the paper.

II. Related Approaches

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, GMW96] 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].

II.A. Previous Surveys

Kogut and Clements [KC94, KC95, Cle96a] provide an extensive classification of existing ADLs. The classification is based on an exhaustive questionnaire of ADL characteristics and features, completed by each language's design team. The survey was conducted in a top-down fashion: the authors used domain analysis techniques to decide what features an ADL should have, and then assessed existing languages with respect to those features.

While their taxonomy is valuable in bettering our understanding of surveyed ADLs, it comes up short in several respects. Domain analysis is typically used in well-understood domains, which is not the case with ADLs. Beyond this, the survey does not provide any deeper insight into what an ADL is, nor does it present its criteria for inclusion of a particular modeling notation in the list. Quite the contrary, the list of surveyed languages contains several that are commonly not considered ADLs, yet little justification is given for their inclusion. Perhaps most illustrative is the example of Modechart, a specification language for hard-real-time computer systems [JM94]. Clements labels Modechart "a language on the edge of ADLs," whose utility to the architecture community lies in its sophisticated analysis and model checking toolset [Cle95]. Tool support alone is not a sufficient reason to consider it an ADL, however. Other similar examples are

- CODE [NB92], a graphical parallel *programming* language;
- Demeter [HSX91], an approach to object-oriented design and programming;
- Resolve [HLOW94], a mathematically-based approach to reusable component-based software development, which is related more closely to formal specification languages, such as Larch [GH93] and Z [Spi89], than to other ADLs; and
- PSDL [KLB93], a rapid prototyping language for real-time systems.

Several of the criteria Kogut and Clements used for ADL evaluation, such as the ability to model requirements and algorithms, are outside an ADL's scope. Furthermore, this kind of survey

runs the risk of not asking all of the relevant questions. Finally, the authors often have to extrapolate very specific information from multiple, potentially subjective, vague, or misunderstood questions.

Vestal's approach [Ves93] is more bottom-up. He surveyed four existing ADLs (LILEANNA, MetaH, Rapide, and QAD [HP93]) and attempted to identify their common properties. He concluded that they all model or support the following concepts, though not to the same degree:

- components,
- connections,
- hierarchical composition, where one component contains an entire subarchitecture,
- computation paradigms, i.e., semantics, constraints, and non-functional properties,
- communication paradigms,
- underlying formal models,
- tool support for modeling, analysis, evaluation, and verification, and
- automatic application code composition.

Although "cursory" and limited in its scope, Vestal's survey contains useful insights that bring us closer to answering the question of what an ADL is. In its approach, our survey is much closer to Vestal's than to Clements and Kogut's.

II.B. Insights from Individual Systems

In [LV95], Luckham and Vera list requirements for an ADL, based on their work on Rapide:

- component abstraction,
- communication abstraction,
- communication integrity, which mandates that only components that are connected in an architecture may communicate in the resulting implementation,
- ability to model dynamic architectures,
- hierarchical composition, and
- relativity, or the ability to relate (map) behaviors between architectures.

As a result of their experience with UniCon, Shaw and colleagues list the following properties an ADL should exhibit [SDK+95]:

- ability to model components, with property assertions, interfaces, and implementations,
- ability to model connectors, with protocols, property assertions and implementations,
- abstraction and encapsulation,
- types and type checking, and
- ability to accommodate analysis tools.

Clearly, the above features alone cannot be considered definitive indicators of how to identify an ADL. They have resulted from limited experience of two research groups with their own languages. However, they represent valuable data points in trying to understand and classify ADLs.

II.C. Attempts at Identifying Underlying Concepts

In [Wolf97], Tracz defines an ADL as consisting of four "C"s: components, connectors, configurations, and constraints. This taxonomy is appealing, especially in its simplicity, but needs further elaboration: justification for and definitions of the four "C"s, aspects of each that need to be modeled, necessary tool support, and so on.

Shaw and Garlan have attempted to identify unifying themes and motivate research in ADLs. Both authors have successfully argued the need to treat connectors explicitly, as first-class entities in an ADL [Sha93, AG94a, AG94b, SG94]. In [SG94], they also elaborate six classes of properties that an ADL should provide: composition, abstraction, reusability, configuration, heterogeneity, and analysis. They demonstrate that other existing notations, such as informal diagrams, modularization facilities provided by programming languages, and MILs, do not satisfy the above properties and hence cannot fulfill architecture modeling needs.

In [SG95], Shaw and Garlan identify seven levels of architecture specification capability:

- capturing architectural information,
- construction of an instance,
- composition of multiple instances,
- selection among design or implementation alternatives,
- verifying adherence of an implementation to specification,
- analysis, and
- automation.

They conclude that, while ADLs invariably provide notations for capturing system descriptions (level 1), few support other levels. It is unclear, however, what set of criteria they applied to the different ADLs and how stringent those criteria were, particularly since this paper will show that a number of ADLs do provide a considerable amount of support for most of the above capabilities.

Finally, in [MTW96], Medvidovic and colleagues argue that, in order to adequately support architecture-based development and analysis, one must model them at four levels of abstraction: internal component semantics, component interfaces, component interconnections in an architecture, and architectural style rules. This taxonomy presents an accurate high-level view of

architecture modeling needs, but is too general to serve as an adequate ADL comparison framework. Furthermore, it lacks any focus on connectors.

II.D. Architecture Interchange

Perhaps the closest the research community has come to a consensus on ADLs has been its endorsement of ACME as an architecture interchange language [GMW95, GMW96]. In order to meaningfully interchange architectural specifications across ADLs, a common basis for all ADLs must be established. Garlan and colleagues believe that common basis to be their core ontology for architectural representation:

- components,
- connectors,
- systems, or configurations of components and connectors,
- ports, or points of interaction with a component,
- roles, or points of interaction with a connector,
- representations, used to model hierarchical compositions, and
- rep-maps, which map a composite component or connector's internal architecture to elements of its external interface.

In ACME, any other aspect of architectural description is represented with property lists (i.e., it is not core).

ACME has resulted from a careful consideration of issues in and notations for modeling architectures. As such, it can be viewed as *the* starting point for studying existing ADLs and developing new ones. However, ACME represents the least common denominator of existing ADLs rather than a definition of an ADL. It also does not provide any means for understanding or classifying those features of an architectural description that are placed in property lists. Finally, certain structural constraints imposed by ACME, such as its requirement that a connector may not be directly attached to another connector, satisfy the needs of some approaches (e.g., Aesop, UniCon, Wright), but not of others (e.g., C2).

III. ADL Classification and Comparison Framework

Individually, none of the above attempts adequately answer the question of what an ADL *is*. Instead, they reflect their authors' views on what an ADL *should have* or *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 complete the framework, the characteristics of individual ADLs

and summaries of discussions on ADLs that occurred at the two International Software Architecture Workshops [Gar95a, Wolf96] were studied. To a large degree, this taxonomy reflects features supported by all, or most, 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, universally-accepted 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 both a concrete syntax and a conceptual framework for characterizing architectures [GMW96]. The conceptual framework reflects characteristics of the domain for which the ADL is intended and/or the architectural style. The 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 (1) *components*, (2) *connectors*, and (3) *architectural configurations*.[2] 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*, *evolution*, and *non-functional properties* (for both). Desirable features of configurations are *understandability, heterogeneity, compositionality, constraints, refinement and traceability, scalability, evolution, dynamism* and *non-functional properties*.

2. "Architectural configurations" will, at various times in this paper, be referred to simply as "configurations" or "topologies."

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 taxonomy is depicted in Fig. 1. The taxonomy 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 V-VIII 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.

ADL
- Architecture Modeling Features
  - **Components**
    - **Interface**
    - Types
    - Semantics
    - Constraints
    - Evolution
    - Non-functional properties
  - **Connectors**
    - Interface
    - Types
    - Semantics
    - Constraints
    - Evolution
    - Non-functional properties
  - **Architectural Configurations**
    - Understandability
    - Compositionality
    - Heterogeneity
    - Constraints
    - Refinement and traceability
    - Scalability
    - Evolution
    - Dynamism
    - Non-functional properties
- Tool Support
  - Active Specification
  - Multiple Views
  - Analysis
  - Refinement
  - Code Generation
  - Dynamism

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

IV. Differentiating ADLs from Other Languages

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

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 the 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 the *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], programming languages, 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, AG94b, SG94]. Programming languages 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.

Darwin has elements of both implementation constraining and independent languages: it ties each primitive component to its implementation, but also enables modeling of high-level, composite components. Given this composition feature and the fact that a valid Darwin architecture need not contain primitive components, we are inclined to consider Darwin an implementation independent language.

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., chemical abstract machine [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 chemical abstract machines, 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.

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

IV.B. Statecharts

Statecharts is a modeling formalism based on finite state machines (FSM) that 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.

There is, however, an even deeper issue in attempting to model architectures as FSMs. Namely, even though it may be useful to represent component or connector semantics with Statecharts, it is doubtful that an adequate architectural breakdown of a system can be achieved from a state-machine perspective. Harel [Har87] agrees with this, arguing that

> *one has to assume some physical and functional description of the system, providing, say, a hierarchical decomposition into subsystems and the functions and activities they support. This description should also identify the external input and output ports and their associated signals. Statecharts can then be used to control these internal activities. Although we are aware of the fact that achieving such a functional decomposition is by no means a trivial matter, we assume that this kind of description is given or can be produced using an existing method.*

IV.C. Chemical Abstract Machine

In the chemical abstract machine (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 the 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.

