A Classification and Comparison Framework for Software Architecture Description Languages

Nenad Medvidovic and Richard M. Taylor

Vol. 26, No. 1
January 2000

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
A Classification and Comparison Framework for Software Architecture Description Languages

Nenad Medvidovic and Richard N. Taylor, Member, IEEE Computer Society

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 classification framework for ADLs. The utility of the definition is demonstrated by using it to differentiate ADLs from other modeling notations. The framework is used to classify and compare several existing ADLs, enabling us, in the process, to identify key properties of ADLs. The comparison highlights areas where existing ADLs provide extensive support and those in which they are deficient, suggesting a research agenda for the future.

Index Terms—Software architecture, architecture description language, component, connector, configuration, definition, classification, comparison.

1 INTRODUCTION

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family [54], [66]. 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” [71]. ADLs have recently become an area of intense research in the software architecture community [11], [16], [73], [37].

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 [14], [12], ArTek [69], C2 [39], [42], Darwin [35], [36], LILEANNA [70], MetaH [6], [72], Rapide [31], [32], SADL [46], [47], UniCon [62], [63], Weaves [20], [21], and Wright [2], [4].

Recently, initial work has been done on an architecture interchange language, ACME [15], 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 [43]. 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 example, Rapide can be viewed as both an ADL and a simulation language, while Clements contends that CODE [49], a parallel programming language, is also an ADL [8].

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

---

1. The full name of the ADL for modeling architectures in the C2 architectural style is “C2ADEL.” To distinguish it from SADL, which resulted from an unrelated project, C2ADEL will be referred to simply as “C2” in this paper.
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 [59]). 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 [47], Darwin’s use of π-calculus to formalize architectural semantics [36], or UniCon’s parser and compiler [62]). 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.

Several researchers have attempted to shed light on these issues, either by surveying what they consider existing ADLs [8], [27], [28], [71] or by listing “essential requirements” for an ADL [32], [62], [64], [65]. In our previous work, we attempted to understand and compare ADLs based on problem areas within software architectures for which they are suited [40]. 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 constituent parts.

The remainder of the paper is organized as follows: Section 2 discusses contributions and shortcomings of other attempts at surveying and classifying ADLs. Section 3 defines our taxonomy of ADLs and demonstrates its utility by determining whether several existing notations are ADLs. Section 4 assesses the above-mentioned ADLs based on the criteria established in Section 3. Discussion and conclusions round out the paper.

2 RELATED APPROACHES

Any effort such as this one is based on discoveries and conclusions of other researchers. We closely examined ADL surveys conducted by Clements and Kogut [8], [27], [28] and Vestal [71]. We also studied several researchers’ attempts at identifying essential ADL characteristics and requirements: Luckham and Vera [32], Shaw et al. [62], Shaw and Garlan [64], [65], and Tracz [70]. As a basis for architectural interchange, ACME [15] gave us key insights into what needs to remain constant across ADLs. Finally, we built upon our conclusions from earlier attempts to shed light on the nature and needs of architecture modeling [40], [42].

2.1 Previous Surveys

Clements and Kogut [8], [27], [28] 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. The survey does not provide any deeper insight into what an ADL is, nor does it present its criteria for including a particular modeling notation. Quite the contrary, several surveyed languages are not commonly 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 [26]. 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 [7]. Tool support alone is not a sufficient reason to consider it an ADL however.

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.² This kind of survey also 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 [71] is more bottom-up. He surveyed four existing ADLs (LILEANNA, MetaH, Rapide, and QAD [22]) and attempted to identify their common properties. He concluded that they all model or support the following concepts to some degree:

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

Although “ cursory” (as he qualifies it) 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 closer to Vestal’s than to Clements and Kogut’s.

In our previous work [40], we attempted to identify the problems or areas of concern that need to be addressed by ADLs:

2. A discussion of the scope of software architectures and, therefore, ADLs, is given by Perry and Wolf [54]. Their conclusions are largely mirrored in the definition of architectures given by Shaw and Garlan [66].
2.3 Attempts at Identifying Underlying Concepts

In [70], 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. Tracz’s taxonomy is similar to Perry and Wolf’s original model of software architectures, which consists of elements, form, and rationale [54]. Perry and Wolf’s elements are Tracz’s components and connectors, their form subsumes an architectural configuration, and the rationale is roughly equivalent to constraints.

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 [4], [61], [64].

In [64], they also elaborate six classes of properties that an ADL should provide: composition, abstraction, re-usability, 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 [65], 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 [43], Medvidovic et al. argue that, in order to adequately support architecture-based development and analysis, one must model architectures 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.

2.4 Architecture Interchange

Perhaps the closest the research community has come to a consensus on ADLs has been the emerging endorsement by a segment of the community of ACME as an architecture interchange language [15]. 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 could be viewed as a good 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 (e.g., a connector may not be directly attached to another connector) satisfy the needs of some approaches (e.g., Aesop, UniCon, and Wright), but not of others (e.g., C2).

3 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 collections of features and requirements shows that there is a common theme among them, which is used as a guide in formulating our 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 three International Software Architecture Workshops [11], [73], [37], were studied. To a large degree, our taxonomy reflects features supported by all, or most, existing ADLs. In certain cases, we also argue for characteristics typically not supported by current ADLs, but which have either been identified in the literature as important for architecture-based development or have resulted from our experience with our own research project in software architectures, C2. Finally, we have tried to learn from and, where relevant, apply the extensive experience with languages for modeling other aspects of software in formulating our framework.

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 Shaw and Garlan [66]:

Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

An ADL is thus a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures [15]. The conceptual framework typically 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).

3.1 Framework Categories

We introduce the top-level categories of our ADL classification and comparison framework in this section. The building blocks of an architectural description are: 1) components, 2) connectors, and 3) architectural configurations. 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, similar to a "boxes and lines" diagram with no explicit underlying semantics. Several other aspects of components, connectors, and configurations are desirable, but not essential: Their benefits have been acknowledged and, possibly, demonstrated in the context of a problem domain or a style, but their absence does not mean that a given language is not an ADL.

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. Conversely, the desired manipulations of architectural models by tools may influence the modeling features provided in an ADL. A large segment of the ADL research community is actively studying the issue of tool support; an effort to identify a canonical "ADL toolkit" is currently under way [17].

The ADL classification and comparison framework is depicted in Fig. 1. It is intended to be extensible and modifiable, which is crucial in a field that is still largely in its infancy. The remainder of this section motivates and further elaborates on each category of the framework.