V. 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 and under various names. ACME, Aesop, C2, Darwin, SADL, UniCon, and Wright share much of their vocabulary and refer to them simply as *components*; in Rapide they are *interfaces*;[3] and in MetaH *processes*. 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.

V.A. Interface

A component's interface is a set of interaction points between it and the external world. As in OO classes or Ada package specifications, a component interface in an ADL specifies those services (messages, operations, and variables) the component provides. In order to be able to adequately reason about a component and the architecture that includes it, ADLs should also provide facilities for specifying component needs, i.e., services required of other components in the architecture. An interface thus defines computational commitments a component can make and constraints on its usage. Interfaces also enable a certain, though rather 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 ACME, Aesop, SADL, 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 Darwin, an interface point is a *service*, in Rapide a *constituent*, and in UniCon a *player*. MetaH distinguishes between *ports*, *events*, and *shared data*.

3. *Interface* is a language construct; the authors commonly refer to components as "components."

The ports in ACME, Aesop, SADL, and Wright are named and typed. Aesop and SADL distinguish between input and output ports (*inputs* and *outputs* in Aesop; *iport* and *oport* in SADL). Wright goes a step further by specifying the expected behavior of the component at that point of interaction. The particular semantics of a port are specified in CSP [Hoa85] as interaction protocols. In the example given in Fig. 2 below, *DataRead* is a simple input (read only) port:

```
component DataUser =
    port DataRead = get → DataRead □ √
    other ports ...
```

Fig. 2. Interaction protocol for a component port in Wright.

A C2 component interface consists of single top and bottom ports. Both incoming and outgoing message traffic is routed through each port. An important distinction among C2 messages is between *requests* and *notifications*. Due to C2's principle of *substrate independence*, a component has no knowledge of components below it in an architecture. For that reason, any messages sent down an architecture must be notifications of that component's internal state; requests may only be sent up.

Component interface specifications in Darwin specify services *provided* and *required* by a component, as well as types of those services. Each service type is further elaborated with an interaction mechanism that implements the service. For example, *trace* services are implemented with *events*, *outputs* are accomplished via *ports*, and *commands* accept *entry* calls.

MetaH specifies input and output ports on components (processes). Ports are strongly typed and connections among them type checked. They are the means for periodic communication: each port has an associated buffer variable and port-to-port communication results in assignment. Aperiodic communication is modeled by output events. Finally, sharable monitors or packages are the means of indicating shared data among components.

Rapide subdivides component interfaces into constituents: *provides*, *requires*, *action*, and *service*. *Provides* and *requires* refer to functions. Connections between them specify synchronous communication. *In* and *out actions* denote the events a component can observe and generate, respectively. Connections between *actions* define asynchronous communication. A service is an aggregation facility for a number of actions and functions. It is a mechanism for abstracting and reusing component interface elements.

Finally, UniCon specifies interfaces as sets of players. Players are visible semantic units through which a component interacts by requesting or providing services and receiving external state and events. Each player consists of a name, a type, and optional attributes such as signature, functional specification, or constraints. UniCon supports a predefined set of player types: *RoutineDef*, *RoutineCall*, *GlobalDataDef*, *GlobalDataUse*, *ReadFile*, *WriteFile*, *ReadNext*, *WriteNext*, *StreamIn*, *StreamOut*, *RPCDef*, *RPCCall*, and *RTLoad*. *PLBundle* denotes a set of players.

V.B. Types

Software reuse is one of the primary goals of architecture-based development [BS92, GAO95, MOT97]. Since architectural decomposition is performed at a level of abstraction above source code, ADLs can support reuse by modeling abstract components as types. Component types can then be instantiated multiple times in an architectural specification and each instance may correspond to a different implementation of the component. Abstract component types can also be parameterized, further facilitating reuse.[4]

All of the surveyed ADLs distinguish component types from instances. Rapide does so with the help of a separate types language [LKA+95]. With the exception of MetaH and UniCon, all ADLs provide extensible component type systems. MetaH and UniCon support only a predefined, built-in set of types. MetaH component types are *process*, *macro*, *mode*, *system*, and *application*.[5] Component types supported by UniCon are *Module*, *Computation*, *SharedData*, *SeqFile*, *Filter*, *Process*, *SchedProcess*, and *General*.

Three ADLs make explicit use of parameterization: ACME, Darwin, and Rapide. ACME provides component templates, which are typed and parameterized macro facilities for specifications of recurring component patterns. Parameterized types in Rapide are type *constructors*; applying them to appropriate arguments results in a type. ACME and Darwin only allow parameterization of component type declarations, while Rapide also allows the behavior associated with a particular type to be parameterized by specifying event patterns, discussed below.

4. A detailed discussion of the role of parameterization in reuse is given in [Kru92].
5. As MetaH is used to specify both the software and the hardware architecture of an application, *system* is a hardware construct, while *application* pertains to both.

V.C. Semantics

In order to be able to perform useful analyses, enforcement of constraints, and consistent mappings of architectures from one level of abstraction to another, component semantics should be modeled. However, several languages do not model component semantics beyond interfaces. SADL and Wright focus on other aspects of architectural description (connectors and refinement). Wright does enable specification of interaction protocols for each component interface point, and while it does not focus on it, it also allows specification of component functionality in CSP.

Underlying semantic models and their expressive power vary across those ADLs that do support specification of component behavior. ACME and UniCon allow semantic information to be specified in components' property lists. ACME places no restrictions on the specification language; however, from its point of view, properties are uninterpreted, so that, strictly speaking, component semantics are outside the scope of the language. Although UniCon's main focus is on non-functional properties of components (see Section V.F), it allows specification of event traces in property lists to describe component behavior.

Aesop does not provide any language mechanisms for specifying component semantics. However, for each architectural style defined in Aesop, it allows the use of style-specific languages for modeling semantics.

MetaH allows specification of component implementation semantics with path declarations. A path declaration consists of an optional identifier, followed by the names of (more primitive) components in that path. MetaH also uses an accompanying language, ControlH, for modeling algorithms in the guidance, navigation, and control (GN&C) domain [BEJV94].

In Rapide, each component specification has an associated *behavior*, which is defined via partially ordered sets of events (posets). Rapide uses event patterns to recognize posets. During poset recognition, free variables in a pattern are bound to specific matching values in a component's poset. Event patterns are used both as triggers and outputs of component state transitions.

C2 currently employs a more primitive semantic model. Component semantics are expressed in terms of causal relationships between input and output messages in its interface. At the level of a configuration, this information can be used to generate linear traces of events, similar to VHDL's [VHDL87] and Verilog's [TM91]. Rapide's posets are a related but more powerful modeling mechanism [LV95].

Finally, Darwin uses π-calculus [MPW92] as its underlying semantic model. A system in the π-calculus is a collection of independent processes which communicate via named channels.

π-calculus is used to model basic component interaction and composition properties, so that each syntactic Darwin construct concerned with requiring, providing, and binding services is modeled in it. It is important to note that using π-calculus in this manner only supports modeling the semantics of composite Darwin components (further discussed in Section VII.B), while primitive components are treated as black boxes.

V.D. 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 dependencies among internal elements of a component, constraints on them must be specified. Constraints may be defined in a separate constraint language, or they may be specified using the notation of the given ADL and its underlying semantic model.

All surveyed languages constrain usage of components by specifying their interfaces as the only legal means of interaction. Formal specification of component semantics further specifies relationships and dependencies among internal elements of a component. Several ADLs provide additional means for specifying constraints on components:

- Aesop, C2, and SADL provide stylistic invariants. Unlike Aesop and SADL, which support multiple architectural styles, C2's invariants, such as the number of communication ports and distinction between requests and notifications, are specific to a single style and are therefore fixed.
- MetaH constrains implementation and usage of a component by specifying its non-functional properties or attributes, such as *Period*, *ExecutionTime*, *Deadline*, *Criticality*, *TimeSliceOption*, and *AllowedBinding*.
- UniCon also constrains component usage with attributes, such as *EntryPoint* into a component and *Priority*. Attributes are either *required* or *optional*. For each attribute, a rule must be specified on how to handle its multiple specifications within a single component. For example, parameters used for component instantiation (*InstFormals*) are *merge*d, while any new occurrence of Priority *replace*s the previous one. UniCon also restricts the types of players that can be provided by certain types of components. For each player, its maximum and minimum numbers of connections are specified (*MaxAssocs* and *MinAssocs*, respectively).
- Rapide uses an Anna-like algebraic constraint language to specify constraints on the abstract state of a component. It also enables specification of pattern constraints on event posets that are generated and observed from a component's interface. Pattern constraints specify how to use a particular component and what the component promises to do. In the example shown in

Fig. 3 below, the constraint implies that all, and only, messages taken in by the *Resource* component are delivered

- Wright specifies protocols of interaction with a component for each port in CSP.

```
type Resource is interface
    public action Receive(Msg : String);
    extern action Results(Msg : String);
constraint
    match
        ((?S in String)(Receive(?S)->Results(?S)))^(*~);
end Resource;
```

Fig. 3. A pattern constraint in Rapide.