The categories identified in the framework are orthogonal to an ADL's scope of applicability. As a model of a system at a high level of abstraction, an ADL is intended (and can only be expected) to provide a partial depiction of the system. The types of information on which the ADL focuses may be the characteristics of an application domain, a style of system composition (i.e., an architectural style), or a specific set of properties (e.g., distribution, concurrency, safety, and so on). Regardless of the focus and nature of the ADL, in general the desired kinds of representation, manipulation, and qualities of architectural models described in the ADL, and identified in Fig. 1, remain constant.

3.1.1 Modeling Components

A component in an architecture is a unit of computation or a data store. Therefore, components are loci of computation and state [62]. Components may be as small as a single procedure or as large as an entire application. Each component may

3. "Architectural configurations" will, at various times in this paper, be referred to simply as "configurations" or "topologies."
require its own data or execution space, or it may share them with other components. As already discussed, explicit component interfaces are a feature required of ADLs. Additional comparison features are those for modeling component types, semantics, constraints, evolution, and non-functional properties. Each is discussed below:

**Interface**—A component’s interface is a set of interaction points between it and the external world. The interface specifies the services (messages, operations, and variables) a component provides. In order to support reasoning about a component and the architecture that includes it, ADLs may 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.

**Types**—Component types are abstractions that encapsulate functionality into reusable blocks. A component type can be instantiated multiple times in a single architecture or it may be reused across architectures. Component types can be parameterized, further facilitating reuse. Explicit modeling of types also aids understandability and analyzability of an architecture in that the properties of a type are shared by all of its instances.

**Semantics**—We define component semantics as a high-level model of a component’s behavior. Such a model is needed to perform analysis, enforce architectural constraints, and ensure consistent mappings of architectures from one level of abstraction to another. Note that a component’s interface also allows a certain, limited degree of reasoning about its semantics. However, the notion of semantics used in this paper refers strictly to models of component behavior.

**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 (or less desirable) to one or more stakeholders [9]. In order to ensure adherence to intended component uses, enforce usage boundaries, and establish dependencies among internal parts of a component, constraints on them must be specified.

**Evolution**—As architectural building blocks, components will continuously evolve. Component evolution can be informally defined as the modification of (a subset of) a component’s properties, e.g., interface, behavior, or implementation. ADLs can ensure that evolution happens in a systematic manner by employing techniques such as subtyping of component types and refinement of component features.

**Nonfunctional Properties**—A component’s nonfunctional properties (e.g., safety, security, performance, portability) typically cannot be directly derived from the specification of its behavior. Early specification of such properties (at the architectural level) is needed to enable simulation of runtime behavior, perform analysis, enforce constraints, map component implementations to processors, and aid in project management.

### 3.1.2 Modeling 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, 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 forth [15], [62]. The features characterizing connectors are their interfaces, types, semantics, constraints, evolution, and nonfunctional properties. Each is defined and motivated below:

**Interface**—A connector’s interface is a set of interaction points between the connector and the components and other connectors attached to it. Since a connector does not perform any application-specific computations, it exports as its interface those services it expects of its attached components. Connector interfaces enable proper connectivity of components and their interaction in an architecture and, thereby, reasoning about architectural configurations.

**Types**—Connector types are abstractions that encapsulate component communication, coordination, and mediation decisions. Architecture-level interactions may be characterized by complex protocols. Making these protocols reusable both within and across architectures requires that ADLs model connectors as types. This is typically done in two ways: as extensible type systems, defined in terms of

4. Although the comparison categories for components and connectors are identical, they were derived and refined independently of each other.
interaction protocols, or as built-in, enumerated types, based on particular implementation mechanisms.

**Semantics**—Similarly to components, connector semantics is defined as a high-level model of a connector’s behavior. Unlike components, whose semantics express application-level functionality, connector semantics entail specifications of (computation-independent) interaction protocols. ADLs may support modeling of connector semantics in order to enable component interaction analysis, consistent refinement of architectures across levels of abstraction, and enforcement of interconnection and communication constraints.

**Constraints**—Connector constraints ensure adherence to intended interaction protocols, establish intraconnector dependencies, and enforce usage boundaries. An example of a simple and easily enforceable constraint is a restriction on the number of components that interact through the connector. Establishing adherence to more complex connector constraints (e.g., minimal throughput) may require access to information external to the given connector (e.g., a model of the attached components’ dynamic semantics).

**Evolution**—Analogously to component evolution, the evolution of a connector is defined as the modification of (a subset of) its properties, e.g., interface, semantics, or constraints on the two. Component interactions in architectures are governed by complex and potentially changing expanding protocols. Furthermore, both individual components and their configurations evolve. ADLs can accommodate this evolution by modifying or refining existing connectors with techniques such as incremental information filtering, subtyping, and refinement.

**Nonfunctional Properties**—A connector’s nonfunctional properties are not entirely derivable from the specification of its semantics. They represent (additional) requirements for correct connector implementation. Modeling nonfunctional properties of connectors enables simulation of runtime behavior, analysis of connectors, constraint enforcement, and selection of appropriate off-the-shelf (OTS) connectors (e.g., message buses) and their mappings to processors.

### 3.1.3 Modeling 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., direct communication links between components hamper evolvability of an architecture) and architectural style constraints (e.g., direct communication links between components are disallowed).

Characteristic features at the level of architectural configurations fall in three general categories:

- qualities of the configuration description: understandability, compositionality, refinement and traceability, and heterogeneity;
- qualities of the described system: heterogeneity, scalability, evolvability, and dynamism;
- properties of the described system: dynamism, constraints, and nonfunctional properties.\(^5\)

Note that the three categories are not entirely orthogonal. Heterogeneity and dynamism each appear in two categories. Heterogeneity may be manifested in multiple employed formalisms in configuration descriptions and multiple programming languages in system implementations. Anticipated dynamism is a system property in that the system may be architected specifically to accommodate the (expected) dynamic change; unanticipated dynamism is a quality that refers to a system’s general suitability for dynamic change.

The differences between the two pairs of features are subtle, particularly in the case of dynamism. While keeping the above categorization in mind, in order to maintain the conceptual simplicity of our framework and avoid confusion, we proceed by describing individual features; we include both notions of heterogeneity and dynamism under single respective headings. We motivate and, where appropriate, define the configuration features below:

**Understandable Specifications**—One role of software architecture is to serve as an early communication conduit for different stakeholders in a project and facilitate understanding of (families of) systems at a high level of abstraction. ADLs must thus 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.

**Compositionality**—Compositionality, or hierarchical composition, is a mechanism that allows architectures to describe software systems at different levels of detail: Complex structure and behavior may be explicitly represented or they may be abstracted away into a single component or connector. Situations may also arise in which an entire architecture becomes a single component in another, larger architecture. Such abstraction mechanisms should be provided as part of an ADLs modeling capabilities.

**Refinement and Traceability**—In addition to providing architects with semantically elaborate facilities for specifying architectures, ADLs must also enable correct and consistent refinement of architectures into executable systems and traceability of changes across levels of architectural refinement. This view is supported by the prevailing argument for developing and using ADLs: They are necessary to bridge the gap between informal, “boxes and lines” diagrams and programming languages which are deemed too low-level for application design activities.

**Heterogeneity**—A goal of software architectures is to facilitate development of large-scale systems, preferably

---

5. The term “quality” is used in the conventional, application-independent manner, e.g., as defined by Ghezzi et al. [18]. The term “property” refers to the characteristics of an application introduced to address specific requirements.
with preexisting components and connectors of varying granularity, possibly specified in different formal modeling languages and implemented in different programming languages, with varying operating system requirements, and supporting different communication protocols. It is therefore important that ADLs be open, i.e., that they provide facilities for architectural specification and development with heterogeneous components and connectors.

**Scalability**—Architectures are intended to provide developers with abstractions needed to cope with the issues of software complexity and size. ADLs must therefore directly support specification and development of large-scale systems that are likely to grow further.

**Evolvability**—New software systems rarely provide entirely unprecedented functionality, but are, rather, variations on a theme. An architecture evolves to reflect and enable evolution of a family of software systems. Since evolution (i.e., maintenance) is the single costliest software development activity [18], system evolvability becomes a key aspect of architecture-based development. ADLs need to augment evolution support at the level of components and connectors with features for their incremental addition, removal, replacement, and reconnection in a configuration.

**Dynamism**—Evolution, as we define it, refers to “offline” changes to an architecture (and the resulting system). Dynamism, on the other hand, refers to modifying the architecture and enacting those modifications in the system while the system is executing. Support for dynamism is important in the case of certain safety- and mission-critical systems, such as air traffic control, telephone switching, and high availability public information systems. Shutting down and restarting such systems for upgrades may incur unacceptable delays, increased cost, and risk [52]. To support architecture-based run-time evolution, ADLs need to provide specific features for modeling dynamic changes and techniques for effecting them in the running system.

**Constraints**—Constraints that depict dependencies in a configuration complement those specific to individual components and connectors. Many global constraints are derived from or directly depend upon local constraints. For example, constraints on valid configurations may be expressed as interaction constraints among constituent 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; safety of an architecture is a function of the safety of its constituents.

**Nonfunctional Properties**—Certain nonfunctional properties are system-level, rather than individual component or connector properties. Configuration-level nonfunctional properties are needed to select appropriate components and connectors, perform analysis, enforce constraints, map architectural building blocks to processors, and aid in project management.

### 3.1.4 Tool Support for Architectural Description

The motivation behind developing formal languages for architectural description is that their formality renders them suitable for reasoning and 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, analysis, evolution, executable system generation, and so forth. The importance of architectural tools is reflected in the on-going effort by a large segment of the community to identify the components that comprise a canonical “ADL toolkit” [17]. Although the results of this work are still preliminary, several general categories have emerged. They reflect the kinds of tool support commonly provided by existing architectural approaches: active specification, multiple views, analysis, refinement, implementation generation, and dynamism. Each is discussed below:

**Active Specification**—ADL tools provide active specification support by reducing the space of possible design options based on the current state of the architecture. Such tools provide design guidance and can significantly reduce a software architect’s cognitive load. They can be either proactive, by suggesting courses of action or disallowing design options that may result in undesirable design states, or reactive, by informing the architect of such states once they are reached during design. Active specification tools can deliver their feedback intrusively, forcing the architect to acknowledge it before continuing, or nonintrusively, allowing the architect to view the feedback at his discretion.

**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 (formal) component and connector specifications, while the managers may require a view of the corresponding system development process. Providing the most appropriate view to a given stakeholder and ensuring inter-view consistency are key issues to be addressed by an ADL toolkit.

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

**Refinement**—The importance of supporting refinement of architectures across levels of detail was briefly argued above and more extensively by Garlan [13] and Moriconi et al. [47]. Refining architectural descriptions is a complex task whose correctness and consistency cannot always be guaranteed by formal proof, but adequate tool support can give architects increased confidence in this respect.

**Implementation Generation**—The ultimate goal of any software design and modeling endeavor is to produce the executable system. An elegant architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. It is therefore desirable, if not imperative, for an ADL toolkit to provide tools to assist in producing source code.
Dynamism—We have argued for the need to model dynamic changes at the level of architecture. However, an ADL’s ability to model dynamic changes is insufficient to guarantee that they will be applied to the executing system in a property-preserving manner. Software tools are needed to analyze the modified architecture to ensure its desirable properties, correctly map the changes expressed in terms of architectural constructs to the implementation modules, ensure continuous execution of the application’s vital subsystems and preservation of state during the modification, and analyze and test the modified application while it is executing.

3.2 Differentiating ADLs from Other Languages

In order to clarify what an ADL is, 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 sometimes 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 items in 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 [55], programming languages, and OO notations and languages (e.g., Unified Modeling Language, or UML [57], [58]). MILs typically describe the uses relationships among modules in an implemented system and support only one type of connection [4], [64]. 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 [34].

It is important to note that there is a less than 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., Weaves, 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.