V.E. Evolution

As design elements, components evolve. ADLs must support the evolution process by allowing subtyping of components and refinement of their features. Only a subset of existing ADLs provide support for evolution. Even within those ADLs, evolution support is limited and often relies on the chosen implementation (programming) language. The remainder of the ADLs view and model components as inherently static.

MetaH and UniCon define component types by enumeration, allowing no subtyping, and hence no evolution support. ACME has only recently introduced types, but currently provides no subtyping features.

Aesop supports behavior-preserving subtyping to create substyles of a given architectural style. Aesop mandates that a subclass must provide strict subtyping behavior for operations that succeed, but may also introduce additional sources of failure with respect to its superclass.

Rapide allows its interface types to inherit from other types by using OO methods, resulting in structural subtyping. Both Rapide and SADL also provide features for refinement of components across levels of abstraction. This mechanism may be used to evolve components by explicating any deferred design decisions, which is somewhat similar to extending inherited behavior in OO languages. It is interesting to note that, in a general case, subtyping is simply a form of refinement. This is, however, not true in the case of Rapide and SADL, both of which place additional constraints on refinement maps in order to prove or demonstrate certain properties of architectures (see Section VII.E).

One ADL that stands out in its support for component evolution is C2. C2 attempts to avoid dependence on subtyping mechanisms provided by any underlying programming language. Its approach is based on the realization that architectural design is a complex activity in which architectures may incorporate components implemented in heterogeneous programming languages; therefore, an ADL cannot rely on a single subtyping method provided by any one language. C2 models conceptual component placeholders as formal parameters, while the implemented components that instantiate them are actual parameters. Multiple subtyping and type-checking relationships among components are allowed: name, interface, behavior, and implementation subtyping, as well as their combinations [MORT96].

V.F. Non-Functional Properties

Specification of non-functional properties of components is needed to enable simulation of their runtime behavior, perform useful analyses on components, enforce constraints, map component implementations to processors, and aid in project management (e.g., by specifying stringent performance requirements, development of a component may need to be assigned to the best engineer). Despite the need for and potential benefits of specifying non-functional properties, there is a striking lack of support for them in existing ADLs.

ACME allows specification of a superset of all ADLs' non-functional properties in its property lists. However, as discussed above, it neither interprets nor does it make any use of them. Aesop allows association of arbitrary text with component specifications. Such arbitrary text may include non functional properties, although such a possibility has not been explicitly considered by Aesop's developers.

The two ADLs that stand out in their ability to express non-functional properties of components are MetaH and UniCon. Both of these languages need such information to analyze architecture for real-time schedulability (both ADLs) and reliability and security (MetaH). Both also use source code location attributes for code generation. Several representative non-functional properties in MetaH are *SourceName*, *SourceFile*, *ClockPeriod*, *Deadline*, and *Criticality.* UniCon allows specification of *Priority*, *Library*, *ImplType* (*source*, *object*, *executable*, *data*, or *whatever*), and *Processor.*

V.G. Summary of ADL Components

Overall, surveyed ADLs provide comprehensive support for modeling components. All of them regard components as first-class entities. Furthermore, all model interfaces and distinguish

between component types and instances. On the other hand, a majority of the ADLs do not support evolution or non-functional properties. It is illustrative that Aesop is the only ADL that provides at least some support for each of the six classification categories and that, of the four ADLs that support five of the categories, C2 and Rapide do not model non-functional properties, and MetaH and UniCon do not support evolution. Every ADL supports or allows at least four of the six categories.

A more complete summary of this section is given in Table 1 below.

Table 1: ADL Support for Modeling Components

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Component*; implementation independent	interface points are *ports*	extensible type system; parameterization enabled with templates	no support; can use other ADLs' semantic models in property lists	via interfaces only	none	allows any attribute in property lists, but does not operate on them
Aesop	*Component*; implementation independent	interface points are *input* and *output ports*	extensible type system	(optional) style-specific languages for specifying semantics	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with components
C2	*Component*; implementation independent	entire interface one *port*; interface elements are *messages* (*notifications* and *requests*)	extensible type system	causal relationships between input and output messages	via interfaces and semantics; stylistic invariants	name, interface, behavior and implementation subtyping (and their combinations)	none
Darwin	*Component*; implementation independent;	interface points are *services* (*provide*d and *require*d)	extensible type system; supports parameterization	π-calculus	via interfaces and semantics	none	none
MetaH	*Process*; implementation constraining	interface points are *ports*	Predefined, enumerated set of types	ControlH for modeling algorithms in the GN&C domain; implementation semantics via paths	via interfaces and semantics; modes; non-functional attributes	none	attributes needed for real-time schedulability, reliability, and security analysis
Rapide	*Interface*; implementation independent	interface points are *constituents* (*provides*, *requires*, *action*, and *service*)	extensible type system; contains a types sublanguage; supports parameterization	partially ordered event sets (posets)	via interfaces and semantics; algebraic constraints on component state; pattern constraints on event posets	inheritance (structural subtyping)	none
SADL	*Component*; implementation independent;	interface points are input and output *ports* (*iports* and *oports*)	extensible type system; component types tightly coupled to styles	none	via interfaces; stylistic invariants	component refinement via pattern maps	none
UniCon	*Component*; implementation constraining	interface points are *players*	predefined, enumerated set of types	event traces in property lists	via interfaces and semantics; attributes; restrictions on players that can be provided by component types	none	attributes for schedulability analysis
Wright	*Component*; implementation independent;	interface points are *ports*; port interaction semantics specified in CSP	extensible type system	not the focus; allowed in CSP	protocols of interaction for each port in CSP	none	none

VI. Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors may not correspond to compilation units in an implemented system. They may be implemented as separately compilable message routing devices (e.g., C2), but may also manifest themselves as shared variables, table entries, buffers, instructions to a linker, dynamic data structures, sequences of procedure calls embedded in code, initialization parameters, client-server protocols, pipes, SQL links between a database and an application, and so on [GMW95, SDK+95]. As such, connector specifications in an ADL may also need to contain hints for implementing a particular kind of connector.

As in the case of components, surveyed ADLs model connectors in various forms and under various names. For example, languages such as ACME, Aesop, C2, SADL, UniCon, and Wright model connectors explicitly and refer to them as *connectors*. In Rapide and MetaH they are *connections*, modeled in-line, and cannot be named, subtyped, or reused (i.e., connectors are not first-class entities).[6] Connectors in Darwin are *bindings* and are also specified in-line, i.e., in the context of a configuration only. 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.

VI.A. 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 an architectural configuration.

Only those ADLs that support modeling of connectors explicitly, independently of configurations in which they are used, support specification of connector interfaces. ACME, Aesop, UniCon, and Wright refer to connector interface points as *roles*. Explicit connection of component ports and connector roles is required in an architectural configuration.

Roles are named and typed, and are in many ways similar to component ports (players in UniCon), discussed in Section V.A. Aesop distinguishes between input and output roles (*sources* and *sinks*). Semantics of each role's interaction protocol in Wright are specified in CSP, similar to port protocols shown in Fig. 2. This allows for analysis of compatibility between connected

6. MetaH does allow a connection to be named optionally, but that feature has not been demonstrated in any examples in published literature about MetaH. Based on that, and given the primitive nature of a MetaH connection, it is unclear what purpose connection names may serve in an architecture.

component ports and connector roles. In UniCon, each role may include optional attributes, such as the type of players that can serve in the role and minimum and maximum number of connections. UniCon also supports only a predefined set of role types: *Source*, *Sink, Reader, Readee*, *Writer*, *Writee*, *Definer, Caller, User*, *Participant*, and *Load*.

In C2, connector interfaces, like component interfaces, are modeled with *ports*. Each port can export multiple messages and sets of messages at two different ports need not be disjoint. In general, the interface of a C2 connector (the messages it understands) is determined by (potentially dynamic) interfaces of components that communicate through it. This added flexibility and dynamism may prove a liability when analyzing for interface mismatches between communicating components.

Although Darwin and Rapide define their connectors in-line, both languages allow abstracting away complex connection behaviors into "connector components," which are then accompanied by a set of simple connections (bindings in Darwin).

Finally, a SADL connector only exports the type of data it supports in its interface. Other information about the connector, such as the number of components it connects, is implicit in the connector type (see Section VI.B) and/or specified as part of the architectural configuration (Section VII).

VI.B. Types

Architecture-level communication may need to be 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 which are defined in terms of communication protocols and are independent of implementation, or as built-in, enumerated types which are based on their implementation mechanisms.

Only those ADLs that model connectors as first-class entities distinguish connector types from instances. This excludes languages like Darwin, MetaH, and Rapide. ACME, Aesop, C2, and Wright base connector types on protocols. ACME also provides a parameterization facility through connector templates.

SADL and UniCon, on the other hand, only allow connectors of prespecified enumerated types. UniCon currently supports *Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcCall*, and *RTScheduler* connector types, while SADL allows a single connector type for each of its supported architectural styles: *BatchSequential, ControlTransfer, Dataflow, Functional*, *ProcessPipeline*, and *SharedMemory.*

MetaH does not support connector types, but it does define three broad categories of connections. In *port* connections, an *out* port of one component may be connected to an *in* port of another. *Event* connections allow outgoing events to be connected to incoming events (event-to-event), as well as to their recipient components (event-to-process and event-to-mode). Finally, *equivalence* connections specify objects that are shared among components.