We have also recently shown that an OO language, such as UML, can be used to model software architectures if it supports certain extensions [41], [60]. These extensions are used to represent architectural abstractions that either differ (e.g., topological constraints) or do not exist (e.g., connectors) in OO design. Extending UML in such a manner is clearly useful in that it supports mapping of an architecture to a more familiar and widely used notation, therefore facilitating broader understanding of the architecture and enabling more extensive tool support for manipulating it. However, it is unrealistic to expect that UML could be extended to model every feature of every ADL; our initial experience indeed confirms this [60]. Moreover, although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides will not match an architect’s mental model of the system as faithfully as the architect’s ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices [65]. We believe this to be a key issue and one that argues against considering a notation like UML an ADL: A given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect’s job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

An ADL typically subsumes a formal semantic theory. That theory is part of the 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 Statecharts [23], partially-ordered event sets [33], communicating sequential processes (CSP) [24], model-based formalisms (e.g., chemical abstract machine, or CHAM [25], Z [67]), algebraic formalisms (e.g., Obj [19]), and axiomatic formalisms (e.g., Anna [30]). Of the above-mentioned formal notations, Z has been demonstrated appropriate for modeling only certain aspects of architectures, such as architectural style rules [1], [42]. 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 two notations, Statecharts and CHAM, are somewhat similar to those of ADLs. Although they do not express systems in terms of components, connectors, and configurations per se, their
perspective. Harel [23] agrees with this view, arguing that
down of a system can be achieved from a state-machine
charts, it is doubtful that an adequate architectural break-
represent component or connector semantics with State-
architectures as FSMs: Although it may be useful to
fore, Statecharts is not an ADL.
determined by studying its constituent components. There-
ªarchitectureº described as a Statechart can only be
case with LILEANNA and ArTek, the topology of an

Statecharts is a modeling formalism based on finite state
machines (FSM) that provides a state encapsulation
construct, support for concurrency, and broadcast commu-
nication. To compare Statecharts to an ADL, the states are
viewed as components, transitions among them as simple
connectors, and their interconnections as configurations. However, Statecharts does not model architectural config-
urations 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
determined by studying its constituent components. There-
fore, Statecharts is not an ADL.

There is an even deeper issue in attempting to model
architectures as FSMs: Although it may be useful to
represent component or connector semantics with State-
charts, it is doubtful that an adequate architectural break-
down of a system can be achieved from a state-machine
perspective. Harel [23] agrees with this view, 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...
Statecharts can then be used to control these internal activities...
We assume that this kind of description is given or can be produced
using an existing method.

3.2.2 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.

4 COMPARISON OF ADLs

This section presents a detailed comparison of existing
ADLs along the dimensions discussed in Section 3.1. We
highlight representative approaches and support our argu-
ments with example ADL specifications. The chosen
to provide various samples instead of a
single representative example is motivated by the the
inability of the research community to identify a model
problem for which all ADLs are likely to be well-suited [68].
Thus, selecting any one candidate problem would likely
draw the (justified) criticism of focusing on the strengths of
only certain languages. This point is related to the
discussion from Section 3: Different ADLs focus on different
application domains, architectural styles, or aspects of the
architectures they model. This is certainly the case with the
ADLs we have studied and which represent a large cross-
section of existing work in the area, as shown in Table 1.

4.1 ADL Support for Modeling Components

Each surveyed ADL models components. 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; in Weaves, tool fragments; and, in
MetaH, processes. In this section, we discuss the support
provided by ADLs for different aspects of components.

4.1.1 Interface

All surveyed ADLs support specification of component
interfaces. They differ in the terminology and the kinds of
information they specify. For example, an interface point in
SADL or Wright is a port and, in UniCon, a player. On the
other hand, in C2, the entire interface is provided through a
single port; individual interface elements are messages.
Weaves combines the two approaches by allowing multiple
component ports, each of which can participate in the
exchange of interface elements, or objects.

ADLs typically distinguish between interface points that
refer to provided and required functionality. MetaH and
Rapide make the additional distinction between synchro-
nous and asynchronous interfaces. For example, provides
and requires interface constituents in Rapide refer to
functions and specify synchronous communication, while
in and out actions denote asynchronous events.

<table>
<thead>
<tr>
<th>ADL</th>
<th>ACME</th>
<th>Aesop</th>
<th>C2</th>
<th>Darwin</th>
<th>MetaH</th>
<th>Rapide</th>
<th>SADL</th>
<th>UniCon</th>
<th>Weaves</th>
<th>Wright</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus</td>
<td>Architectural interchange, predominantly at the structural level</td>
<td>Specification of architectures in specific styles</td>
<td>Architectures of highly-distributed, evolvable, and dynamic systems</td>
<td>Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings</td>
<td>Architectures in the guidance, navigation, and control (GN&amp;C) domain</td>
<td>Modeling and simulation of the dynamic behavior described by an architecture</td>
<td>Formal refinement of architectures across levels of detail</td>
<td>Glue code generation for interconnecting existing components using common interaction protocols</td>
<td>Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing</td>
<td>Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems</td>
</tr>
</tbody>
</table>

TABLE 1
ADL Scope and Applicability
PRODUCTION input IS StreamIn
MAXASSOCs (1)
MINASSOCs (1)
SIGNATURE (*line*)
PORTBINDING (stdin)
END input

Fig. 2. Specification of a component player in UniCon.

```
port DataRead = get → DataRead □ √
```

Fig. 3. Interaction protocol for a component port in Wright: → denotes event transitions, √ a successfully terminating process, □ nondeterministic choice, and | deterministic choice.

Interface points are typed in a number of ADLs: ACME, Aesop, Darwin, MetaH, SADL, and UniCon. UniCon supports a predefined set of common player types, including RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, ReadFile, WriteFile, RPCDef, and RPCCall. On the other hand, ports in C2 and Weaves are type-indifferent in order to maximize the flexibility of interconnection. Weaves ports perform wrapping and unwrapping of data objects by means of envelopes, which hide the types of the underlying data objects, while C2 ports are designed to handle any C2 messages.

Finally, Wright and UniCon allow specification of expected component behavior or constraints on component usage relevant to each point of interaction. For example, UniCon allows specification of the number of associations in which a player can be involved. Fig. 2 depicts the constraint that the input player of the StreamIn type is bound to standard input and participates in exactly one association in a given architecture.

Wright specifies the protocol of interaction at each port in CSP [24]. In the example given in Fig. 3, DataRead is a simple input (read only) port.6

### 4.1.2 Types

All of the surveyed ADLs distinguish component types from instances. Rapide does so with the help of a separate types language [31]. Weaves distinguishes between sockets and tool fragments that populate them. 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. 7 Component types supported by UniCon are: Module, Computation, SharedData, SeqFile, Filter, Process, SchadProcess, and General.

Several ADLs (ACME, Darwin, Rapide, SADL, and Wright) make explicit use of parameterization of component interface signatures. This is typically done in the manner similar to programming languages such as Ada and C++. Rapide and Wright also allow the behavior associated with a particular type to be parameterized. Rapide does so by specifying event patterns, discussed below. Wright

### 4.1.3 Semantics

All ADLs support specification of component semantics, although to varying degrees. The ADLs’ underlying semantic models range from expressing semantic information in component property lists (UniCon) to models of dynamic component behavior (Rapide and Wright). 8 Other points along this spectrum are arbitrarily complex behavioral specifications that are treated as uninterpreted annotations (ACME); an accompanying language for modeling algorithms in the ADL’s domain (MetaH); specification of static component semantics via invariants and operation pre- and post-conditions (C2); and models of interaction and composition properties of composite components expressed in the calculus (Darwin).

Rapide introduces a unique mechanism for expressing both a component’s behavior and its interaction with other components: 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 poset. Event patterns are used both as triggers and outputs of component state transitions. Fig. 4 shows an example of a simple Rapide component with a causal relationship between events: When the Application component observes a Receive event, it generates a Results event in response; the two events have the same string parameter.

### 4.1.4 Constraints

All ADLs constrain the usage of a component by specifying its interface as the only legal means of interaction. Formal specification of component semantics further specifies relationships and dependencies among internal elements of a component. Several additional means for constraining components are common.

A number of ADLs provide stylistic invariants (Aesop, C2, SADL, and Wright). An example stylistic invariant is C2’s requirement that a component have exactly two communication ports, one each on its top and bottom sides. A component can also be constrained via attributes. Fig. 2 shows how a UniCon component is constrained by restricting the number of associations in which its players can participate. MetaH also constrains the implementation and usage of a component by specifying its (nonfunctional) attributes, such as: ExecutionTime, Deadline, and Criticality. Finally, Rapide enables specification of pattern constraints

6. In all examples, we adhere to each ADL’s presentation conventions (naming, capitalization, highlighting, etc.).

7. 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.

8. As discussed in the preceding section, Wright uses CSP to specify a component’s computation.
possible subtyping relationships: Well_1

configurations, their true focus. Therefore, we will defer architectures. Refinement of components and connectors order to prove or demonstrate certain properties of which place additional constraints on refinement maps in This is, however, not true of Rapide and SADL, both of subtyping is simply a form of refinement in a general case. deferred design decisions, which is somewhat similar to may be used to evolve components by explicating any components across levels of abstraction. This mechanism must subtyping relationship mandates that it but may change its interface and implementation; Matrix, possibly extends) the behavior of the component on event posets that are generated and observed from a component’s interface. In the example shown in Fig. 5, the constraint implies that all, and only, messages taken in by the Resource component are delivered.

4.1.5 Evolution

A number of ADLs view and model components as inherently static. For example, MetaH and UniCon define component types by enumeration, allowing no subtyping, and hence no evolution support; Weaves considers tool fragment evolution outside its scope. Several ADLs support component evolution via subtyping. They typically support a limited notion of subtyping or rely on the mechanisms provided by the underlying programming language. For example, ACME supports strictly structural subtyping with its `extends` feature, while Rapide evolves components via OO inheritance. SADL allows the specification of high-level properties that must be satisfied by subtypes: The example in Fig. 6 specifies that Local_Client is the subtype of Client such that all of its instances satisfy the predicate Local.

Aesop and C2 provide more extensive component subtyping support. Aesop enforces behavior-preserving subtyping to create styles of a given architectural style. An Aesop subclass must provide strict subtyping behavior for operations that succeed, but may also introduce additional sources of failure with respect to its superclass. C2, on the other hand, supports multiple subtyping relationships among components: `name`, `interface`, `behavior`, and `implementation` [39], [42]. Different combinations of these relationships are specified using the keywords `and` and `not`. Fig. 7 demonstrates two possible subtyping relationships: Well_1 preserves (and possibly extends) the behavior of the component Matrix, but may change its interface and implementation; Well_2’s subtyping relationship mandates that it must alter Matrix’s interface.

Rapide and SADL also provide features for refining 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. Indeed, subtyping is simply a form of refinement in a general case. This is, however, not true of Rapide and SADL, both of which place additional constraints on refinement maps in order to prove or demonstrate certain properties of architectures. Refinement of components and connectors in Rapide and SADL is a byproduct of the refinement of configurations, their true focus. Therefore, we will defer further discussion of this issue until Section 4.3.

4.1.6 Nonfunctional Properties

Despite the need for and benefits of specifying nonfunctional properties, there is a notable lack of support for them in existing ADLs. ACME, Aesop, and Weaves allow specification of arbitrary component properties and/or annotations. However, none of them interprets such properties nor do they make direct use of them.

MetaH and UniCon provide more advanced support for modeling nonfunctional properties. They require such information to analyze architecture for real-time schedulability (both ADLs) and reliability and security (MetaH). Both also use source code location attributes for implementation generation. Several representative nonfunctional 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`.

4.1.7 Summary of ADL Components

Overall, surveyed ADLs provide comprehensive support for modeling components. All of them regard components as first-class entities. Furthermore, all models interface and distinguish between component types and instances. On the other hand, a majority of the ADLs do not support evolution or nonfunctional 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 five ADLs that support five of the categories, C2 and Rapide do not model nonfunctional properties, and MetaH, UniCon, and Weaves 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 2.

4.2 ADL Support for Modeling Connectors

ADLs model connectors in various forms and under various names. For example, ACME, Aesop, C2, SADL, UniCon, and Wright model connectors explicitly and refer to them as connectors. Weaves also models connectors explicitly, but refers to them as transport services. Rapide and MetaH connections and Darwin bindings are modeled in-line, and cannot be named, subtyped, or reused (i.e., connectors are not first-class entities). Darwin and Rapide do allow abstracting away complex connection behaviors into “connector components.” In this section, we compare existing ADLs with respect to the support they provide for different aspects of connectors.

4.2.1 Interface

In general, only the ADLs that model connectors as first-class entities support explicit specification of connector interfaces. Most such ADLs model component and connector interfaces in the same manner, but refer to them differently. Thus, connector interface points in
ACME, Aesop, UniCon, and Wright are roles, which are named and typed. Explicit connection of component ports (players in UniCon) and connector roles is required in an architectural configuration. Wright supports CSP specifications of each role's interaction protocol in the same manner as port protocols (see Fig. 3). This allows compatibility analysis of connected ports and 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 supports only a predefined set of role types, including: Source, Sink, Reader, Readee, Writer, Writee, Definer, and Caller. An example UniCon role is shown in Fig. 8. It belongs to the Pipe connector type and is constrained to be connected to at most a single player. Note that, unlike the player shown in Fig. 2, which must participate in exactly one association, this role does not have to be connected to a player.