VI.C. Semantics

To perform useful analyses of component interactions, consistent refinement mappings across levels of architectural abstraction, and enforcement of interconnection and communication constraints, architectural 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 ADLs that do model connectors explicitly, such as C2, do not always provide means for defining their semantics.

ADLs generally use a single mechanism for specifying the semantics of both components and connectors. For example, ACME allows connector semantics to be specified in its property lists using any specification language, but considers them uninterpreted; Rapide uses posets to describe communication patterns among its components; Wright models connector *glue* and event trace specifications with CSP; and UniCon allows specification of semantic information for connectors in property lists (e.g., a real-time scheduling algorithm or path traces through real-time code). Additionally, connector semantics in UniCon, as well as SADL, are implicit in their connector types. For example, declaring a component to be a *pipe* implies certain functional properties.

One exception to this rule is Aesop, which uses a different semantic model for its connectors than it does for components. Namely, Aesop does not use style-specific formal languages, but (optionally) employs Wright to specify connector semantics. Finally, while C2 does not model the behavior of connectors, it does provide an insight into how a connector will behave by specifying its message filtering policies: *no_filtering*, *notification_filtering*, *prioritized*, and *message_sink*.

VI.D. Constraints

In order to ensure adherence to intended interaction protocols, establish intra-connector dependencies, and enforce usage boundaries, connector constraints must be specified. With the exception of C2, whose connector interfaces are a function of their attached components (see Section VI.A), ADLs that model connectors as first-class objects constrain their usage via interfaces. None of the ADLs that specify connections in-line (Darwin, MetaH, and Rapide) place any such constraints on them.

Implementation and usage of connectors is further constrained in those ADLs that model connector semantics (see Section VI.C). Aesop, C2, and SADL may also impose stylistic invariants, such as C2's restriction that each connector port may only be attached to a single other port. UniCon can also restrict the number of component players attached to a connector role by using the *MinConns* and *MaxConns* attributes. Finally, the types of players that can serve in a given role are constrained in UniCon via the *Accept* attribute and in Wright (and, transitively, in Aesop) by specifying interaction protocols for each role.

VI.E. Evolution

Component interactions are governed by complex and ever changing and expanding protocols. Maximizing connector reuse is achieved by modifying or refining existing connectors whenever possible. As with components, ADLs can support connector evolution with subtyping and refinement.

Even fewer ADLs support evolution of connectors than do evolution of components. ADLs that do not model connectors as first-class objects (Darwin, MetaH, and Rapide) also provide no facilities for their evolution. Others either 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, similar to their support for component evolution discussed in Section V.E. Aesop supports behavior preserving subtyping, while SADL supports refinements of connectors across styles and levels of abstraction.

VI.F. Non-Functional Properties

Modeling non-functional properties of connectors enables simulation of runtime behavior, useful analyses of connectors, constraint enforcement, and selection of appropriate OTS connectors (e.g., message busses) and their mappings to processors. Of the surveyed ADLs, only UniCon supports explicit specification of non-functional connector properties. UniCon uses such information to analyze an architecture for real-time schedulability. Its *SchedProcess* connector has an *Algorithm* attribute. If the value of *Algorithm* is set to *RateMonotonic*, UniCon uses trace, period, execution time, and priority information for schedulability analysis. As already mentioned, ACME allows specification of a superset of all ADLs' non-functional properties in its property lists, but does not directly utilize them, while Aesop allows association of arbitrary text with its connectors.

VI.G. Summary of ADL Connectors

The support provided by the ADLs for modeling connectors is considerably less extensive than for components. Three ADLs (Darwin, MetaH, and Rapide) do not regard connectors as first-class entities, but rather model them in-line. Their connectors are always specified as instances and cannot be manipulated during design or reused in the future. Overall, their support for connectors is negligible, as can be observed in Table 2 below.

All ADLs that model connectors explicitly also model their interfaces and distinguish connector types from instances. It is interesting to note that, as in the case of components, support for evolution and non-functional properties is rare, and that Aesop is again the only ADL that provides at least some support for each classification category.

A more complete summary of this section is given in Table 2.

Table 2: ADL Support for Modeling Connectors

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols	no support; can use other ADLs' semantic models in property lists	via interfaces only	none	allows any attribute in property lists, but does not operate on them
Aesop	*Connector*; explicit	interface points are *roles*	extensible type system, based on protocols	(optional) semantics specified in Wright (CSP)	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with connectors
C2	*Connector*; explicit	interface between connector and each component given through a separate *port*; interface elements are *messages*	extensible type system, based on protocols	partial semantics specified via message filters	via semantics; stylistic invariants (each port participates in one link only)	none; current focus is on component evolution	none
Darwin	*Binding*; in-line; no explicit modeling of component interactions	none; allows "connection components"	none	none	none	none	none
MetaH	*Connection*; in-line; allows connections to be optionally named	none	none; supports three general classes of connections: port, event, and equivalence	none	none	none	none
Rapide	*Connection*; in-line; complex reusable connectors only via "connection components"	none; allows "connection components"	none	posets; conditional connections	none	none	none
SADL	*Connector*; explicit	connector signature specify the supported data type	predefined, enumerated set of types, one per style	implicit in connector's type (e.g., dataflow)	via interfaces; stylistic invariants	connector refinement via pattern maps	none
UniCon	*Connector*; explicit	interface points are *roles*	predefined, enumerated set of types	implicit in connector's type; semantic information can be given in property lists	via interfaces; restricts the type of players that can be used in a given role	none	attributes for schedulability analysis
Wright	*Connector*; explicit	interface points are *roles*; role interaction semantics specified in CSP	extensible type system, based on protocols	connector *glue* semantics in CSP	via interfaces and semantics; protocols of interaction for each role in CSP	supports type conformance for behaviorally related protocols	none

VII. 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, and so on. Descriptions of configurations also enable analyses of architectures for adherence to design heuristics, e.g., to determine whether an architecture is "too deep," which may affect performance due to message traffic across many levels and/or process splits, or "too broad," which may result in too many dependencies among components (a "component soup" architecture). Finally, architectural description is necessary to establish adherence to architectural style constraints, such as C2's rule that there are no direct communication links between components.

Architectures are likely to describe large, long-lived software systems that may evolve over time. The changes to an architecture may be planned or unplanned; they may also occur before or during system execution. ADLs must support such changes through features for modeling evolution (before execution) and dynamism (during execution). Another key role for modeling architectural configurations is to facilitate communication for the many stakeholders in the development of a system. The goal of configurations is to abstract away the details of individual components and connectors. They depict the system at a high level that can potentially be understood by people with various levels of technical expertise and familiarity with the problem at hand. This section will investigate whether and to what degree various ADLs fulfill these roles.

VII.A. Understandable Specifications

One of the major roles of software architectures is that they facilitate understanding of (families of) systems at a high level of abstraction. To truly enable easy communication about a system among developers and other stakeholders, ADLs must model structural (topological) information with simple and understandable syntax. The structure of a system should ideally be clear from a configuration specification alone, i.e., without having to study component and connector specifications.

Configuration descriptions in *in-line configuration ADLs*, such as Darwin, MetaH, and Rapide tend to be encumbered with connector details, while *explicit configuration ADLs*, such as

ACME, Aesop, C2, SADL, UniCon, and Wright have the best potential to facilitate understandability of architectural structure. Clearly, whether this potential is realized or not will also depend on the particular ADL's syntax. For example, UniCon falls in the latter category, but it allows the connections between players and roles to appear in any order, possibly distributed among individual component and connector specifications; establishing what the topology of such an architecture is may (unnecessarily) require studying a significant portion of the architectural description.

Several languages provide a graphical, in addition to the textual, notation. Graphical specification of architectural configurations is another means of achieving understandability. However, this is only the case if there is a precise relationship between a graphical description and the underlying model, such that the textual and graphical descriptions are interchangeable. Languages like Aesop, C2, Darwin, MetaH, Rapide, and UniCon support such "semantically sound" graphical notations, while ACME, SADL, and Wright do not. It is important to note that a graphical specification of an architecture may not contain all the information in its textual counterpart (e.g., formal component and connector specifications), and vice versa (e.g., graphical layout information). Additional tool support is needed to make the two truly interchangeable (see Section VIII.B).

VII.B. 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 individual 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: ACME *templates*, *representations*, and *rep-maps*; Aesop *representations*; composite components in Darwin and UniCon; internal component architecture in C2 (shown in Fig. 4 and discussed below); MetaH *macros*; and Rapide *maps*. Other ADLs, such as SADL and Wright, allow hierarchical composition in principle, but provide no specific constructs to support it. It is interesting to note that Darwin and UniCon do not have an explicit constructs for modeling architectures, but model them simply as composite components.

VII.C. Heterogeneity

A goal of software architectures is to facilitate development of large-scale systems, preferably with pre-existing components and connectors of varying granularity, specified by different designers, potentially in different formal modeling languages, implemented by different developers, possibly in different programming languages, with varying operating system requirements, and supporting different communication protocols. It is therefore important that ADLs provide facilities for architectural specification and development with heterogeneous components and connectors.