SADL, C2, and Weaves model connector interfaces differently from component interfaces. A SADL connector is defined as part of the design vocabulary for a particular architectural style. The specification of the connector in an architecture only specifies the type of data the connector supports (e.g., the connector declared in Fig. 9a expects a token sequence). Other information about the connector, such as its arity and the constraints on its usage, is given in the definition of its style (Fig. 9b).

The interfaces of C2 and Weaves connectors are generic: The connectors are indifferent to the types of data they handle; their main task is to mediate and coordinate the communication among components. Additionally, a C2 connector can support an arbitrary number of components. In C2, this feature is referred to as context-reflection: The interface of a connector is determined by (potentially dynamic) interfaces of components that communicate through it, as depicted in Fig. 10.

### 4.2.2 Types

Only ADLs that model connectors as first-class entities distinguish connector types from instances. This excludes Darwin, MetaH, and Rapide. Although MetaH does not support connector types, it does define three broad categories of connections: port connections, which connect an out port of one component to an in port of another; event connections, which connect outgoing events to incoming events (event-to-event) or to their recipient components (event-to-process and
ACME, Aesop, C2, SADL, and Wright base connector types on interaction protocols. UniCon, on the other hand, only allows connectors of prespecified enumerated types: Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcCall, and RTScheduler. ACME and SADL also provide parameterization facilities that enable flexible specification of connector signatures and of constraints on connector semantics. Similarly to its components, Wright allows a connector to be parameterized by the specification of its behavior (glue).

**4.2.3 Semantics**

It is interesting to note that ADLs that do not model connectors as first-class objects, e.g., Rapide, may model connector semantics, while languages that do model connectors explicitly, such as ACME, do not always provide means for defining their semantics. ADLs tend to use a single mechanism for specifying the semantics of both components and connectors. For example, Rapide uses posets to describe communication patterns among its components; Wright models connector glue and event trace specifications with CSP, as shown in Fig. 11; 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 are implicit in their (predefined) connector types. For example, declaring a connector to be a pipe implies certain functional properties.

Several ADLs use a different semantic model for their connectors than for components. For example, as demonstrated in Fig. 9, SADL provides a constraint language for specifying style-specific connector semantics. C2 models a connector’s message filtering policy: message_sink, no_filtering, message_filtering, and prioritized. Finally, Weaves employs a set of naming conventions that imply its transport services’ semantics. For example, a single-reader, single-writer queue transport service is named Queue_1_1.

**4.2.4 Constraints**

With the exception of C2 and Weaves, whose connector interfaces are a function of their attached components (see Section 4.2.1), 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. Aesop, C2, SADL, and Wright also impose stylistic invariants, such as C2’s restriction that each connector port may only be attached to a single other port. UniCon restricts the number of component players attached to a connector role by using the MinConns and MaxConns attributes. Additionally, the types of players that can serve in a given role are constrained in UniCon via the Accept attribute and in Wright by specifying interaction protocols for the role (see Fig. 11). For example, the output UniCon role from Fig. 8 can be constrained to accept the StreamIn player of the Filter component type (see Fig. 12).

**4.2.5 Evolution**

ADLs that do not model connectors as first-class objects (Darwin, MetaH, and Rapide) also provide no facilities for their evolution. Others focus on configuration-level evolution (Weaves) or provide a predefined set of connector types with no language features for evolution support (UniCon).

Several ADLs employ identical mechanisms for connector and component evolution: ACME supports structural connector subtyping, Aesop supports behavior preserving subtyping, and SADL supports subtyping of connectors and their refinements across styles and levels of abstraction. C2 connectors are inherently evolvable because of their context-reflective interfaces; C2 connectors also evolve by altering their filtering policies. Finally, Wright supports
connector evolution via parameterization, where, e.g., the same connector can be instantiated with a different glue.

4.2.6 Nonfunctional Properties

UniCon is the lone ADL that supports explicit specification of nonfunctional connector properties, using 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 with their components, ACME, Aesop, and Weaves allow specification of arbitrary, but uninterpreted connector annotations.

4.2.7 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 3.

All ADLs that model connectors explicitly also model their interfaces and distinguish connector types from instances. It is

<table>
<thead>
<tr>
<th>Features</th>
<th>Characteristics</th>
<th>Interface</th>
<th>Types</th>
<th>Semantics</th>
<th>Constraints</th>
<th>Evolution</th>
<th>Non-Functional Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>Connector; explicit</td>
<td>interface points are roles</td>
<td>extensible type system, based on protocols; parameterization via templates</td>
<td>no support; can use other ADLs’ semantic models in property lists</td>
<td>via interfaces and structural for type instances</td>
<td>structural subtyping via the extends feature</td>
<td>allows any attribute in property lists, but does not operate on them</td>
</tr>
<tr>
<td>Aesop</td>
<td>Connector; explicit</td>
<td>interface points are roles</td>
<td>extensible type system, based on protocols</td>
<td>(optional) semantics specified using Wright</td>
<td>via interfaces and semantics; stylistic invariants</td>
<td>behavior-preserving subtyping</td>
<td>allows association of arbitrary text with connectors</td>
</tr>
<tr>
<td>C2</td>
<td>Connector; explicit</td>
<td>interface with each component via a separate port; interface elements are provided and required</td>
<td>extensible type system, based on protocols</td>
<td>partial semantics specified via message filters</td>
<td>via semantics; stylistic invariants (each port participates in one link only)</td>
<td>context-reflective interfaces; evolvable filtering mechanisms</td>
<td>none</td>
</tr>
<tr>
<td>Darwin</td>
<td>Binding; in-line; no explicit modeling of component interactions</td>
<td>none; allows “connection components”</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>MetaH</td>
<td>Connection; in-line; allows connections to be optionally named</td>
<td>none; supports three general classes of connections: port, event, and equivalence</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Rapide</td>
<td>Connection; in-line; complex reusable connectors only via “connection components”</td>
<td>none; allows “connection components”</td>
<td>none; po-sets; conditional connections</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>SADL</td>
<td>Connector; explicit</td>
<td>connector signature specifies the supported data types</td>
<td>extensible type system; parameterized signatures and constraints</td>
<td>axioms in the constraint language</td>
<td>via interfaces; stylistic invariants</td>
<td>subtyping; connector refinement via pattern maps</td>
<td>requires connector modification (see Section 4.3.9)</td>
</tr>
<tr>
<td>UniCon</td>
<td>Connector; explicit</td>
<td>interface points are roles</td>
<td>predefined, enumerated set of types</td>
<td>implicit in connector’s type; semantic information can be given in property lists</td>
<td>via interfaces; restricts the type of players that can be used in a given role</td>
<td>none</td>
<td>attributes for schedulability analysis</td>
</tr>
<tr>
<td>Weaves</td>
<td>Transport services; explicit</td>
<td>interface points are the encapsulating socket pads</td>
<td>extensible type system; types are connector sockets</td>
<td>via naming conventions</td>
<td>via interface</td>
<td>none</td>
<td>allows association of arbitrary, uninterpreted annotations with transport services</td>
</tr>
<tr>
<td>Wright</td>
<td>Connector; explicit</td>
<td>interface points are roles; role interaction semantics specified in CSP</td>
<td>extensible type system, based on protocols; parameterizable number of roles and glue</td>
<td>connector glue semantics in CSP</td>
<td>via interfaces and semantics; protocols of interaction for each role in CSP; stylistic invariants</td>
<td>via different parameter instantiations</td>
<td>none</td>
</tr>
</tbody>
</table>
interesting to note that, as in the case of components, support for evolution and nonfunctional 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 3.