Although no ADL provides explicit support for multiple specification languages, ACME, Aesop, and Darwin do allow it in principle. ACME's property lists are open, and will accept any modeling notation. To actually achieve architectural interchange, however, explicit mappings are required from architectural models described in one notation to another. Aesop allows style-specific modeling languages for component semantics, in addition to using Wright for modeling connectors. The possibility of using multiple notations for components within a *single* style is not precluded either. Finally, Darwin uses π-calculus to model external (visible) component characteristics and the semantics of composite components. At the same time, it leaves open the choice of specification languages for the semantics of primitive components.

Of the ADLs that support implementation of architectures, several are tightly tied to a particular programming language. For example, Aesop and Darwin only support development with components implemented in C++, while MetaH is exclusively tied to Ada[7] and UniCon to C. On the other hand, C2 currently supports development in C++, Ada, and Java, while Rapide supports construction of executable systems specified in VHDL, C, C++, Ada, and Rapide itself.

MetaH places additional restrictions on components, requiring that each component contain a loop with a call to the predeclared procedure `KERNEL.AWAIT_DISPATCH` to periodically dispatch a process. Any existing components have to be modified to include this construct before they can be used in a MetaH architecture.

C2 places a somewhat similar constraint on OTS components in that it assumes some form of message-based communication. C2 remedies this by assuming a particular internal component structure, shown in Fig. 4, in which an OTS component is effectively wrapped as a C2 component's *internal object*, and all message traffic is handled by the component's *dialog*.

7. Some preliminary work has recently been done in MetaH to also support components developed in C.

ADLs may also preclude reuse of many existing components and connectors by allowing only certain types of each. For example, UniCon can use existing filters and sequential files, but not spreadsheets, constraint solvers, or relational databases. Similarly, component and connector types in SADL are directly dependent upon the set of architectural styles it currently supports.

Finally, most surveyed ADLs support modeling of both fine and coarse-grain components. At one extreme are components that describe a single operation, such as *computations* in UniCon or *procedures* in MetaH, while the other can be achieved by hierarchical composition, discussed in Section VII.B above and, in the case of C2, also depicted in Fig. 4.

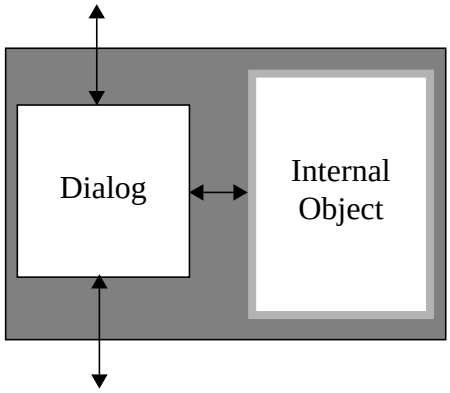

Fig. 4. Internal architecture of a C2 component: an OTS component is placed inside a C2 component's internal object. Hierarchical composition is achieved by placing a C2 architecture inside the internal object.

VII.D. Constraints

Constraints that depict desired dependencies among components and connectors in a configuration are as important as those specific to individual components and connectors. In general, however, existing ADLs have focused more on local than configuration-level constraints. Many global constraints are derived from or directly dependent upon local constraints. For example, constraints on the validity of a configuration may be expressed as a set of interaction constraints among components and connectors, which in turn are expressed through their interfaces and protocols; performance of a system described by a configuration will depend upon the performance of each individual architectural element; and safety of an architecture is always a function of the safety of its constituents.

A handful of ADLs do provide facilities for global constraint specification. ACME, Aesop, UniCon, and Wright require that a connector role always be attached to a component port/player.

Similarly, Darwin only allows bindings between provided and required services. Rapide's timed poset language [LVB+93] can be used to constrain configurations, as well as individual components. Furthermore, refinement maps in SADL and Rapide provide constraints on valid refinements of a configuration. Finally, MetaH allows explicit constraint of *applications* in the manner similar to that of individual components, i.e., with non-functional attributes.

Aesop, C2, and SADL specify stylistic invariants. For example, the C2 style mandates that only a single architectural element may be attached to each component and connector port and that no direct component-to-component links may exist [MTW96]. Aesop and SADL allow specification of structural invariants corresponding to different styles, while in C2 they refer to a single (C2) style.

VII.E. Refinement and Traceability

The most common argument for creating and using ADLs is that they are necessary to bridge the gap between informal, high-level "boxes and lines" diagrams and programming languages, which are deemed too low-level. We have thus far seen that ADLs provide architects with expressive and semantically elaborate facilities for specification of architectures. However, an ADL must also enable correct and consistent refinement of architectures to executable systems and traceability of changes across levels of architectural 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 III). MetaH and UniCon, as well as Darwin, allow the specification of a source file that corresponds to the given architectural element. There are several problems with this approach. Primarily, there is an assumption that the relationship between elements of an architectural description and those of the resulting executable system will be 1-to-1. This is not always necessary, and may also be unreasonable, as architectures are intended to describe systems at a higher level of abstraction than source code modules.[8] Secondly, there is no guarantee that the specified source modules will correctly implement the desired behavior. This brings us to the third problem: even if the specified modules currently implement the needed behavior correctly, this approach provides no means of ensuring that any future changes to those modules are traced back to the architecture and vice versa.

SADL and Rapide are the only two ADLs we studied that provide extensive support for refinement and traceability of architectural configurations. Both languages provide maps for

8. This is one of the differences between an ADL and an MIL, discussed in Section IV.

refining architectures across different levels of abstraction. SADL uses the maps to enable correct architecture refinements across styles,[9] while Rapide generates comparative simulations of architectures at different abstraction levels. Both languages thus provide the means for traceability of design decisions and changes from one level of architectural specification (or implementation) to another.

Each approach has certain drawbacks that the other alleviates, giving hope that a hybrid approach may be both possible and useful. SADL formally defines the mapping patterns and proves their validity according to a very strict correctness-preserving criterion ("interpretation mapping") [MQR95]. The proof of each map is performed only once, after which it may be used repeatedly. The strictness of the correctness criterion may render it impractical in certain cases, however. For example, some design decisions may be deliberately delayed and left out of a high-level architecture. SADL would then simply disallow those decisions to be made at a lower level of abstraction, regarding them as inconsistent with respect to its interpretation mapping.

Rapide maps need not adhere to such stringent rules. Instead, Rapide requires behavioral conformance and communication integrity in architectures at two different levels of abstraction. This approach may not be restrictive enough, as it is possible for a higher-level architecture to produce behaviors that are eliminated as more design decisions are made at the lower levels. This is precisely the reason Moriconi and colleagues opted for interpretation mappings in SADL. Rapide also fully places the responsibility of ensuring the correctness of a map on the architect.

Garlan has recently argued that refinement should not be consistent with respect to a single (immutable) law, but rather with respect to particular properties of interest, be they conservative extension (SADL), computational behavior (Rapide), or something entirely different, such as performance [Gar96]. This may be a good starting point towards a successful marriage of the two approaches.

VII.F. 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. For the purpose of this discussion, we can generalize the issues inherent in scaling software systems, so that an architectural configuration, such as that depicted in Fig. 5, can be scaled up in two ways: by

9. Moriconi and colleagues [MQR95] make the claim that different styles are at different levels of abstraction. While this is sometimes true, e.g., the "shared variable" style is at a lower level of abstraction than the "dataflow" style, it is arguable whether this is always the case.

adding components and connectors along its boundaries (Fig. 5a), and by adding elements to architecture's interior (Fig. 5b). To support the former, ADLs can employ compositionality features, discussed in Section VII.B by treating the original architecture as a single, composite component, which is then attached to new components and connectors. Objectively evaluating an ADLs ability to support the latter, i.e., adding internal elements, is more difficult, but certain heuristics can be of help.

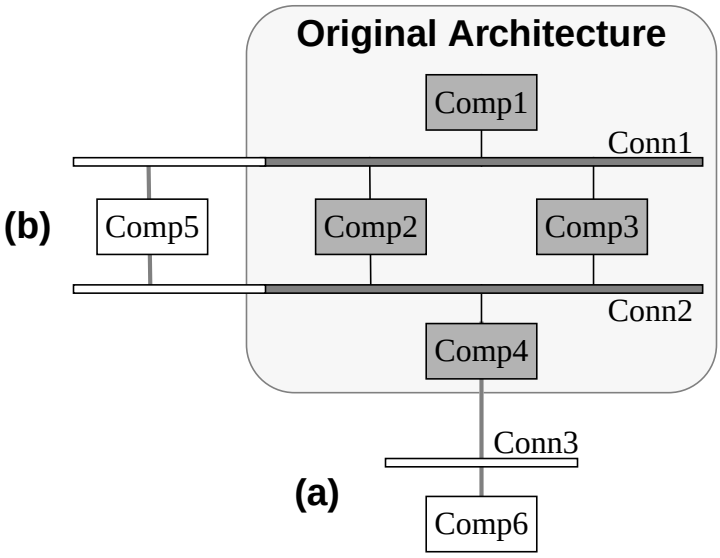

Fig. 5. An existing architecture is being scaled up: (a) by expanding the architecture "outward" and (b) by adding new components/connectors to its interior.

It is generally easier to expand architectures described in *explicit configuration ADLs* (ACME, Aesop, C2, SADL, UniCon, and Wright) than *in-line configuration ADLs* (Darwin, MetaH, and Rapide): connectors in the latter are described solely in terms of the components they connect; adding new components or connectors may require direct modification of existing connector instances.

ADLs, such as C2 and UniCon, that allow a variable number of components to be attached to a single connector are better suited to scaling up than those, such as ACME, Aesop, or Wright, which specify the exact number of components a connector can handle. For example, ACME, Aesop, and Wright could not handle the extension to the architecture shown in Fig. 5b without redefining *Conn1* and *Conn2*, while C2 and UniCon can.[10]

10. In UniCon, the *MaxConns* role attribute is unbounded by default.

It is important to remember that these are heuristics and should not be used as the only criteria in excluding a candidate ADL from consideration. The ultimate determinant of an ADL's support for modeling scalable configurations is not its supposed elegance in specifying large architectures and extending existing ones. As already discussed, most ADLs provide features for compositionality, so that a configuration of any size may be represented relatively succinctly at a high-enough level of abstraction. What scalability ultimately comes down to is the ability of developers to implement and/or analyze large-scale systems based on those descriptions. To date, only a subset of the existing ADLs have been applied to large-scale, "real-world" examples:

- Wright was used to model and analyze the *Runtime Infrastructure* (RTI) of the Department of Defense (DoD) *High-Level Architecture for Simulations* (HLA) [All96]. The original specification for RTI was over 100 pages long. Wright was able to condense the specification and detect several inconsistencies and weaknesses in it.
- SADL was applied to an operational power-control system, used by the Tokyo Electric Power Company. The system was implemented in 200,000 lines of Fortran 77 code. SADL was used to formalize the system's reference architecture and ensure its consistency with the implementation architecture.
- Rapide has been used in several large-scale projects thus far. A representative example is the X/Open Distributed Transaction Processing (DTP) Industry Standard. The documentation for the standard is over 400 pages long. Its reference architecture and subsequent extensions have been successfully specified and simulated in Rapide [LKA+95].

It is telling that 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.

VII.G. Evolution

Support for software evolution is a key aspect of architecture-based development. Architectures evolve to reflect and enable 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 (Section V.E) and connectors (Section VI.E) 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 in the manner depicted in Fig. 5. The issues inherent in doing so were discussed in the preceding subsection. The arguments that were applied to scalability also largely apply to incrementality: in general, *explicit configuration ADLs* can support incremental development more easily and effectively

than *in-line configuration ADLs*; ADLs that allow variable numbers of components to communicate through a connector are well suited for incremental development, particularly when faced with unplanned architectural changes.

Another view of incrementality is an ADL's tolerance and/or support for incomplete architectural descriptions. Incomplete architectures are common during design, as some decisions are deferred and others have not yet become relevant. It would therefore be advantageous for an ADL to allow incomplete descriptions. However, most existing ADLs and their supporting toolsets have been built around the notion that precisely these kinds of situations must be prevented. For example, Darwin, MetaH, Rapide, and UniCon compilers, constraint checkers, and runtime systems have been constructed to raise exceptions if such situation arise. In this case, an ADL, such as Wright, which focuses its analyses on information local to a single connector is better suited to accommodate expansion of the architecture 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 inheritance, subtyping, or other evolution mechanisms is insufficient. However, no existing ADLs provide direct support for families of systems; no counterpart to subtyping or inheritance exists at the level of configurations. One approach may be to exploit compositionality features and apply subtyping or inheritance to composite components. Another possible solution would take advantage of an ADL's support for non-functional attributes: in addition to components and connectors used in a configuration, such an ADL could also specify the application family to which the architecture belongs.

VII.H. Dynamism

Explicit modeling of software architectures is intended to support development and evolution of large and potentially long-running systems. Such systems may need to be modified to remove operational bottlenecks, improve performance, or upgrade their functionality. Being able to evolve such systems during execution may thus be necessary. Architectural 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 C2, Darwin, and Rapide. Darwin and Rapide support only *constrained* dynamic manipulation of

architectures, where all runtime changes must be known a priori [Ore96]. Darwin allows runtime replication of components via dynamic instantiation (the *dyn* operator) and conditional configuration. Darwin only provides support for unidirectional communication with a dynamically created component: it allows the component to request services of other components, but not to declare bindings to services it provides. Establishing this binding is the responsibility of component designers who must ensure that the component passes service references in messages to form bindings dynamically.

Rapide supports conditional configuration and dynamic generation of events. Its *where* clause enables a form of architectural rewiring at runtime, using the *link* and *unlink* operators. Rapide connection rules use poset patterns to generate new sets of poset matching events dynamically [Rap96].

On the other hand, C2 supports *pure* dynamic manipulation, where no restrictions are made on the types of allowed dynamic changes at architecture specification time. C2's architecture construction notation (*ACN*) specifies a set of operations for insertion, removal, and rewiring of elements in an architecture at runtime: *AddComponent*, *RemoveComponent*, *Weld*, and *Unweld* [Med96, Ore96]. C2's message-based communication and support for partial communication and partial service utilization [MTW96, TMA+96, MT96] alleviates the problem, encountered in Darwin, of using services provided by a dynamically inserted component.

VII.I. Non-Functional Properties

As discussed previously, non-functional properties are needed to perform useful analyses enforce the desired constraints, map architectural building blocks to processors, and aid in project management. All the ADLs that support specification of non-functional properties in components and connectors (ACME, Aesop, MetaH, and UniCon) also support hierarchical composition; hence, they can always specify such properties on composite components which encompass entire architectures. However, MetaH is the only language that supports direct modeling of non-functional properties of architectures (MetaH *applications*), such as processor on which the system will execute and clock period. Finally, Rapide allows modeling of timing information in its constraint language with its timed poset model [LVB+93].

VII.J. Summary of ADL Configurations

It is at the level of configurations that the foci of some ADLs can be more easily noticed. For example, SADL's particular contribution is in architectural refinement, while Darwin mostly focuses on system compositionality and dynamism. No single ADL satisfies all of the classification criteria, although Rapide comes close. Coverage of several criteria is sparse across ADLs: refinement and traceability, evolution, dynamism, and non-functional properties. These are good indicators of where future research should be directed. On the other hand, most ADLs allow or also provide explicit support for understandability, compositionality, and heterogeneity.

A more complete summary of this section is given in Table 3 below.

Table 3: ADL Support for Modeling Architectural Configurations

Features / *ADL*	Character.	Understand.	Composition.	Heterogen.	Constraints	Refinement/ Traceability	Scalability	Evolution	Dynamism	Non-Funct. Properties
ACME	*Attachments*; explicit	explicit, concise textual specification	provided via templates, representations, and rep-maps	open property lists; required explicit mappings across ADLs	ports may only be attached to roles and vice versa	none	aided by explicit configurations; hampered by fixed number of roles	aided by explicit configurations; no support for application families;	none	none
Aesop	*Configuration*; explicit	explicit, concise graphical specification; parallel type hierarchy for visualization	provided via representations	allows multiple languages for modeling semantics; supports development in C++ only	ports may only be attached to roles and vice versa; programmable stylistic invariants	none	aided by explicit configurations; hampered by fixed number of roles	no support for partial architectures or application families; aided by explicit configurations	none	none
C2	*Architectural Topology*; explicit	explicit, concise textual and graphical specification	allowed; supported via internal component architecture	enabled by internal component architecture; supports development in C++, Java, and Ada	fixed stylistic invariants	none	aided by explicit configurations and variable number of connector ports	allows partial architectures; aided by explicit configurations; no support for application families;	pure dynamism: element insertion, removal, and rewiring	none
Darwin	*Bind*ing; in-line	implicit textual specification which contains many connector details; provides graphical notation	supported by language's composite component feature	allows multiple languages for modeling semantics of primitive components; supports development in C++ only	provided services may only be bound to required services and vice versa	supports system generation when implementation constraining	hampered by in-line configurations	no support for partial architectures or application families; hampered by in-line configurations;	constrained dynamism: runtime replication of components and conditional configuration	none
MetaH	*Connections*; in-line	implicit textual specification which contains many connector details; provides graphical notation	supported via macros	supports development in Ada only; requires all components to contain a process dispatch loop	*applications* are constrained with non-functional attributes	supports system generation; implementation constraining	hampered by in-line configurations	no support for partial architectures or application families; hampered by in-line configurations;	none	supports attributes such as execution processor and clock period
Rapide	*Connect*; in-line	implicit textual specification which contains many connector details; provides graphical notation	mappings relate an architecture to an interface	supports development in VHDL, C/C++, Ada, and Rapide	refinement maps constrain valid refinements; timed poset constraint language	refinement maps enable comparative simulations of architectures at different levels	hampered by in-line configurations; used in large-scale projects	no support for partial architectures or application families; hampered by in-line configurations;	constrained dynamism: conditional configuration and dynamic event generation	timed poset model allows modeling of timing information in the constraint language
SADL	*Configuration*; explicit	explicit, concise textual specification	allowed in principle; no support	component and connector types are tightly tied to its supported styles	programmable stylistic invariants; refinement maps constrain valid refinements	refinement maps enable correct refinements across styles	aided by explicit configurations; used in large-scale project	no support for partial architectures or application families; aided by explicit configurations;	none	none
UniCon	*Connect*; explicit	explicit textual and graphical specification; configuration description may be distributed	supported through composite components and connectors	supports only predefined component and connector types	players may only be attached to roles and vice versa	supports system generation; implementation constraining	aided by explicit configurations and variable number of connector roles	no support for partial architectures or application families; aided by explicit configurations;	none	none
Wright	*Attachments*; explicit	explicit, concise textual specification	allowed in principle; no support	supports both fine- and coarse-grain elements	ports can only be attached to roles and vice versa	none	aided by explicit configurations; hampered by fixed number of roles; used in large-scale project	suited for partial specification; aided by explicit configurations; no support for application families;	none	none