4.3 ADL Support for Modeling Configurations

Explicit architectural configuration facilitates communication among a system’s many stakeholders, who are likely to have various levels of technical expertise and familiarity with the problem at hand. This is accomplished by abstracting away the details of individual components and connectors and representing the system’s structure at a high level. In this section, we discuss the key aspects of explicit configurations and compare surveyed ADLs with respect to them.

4.3.1 Understandable Specifications

Configuration descriptions in in-line configuration ADLs (e.g., Rapide) tend to be encumbered with connector details. On the other hand, explicit configuration ADLs (e.g., 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 instantiations, as shown in Fig. 13.

Several languages provide a graphical notation as another means of achieving understandability. An example of an architecture modeled using C2’s graphical notation was shown in Fig. 10. A graphical architectural description may actually hinder understanding unless there is a precise relationship between it and the underlying model, i.e., unless the textual and graphical descriptions are interchangeable. Languages like C2, Darwin, and UniCon support such “semantically sound” graphical notations, while ACME, SADL, and Wright do not.\(^9\)

4.3.2 Compositionality

Most ADLs provide explicit features to support hierarchical composition of components, where the syntax for specifying composite components typically resembles that for specifying configurations. Wright allows both composite components and connectors: The computation (glue) of a composite component (connector) is represented by an architectural description, rather than in CSP. It is interesting to note that Darwin and UniCon do not have explicit constructs for modeling architectures. Instead, they both model architectures as composite components. The statement sequence shown in Fig. 13 occurs inside the specification of a UniCon composite component. An example of a Darwin component illustrating its support for compositionality is shown in Fig. 15.

4.3.3 Refinement and Traceability

Architectural refinement and traceability of architectural decisions, properties, and relationships across refinements is still very much an open research area. Support for them in existing ADLs is limited. Several ADLs enable system generation directly from an architectural specification. These are typically the implementation constraining languages (see Section 3) in which a source file corresponds to each architectural element. There are several problems with this approach to refining an architecture. Primarily, there is an assumption that the relationship between elements of an architectural description and those of the resulting executable system will be one-to-one. This may be unnecessary, and even unreasonable, as architectures describe systems at

\(^9\) 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 support is needed to make the two truly interchangeable (see Section 4.4.2).
a higher level of abstraction than source code modules. There is also no guarantee that the specified source modules will correctly implement the desired behavior. Finally, even if the specified modules currently implement the needed behavior correctly, this approach provides no means of ensuring that future changes to those modules are traced back to the architecture and vice versa.

SADL and Rapide support refinement and traceability more extensively. They provide maps for refining architectures across different levels of abstraction. SADL uses its maps (see Fig. 14) to support the correctness of architectural refinements. While Rapide generates comparative simulations of architectures at different levels. Both languages thus provide the means for tracing design decisions and changes from one level of architectural specification (or implementation) to another. They enforce different refinement rules, however: SADL’s stringent correctness-preserving criterion ensures that all decisions made at a given level are maintained at all subsequent levels, but disallows new decisions being introduced; Rapide’s maps allow new decisions, but may also eliminate high-level behaviors at the lower levels. Garlan has recently argued for a marriage of the two approaches [13].

4.3.4 Heterogeneity

No ADL provides explicit support for multiple formal specification languages. Of those ADLs that support implementation of architectures, several are also tightly tied to a particular programming language. For example, Aesop and Darwin only support development with components implemented in C++, while MetaH is restricted to Ada and UniCon to C++. On the other hand, C2 currently supports development in C++, Ada, and Java, while Weaves supports interconnection of tool fragments implemented in C, C++, Objective C, and Fortran.

Several ADLs place restrictions that limit the number and kinds of components and connectors they can support. For example, MetaH requires each component to include a loop with a call to the predeclared procedure KERNEL_WAIT_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. Similarly, UniCon allows certain types of components and connectors (e.g., pipes, filters, and sequential files), but requires wrappers for others (e.g., spreadsheets, constraint solvers, or relational databases).

Finally, another aspect of heterogeneity is the granularity of components. 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 extreme can be achieved by hierarchical composition, discussed in Section 4.3.2.

4.3.5 Scalability

We consider the impact of scaling an architecture along two general dimensions: adding elements to the architecture’s interior (Fig. 16a) and adding them along the architecture’s boundaries (Fig. 16b). To support the former, ADLs can, minimally, employ compositionality features, discussed in Section 4.3.2. The original architecture is treated as a single, composite component, which is then attached to new components and connectors. Objectively evaluating an ADL’s ability to support the latter is more difficult, but certain heuristics can be of help.

It is generally easier to expand architectures described in explicit configuration ADLs than in-line configuration ADLs. Connectors in the latter are described solely in terms of the components they connect and adding new components may require modifications to existing connector instances. Additionally, ADLs that allow a variable number of components to be attached to a single connector are better suited to scaling up than those that specify the exact number of components a connector can service. For example, ACME and Aesop could not handle the extension to the architecture shown in Fig. 16a without redefining Conn1 and Conn2, while C2 and UniCon can.