VIII. Tool Support for ADLs

A major impetus behind developing formal languages for architectural description is that their formality renders them suitable to manipulation by software tools. A supporting toolset that accompanies an ADL is, strictly speaking, not a part of the language. However, the usefulness of an ADL is directly related to the kinds of tools it provides to support architectural design, evolution (both static and dynamic), 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, with the exception of Rapide, they tend to focus on a single area of interest, such as analysis (e.g., Wright) or refinement (e.g., SADL). Furthermore, within these areas, ADLs tend to 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 the different languages, attempting to highlight the biggest shortcomings.

VIII.A. Active Specification

Active specification support can significantly reduce the cognitive load on software architects. Only a handful of existing ADLs provide tools that actively support specification of architectures. In general, such tools can be proactive or reactive. Proactive specification tools act in a proscriptive manner, similar to syntax-directed editors for programming languages: they limit the available design decisions based on the current state of architectural design. For example, such tools may prevent selection of components whose interfaces do not match those currently in the architecture or disallow invocation of analysis tools on incomplete architectures.

UniCon's graphical editor operates in this manner. It invokes UniCon's language processing facilities to *prevent* errors during design, rather than correct them after the fact. Furthermore, the editor limits the kinds of players and roles that can be assigned to different types of components and connectors, respectively.

Aesop provides a syntax-directed editor for specifying computational behavior of *filters*. Although no other types of components are currently supported, integration with external editors for such components is allowed. Aesop also provides a type hierarchy for visualizations of its architectural elements, where each component and connector class has an associated visualization

class. For example, the *pipe* subclass of *connector* refers to the *arrow* visualization class, which is a subclass of the more general *connector_line* class. These classes can refer to external editors, so that, e.g., a visualization class in the pipe-and-filter style invokes an editor on filter code.

Darwin's *Software Architect's Assistant* [NKM96] is another example of a proactive specification tool. The *Assistant* automatically adds services of appropriate types to components that are bound together. It also maintains the consistency of data types of connected ports: changing one port's type is automatically propagated to all ports which are bound to it. Finally, the choice of component properties during specification is constrained via dialogs.

Reactive specification tools detect *existing* errors. They may either only inform the architect of the error (*non-intrusive*) or also force him to correct it before moving on (*intrusive*). In the former case, once an inconsistency is detected, the tool informs the architect, but allows him to remedy the problem as he sees fit or ignore it altogether. The C2 design environment, *Argo*, provides non-intrusive active specification support with its design critics and to-do lists. In the latter case, the architect is forced to remedy the current problem before moving on. Certain features of MetaH's graphical editor can be characterized as intrusive. These are described below.

An ADL may provide both proactive and reactive specification support. For example, the choice of properties for the different types of MetaH components is limited during their design with menus (proactive). On the other hand, the MetaH graphical editor gives the architect full freedom to manipulate the architecture until the *Apply* button is depressed, after which any errors must be rectified before the architect may continue with the design (reactive).

VIII.B. Multiple Views

When defining an architecture, different stakeholders (e.g., architects, developers, managers, customers) may require different views of the architecture. The customers may be satisfied with a high-level, "boxes-and-lines" description, the developers may want detailed (textual) component and connector models, while the managers may require a view of the corresponding system development process.

Several ADLs support at least two views of an architecture: textual and graphical. Some of them, e.g., Aesop, Darwin, MetaH, Rapide, and UniCon, provide automated support for alternating between the views. Others, such as C2, currently do not. Aesop, MetaH, and UniCon also distinguish different types of components and connectors iconically, while C2, Darwin, and Rapide do not. Each of these ADLs allows both top-level and detailed views of composite elements.

Support for other views is sparse. Aesop allows style-specific visualizations, but currently only supports the pipe-and-filter style. C2's *Argo* design environment provides a view of the development process that corresponds to the architecture [RR96]. Darwin's *Assistant* provides a hierarchical view of the architecture which shows all the component types and the "include" relationships among them in a tree structure. Rapide allows visualization of an architecture's execution behavior by first building an executable simulation of the architecture (using the Rapide *Simulator*) and then animating its execution (using Rapide *Animation Tools*). Rapide also provides *Poset Browser*, a tool that allows viewing events generated by the simulation. Event filtering facilities can be used to view only the events of interest.

VIII.C. Analysis

Architectural descriptions are often intended to model large, distributed, concurrent systems. The ability to evaluate the properties of such systems upstream, at architectural level, can substantially lessen the cost of any errors. Given that many unnecessary details are abstracted away in architectures, this task may also be easier than at source code level. Analysis of architectures is thus the primary focus of ADL toolset developers.

The types of analyses for which an ADL is well suited depend on its underlying semantic model and, to a lesser extent, its specification features. For example, Wright, which is based on CSP, analyzes individual connectors and components attached to them for deadlocks. Aesop currently provides facilities for checking for type consistency, cycles, resource conflicts, and scheduling feasibility in its architectures. It also uses Wright's tools to analyze connectors. C2 uses critics to establish adherence to style rules and design guidelines. Darwin enables analysis of architectures by instantiating parameters and dynamic components to enact "what if" scenarios. Similarly, Rapide *Poset Browser*'s event filtering features and *Animation Tools* facilitate analysis of architectures through simulation. MetaH and UniCon both currently support schedulability analysis by specifying non-functional properties, such as criticality and priority.[11] Finally, given two architectures, SADL can establish their relative correctness with respect to a refinement map.

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.[12] Darwin, MetaH, Rapide, and UniCon also have compilers, which enable these languages to generate executable systems from architectural descriptions.

11. In the future, MetaH is also intended to support reliability and security analyses, while UniCon is open with respect to analysis tools.
12. Currently, only a subset of the textual C2 notation is parsed [MOT97].

Aesop must provide style-specific compilers that can process the style-specific formal notations used in modeling components. For example, Aesop currently provides a compiler for the pipe-and-filter style and its substyles, such as pipeline. Wright does not have a compiler, but it 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 type information, non-functional attributes, component and connector interfaces, and semantic models. In addition to this, Rapide, which supports explicit specification of other types of constraints, also provides means for their checking and enforcement. Its *Constraint Checker* analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture.

VIII.D. Refinement

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

By supporting compilation of architectural descriptions, Aesop, Darwin, MetaH, Rapide, and UniCon thus support refinement of architectural models to executable code. Darwin, MetaH, and UniCon achieve this in a manner similar to MILs: architectural components are implemented in a programming language and the architectural description serves only to ensure proper interconnection and communication among them. The drawbacks of this approach were discussed in Section VII.E. Rapide, on the other hand, provides an executable sublanguage, which contains many common programming language control structures and provides support for concurrency [Rap96]. Aesop allows both approaches: it supports compilation of components modeled in a style-specific language as well as implementation in a traditional programming language.

Only SADL and Rapide provide tool support for refinement of architectures across *multiple* levels of abstraction and specificity. SADL's support is partial. It requires manual proofs of mappings of constructs between an abstract and a concrete architectural style. However, such a proof need be performed only once, after which SADL provides a tool that checks automatically whether two architectural descriptions adhere to the mapping.[13]

Rapide's event pattern mappings ensure behavioral consistency between two architectures. Rapide maps specify the relationship between events in a concrete and an abstract architecture. Maps are compiled using *Simulator*'s compiler and then the *Constraint Checker* is used to verify that the events generated during simulation of the concrete architecture satisfy the constraints in the abstract architecture.

VIII.E. Code Generation

The ultimate goal of any software design and modeling endeavor 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, already discussed, problems of consistency and traceability between an architecture and its implementation. It is, therefore, desirable, if not imperative, for an ADL to provide source code generation tools.

A large number of ADLs, but not all, do so. Aesop provides a C++ class hierarchy for its concepts and operations, such as components, connectors, ports, roles, connecting a port to a role, and so on. This hierarchy provides a basis from which an implementation of an architecture may be produced. For example, Aesop generates C++ code for architectures in the pipe-and-filter style.

A similar approach is used in C2: we developed a framework of abstract classes for C2 concepts [MOT97]. The framework implements interconnection and message passing protocols and enables generation of top-level ("main") application routines. Components and connectors used in C2 applications are subclassed from the appropriate framework classes, allowing developers to focus on application-level issues. The framework has been implemented in C++ and Java; its subset is also available in Ada. We have been able to successfully reuse the Q interprocess communication library [MHO96] to enable message exchange between C2 components implemented in C++ and Ada.

As already discussed, Darwin, MetaH, and UniCon require preexisting component implementations in C++, Ada, and C, respectively, in order to generate applications. Rapide can construct executable systems in the same manner in C, C++, Ada, and VHDL, or it can use its executable sublanguage.

---

13. Originally, SADL's authors had planned to provide a tool that would take as its inputs an abstract architecture and a refinement map and generate a more concrete architecture, which would be correct by construction with respect to the abstract architecture.

On the other hand, SADL, ACME, and Wright are currently used strictly as modeling notations and provide no code generation support. It is interesting to note that, while SADL focuses on refining architectures, it does not take the final step from architectural descriptions to source code.

VIII.F. 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. Darwin and Rapide can model only planned modifications at runtime: both support conditional configuration; Darwin also allows component replication. Their 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. *ArchShell* supports modification of an architecture at runtime by dynamically loading and linking new architectural elements into the architecture. Furthermore, while the application is running, users can interactively send C2 requests and notifications to architectural elements. Some of *ArchShell*'s features were enabled or made easier to implement because of Java's interpreted and multi-threaded nature. However, we believe that feasibility of majority of the concepts behind the tool is independent of the underlying programming language.

VIII.G. Summary of ADL Tool Support

Existing ADLs span a broad spectrum in terms of the design and development tools they provide. On the one hand, ACME currently only facilitates visualization of its architectures, while SADL's toolset consists primarily of a refinement consistency checker. On the other hand, Darwin, Rapide, and UniCon provide powerful architecture modeling environments; Darwin is the only ADL that provides tool support in all classification categories. C2 supplies a number of tools that span all but one of the categories. However, those tools are currently not fully interoperable and have not yet been integrated into a cohesive environment. Overall, existing ADLs have put the greatest emphasis on visualization and analysis of architectures and the least on refinement and dynamism.

A more complete summary of this section is given in Table 4 below.

Table 4: ADL Tool Support

Features / *ADL*	Active Specification	Multiple Views	Analysis	Refinement	Code Generation	Dynamism
ACME	none	textual; "weblets" in *ACME-Web*; animation of pipe-and-filter architectures; architecture views in terms of high-level (template), as well as basic constructs	parser	none	none	none
Aesop	syntax-directed editor for components; visualization classes invoke specialized external editors	textual and graphical; style-specific visualizations; parallel visualization type hierarchy; component and connector types distinguished iconically	parser; style-specific compiler; type checker; cycle checker; checker for resource conflicts and scheduling feasibility	none	*build* tool constructs system glue code in C++ for pipe-and-filter style	none
C2	design critics and to-do lists in *Argo*	textual and graphical; view of development process	parser; critics to establish adherence to style rules and design heuristics	none	class framework enables generation of C/C++, Ada, and Java code	*ArchShell* allows *pure* dynamic manipulation of architectures
Darwin	automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties;	textual, graphical, and hierarchical system view	parser; compiler; "what if" scenarios by instantiating parameters and dynamic components	compiler; primitive components are implemented in a traditional programming language	compiler generates C++ code	compilation and runtime support for *constrained* dynamic change of architectures (replication and conditional configuration)
MetaH	graphical editor requires error correction once architecture changes are *applied* and constrains the choice of component properties via menus	textual and graphical; component types distinguished iconically	parser; compiler; schedulability, reliability, and security analysis	compiler; primitive components are implemented in a traditional programming language	compiler generates Ada code (C code generation planned)	none
Rapide	none	textual and graphical; visualization of execution behavior by animating simulations	parser; compiler; analysis via event filtering and animation; constraint checker to ensure valid mappings	compiler for executable sublanguage; tools to compile and verify event pattern maps during simulation	executable system construction in C/C++, Ada, VHDL, and Rapide	compilation and runtime support for *constrained* dynamic change of architectures (conditional configuration)
SADL	none	textual only	parser; analysis of relative correctness of architectures with respect to a refinement map	checker for adherence of architectures to a manually-proved mapping	none	none
UniCon	graphical editor prevents errors during design by invoking language checker	textual and graphical; component and connector types distinguished iconically	parser; compiler; schedulability analysis	compiler; primitive components are implemented in a traditional programming language	compiler generates C code	none
Wright	none	textual only; model checker provides a textual equivalent of CSP symbols	parser; model checker for type conformance of ports to roles; analysis of individual connectors for deadlock	none	none	none

IX. Conclusions

Classifying and comparing any two languages objectively is a difficult task. For example, a programming language, 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 programming languages (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 accompanying 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].

The comparison of existing ADLs highlighted several areas where they provide extensive support, both in terms of architecture modeling capabilities and tool support. For example, a number of languages use powerful formal notations for modeling component and connector semantics. They also provide a plethora of architecture visualization and analysis tools. On the other hand, the survey also pointed out areas in which existing ADLs are severely lacking. Only a handful support the specification of non-functional properties, even though such properties may be essential for system implementation and management of the corresponding development process. Architectural refinement and constraint specification have also remained largely

unexplored. Finally, both tools and notations for supporting architectural dynamism are still in their infancy. Only one ADL has even attempted to achieve *pure* dynamism thus far.

Perhaps most surprising is the inconsistency with which ADLs support connectors, especially given their argued primary role in architectural descriptions. Several ADLs provide only minimal connector modeling capabilities. Others either only allow *modeling* of complex connectors (e.g., Wright) or implementation of *simple* ones (e.g., UniCon). No existing ADL has explored the issues inherent in implementing complex connectors, possibly by employing existing research and commercial connector technologies, such as Field [Rei90], SoftBench [Cag90], Polylith [Pur94], Tooltalk [JH93], and CORBA [OHE96]. This remains a wide open research issue.

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 had to resort to heuristics and subjective criteria in comparing ADLs at times, indicating areas where future work should be concentrated. But 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 future research and architecture interchange efforts more precisely.

X. 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.

[Cag90] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, pages 36–47, June 1990.

[Cle95] P. C. Clements. Formal Methods in Describing Architectures. In *Proceedings of the Workshop on Formal Methods and Architecture*, Monterey, CA, 1995.

[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.

[Gar95a] D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.

[Gar95b] D. Garlan. An Introduction to the Aesop System. July 1995. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps

[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.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GKMM96] D. Garlan, A. Kompanek, R. Melton, and R. Monroe. Architectural Style: An Object-Oriented Approach. Submitted for publication, February 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.

[GMW96] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Interchange Language. Submitted for publication, 1996.

[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.

[HLOW94] W. D. Heym, T. J. Long, W. F. Ogden, and B. W. Weide. Mathematical Foundations and Notation of RESOLVE. Technical Report OSU-CISRC-8/93-TR45, Ohio State University, August 1994.

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

[HP93] P. Haggar and J. Purtilo. Overview of QAD, an Interface Description Language. Draft, University of Maryland, January 1993.

[HSX91] W. L. Hursch, L. M. Seiter, and C. Xiao. In any CASE: Demeter. *American Programmer*, pages 46-56, October 1991.

[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.

[JH93] A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication*. SunSoft Press/Prentice Hall, April 1993.

[JM94] F. Jahanian and A. K. Mok. Modechart: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, pages 933-947, December 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.

[KLB93] B. Kramer, Luqi, and V. Berzins. Compositional Semantics of a Real-Time Prototyping Language. *IEEE Transactions on Software Engineering*, pages 453-477, May 1993.

[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.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *IEE Software Engineering Journal*, pages 73-82, March 1993.

[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.

[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.

[MHO96] M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996. Also issued as CU Technical Report CU-CS-782-95.

[MK95] J. Magee and J. Kramer. Modelling Distributed Software Architectures. In

Proceedings of the First International Workshop on Architectures for Software Systems, pages 206-222, Seattle, WA, April 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.

[MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. To appear in *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, Boston, MA, May 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. To appear 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.

[NKM96] K. Ng, J. Kramer, and J. Magee. A CASE Tool for Software Architecture Design. *Journal of Automated Software Engineering (JASE), Special Issue on CASE-95*, 1996.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.

[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.

[Pur94] J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, pages 151-174, January 1994.

[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.

[Rap96] Rapide Design Team. Rapide 1.0 Language Reference Manual. Program Analysis and Verification Group, Computer Systems Lab, Stanford University, January 1996.

[Rei90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[RHR96] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending Design Environments to Software Architecture Design. In *Proceedings of the 1996 Knowledge-Based*

Software Engineering Conference (KBSE), pages 63-72, Syracuse, NY, September 1996.

[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.

[SDZ96] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, May 1996.

[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.

[Sha93] M. Shaw. Procedure Calls are the Assembly Language of System Interconnection: Connectors Deserve First Class Status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[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.

[TM91] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[TMA+95] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 17)*, Seattle, WA, April 1995, pages 295-304.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.

[Tra93a] W. Tracz. Parameterized Programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing (SAC'93)*, February 1993.

[Tra93b] 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.

[VHDL87] IEEE, Inc. *IEEE Standard VHDL Language Reference Manual*. IEEE Standard 1076-1987. Los Alamitos, CA, IEEE CS Press, 1987.

[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.