To properly evaluate an ADL’s support for scalability, these heuristics should be accompanied by other criteria. The ultimate determinant of scalability support is the ability of developers to implement and/or analyze large systems based on the architectural descriptions given in an ADL. For example, as an in-line configuration language, Rapide has been highlighted as an ADL whose features may hamper scalability, yet it has been used to specify architectures of several large, real-world systems. Several other ADLs have been applied to large-scale examples:

- Wright was used to model and analyze the Runtime Infrastructure (RTI) of the Department of Defense High-Level Architecture for Simulations [5], whose original specification was over 100 pages long.
- SADL ensured the consistency between the reference architecture and the implementation of a power-control system used by the Tokyo Electric Power Company, implemented in 200,000 Fortran 77 lines of code (LOC).
- C2 has been used in the specification and implementation of its supporting environment, consisting of a number of large custom-built and OTS components [42], [52]. The custom-built components
comprise over 100,000 Java LOC; the OTS components comprise several million LOC.

- Weaves has been used in satellite telemetry processing applications, whose size has ranged between 100,000 and over 1,000,000 LOC.
- A representative example of Rapide’s use is the X/Open Distributed Transaction Processing Industry Standard, whose documentation is over 400 pages long. X/Open’s reference architecture and subsequent extensions have been successfully specified and simulated in Rapide [31].

### 4.3.6 Evolvability

Evolvability 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. 16. The issues inherent in doing so were discussed in Section 4.3.5. Another view of evolvability 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 situations 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 families of applications. One way in which all ADLs support families is by separating component and connector types from instances. For example, Weaves supports specification of architectural frameworks, which are populated with sockets, rather than actual tool fragments and transport services. Each instantiation of an architecture can then be considered a member of the same family. This is a limited notion of family, as it does not allow the architecture itself to be modified. Furthermore, the family to which an application belongs is implicit in its architecture.

ACME is the only surveyed language that specifies architectural families explicitly, as first-class language constructs, and supports their evolution. The component and connector types declared in a family provide a design vocabulary for all systems that are declared as members of that family. The example given in Fig. 17 shows the declaration of a simple ACME family and its evolution.

### 4.3.7 Dynamism

The majority of existing ADLs view configurations statically. The exceptions are C2, Darwin, Rapide, and Weaves. Darwin and Rapide support only constrained dynamic manipulation of architectures, where all runtime changes must be known a priori [51], [52]. Darwin allows runtime replication of components via dynamic instantiation, as well as deletion and rebinding of components by interpreting Darwin scripts. An example of dynamic instantiation in Darwin is given in Fig. 18: Invoking the service `create_inst` with a `data` parameter results in a new instance of component `comp` to which `data` is passed.

Rapide supports conditional configuration: Its `where` clause enables architectural rewiring at runtime, using the `link` and `unlink` operators. Recently, Wright has adopted a similar approach to dynamic architecture changes: It distinguishes between communication and control events, where the control events are used to specify conditions under which dynamic changes are allowed [3]. The reconfiguration actions that are triggered in response to control events are: `new`, `del`, `attach`, and `detach`.

C2 and Weaves support dynamic manipulation without any restrictions on the types of permitted changes. Instead, arbitrary modifications are allowed in principle; their consistency is ensured at system runtime. C2’s architecture modification (sub)language (AML) specifies a set of operations for insertion, removal, and rewiring of elements in an architecture at runtime: `addComponent`, `removeComponent`, `weld`, and `unweld` [38], [52]. For example, the extension to the architecture depicted in Fig. 16a is specified in C2’s AML as shown in Fig. 19. Weaves provides similar support by exporting an application programmable interface (API) to a model of a weave.

### 4.3.8 Constraints

Most ADLs enforce built-in constraints on what they consider to be valid configurations. For example, UniCon always requires a connector role to be attached to a component player, while Darwin only allows bindings between provided and required services. On the other hand, several ADLs provide facilities for specifying
4.4 Tool Support for ADLs

The need for tool support in architectures is well-recognized. However, there is a definite gap between what the research community identifies as desirable and the state of the practice. While every surveyed ADL provides some tool support, with the exception of C2 and Rapide, they tend to focus on a single area of interest, such as analysis (e.g., Wright), refinement (e.g., SADL), or dynamism (e.g., Weaves). 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.

4.4.1 Active Specification

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 prescriptive 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. Similarly, C2’s DRADEL development environment proactively guides the “architecting” process by disallowing certain operations (e.g., architectural type checking) before others are completed (e.g., topological constraint checking) [42]. Darwin’s Software Architect’s Assistant [50] is another example of a proactive specification tool. The Assistant automatically adds services (i.e., interface points) 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.

Reactive specification tools detect existing errors. They may either only inform the architect of the error (nonintrusive) 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. C2’s DRADEL environment includes a type checker that provides nonintrusive support: The architect can proceed to the implementation generation phase even in the presence of type mismatches. 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: The MetaH 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.

4.4.2 Multiple Views

Most ADLs support at least two views of an architecture—textual and graphical—and provide automated support for alternating between them. Aesop, MetaH, UniCon, and Weaves also distinguish different types of components and...
connectors iconically and allow both top-level and detailed views of composite objects.

Support for other views is sparse. C2’s Argo design environment provides a view of the architecture-centered development process [59]. Darwin’s Software Architect’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 and C2 allow visualization of an architecture’s execution behavior by building an executable simulation of the architecture and providing tools for viewing and filtering events generated by the simulation. In particular, Rapide uses its Simulator tool to build the simulation and its Animation Tools to animate its execution. Rapide also provides Poset Browser, a tool that allows viewing events generated by the simulation. Weaves adopts a similar approach: It allows insertion of low-overhead observers into a weave to support real-time execution animation.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>Attachments: explicit, con-</td>
<td>explicit, con-</td>
<td>provided via tem-</td>
<td>rep-maps</td>
<td>open property</td>
<td>aided by explicit</td>
<td>none</td>
<td>ports may only be attached to roles and vice versa</td>
<td>allows any attributes in property lists, but does not operate on them</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cise textual specification</td>
<td>cise textual specification</td>
<td>plates, rep-</td>
<td></td>
<td>lists; required</td>
<td>configurations;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>repre-</td>
<td>explicit mappings across</td>
<td>hampered by fixed number of roles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>senta-</td>
<td>ADLs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aesop</td>
<td>Configura-</td>
<td>explicit, con-</td>
<td>provided via repre-</td>
<td>none</td>
<td>allows multiple</td>
<td>aided by explicit</td>
<td>none</td>
<td>ports may only be attached to roles and vice versa</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tory: parallel type hierarchy for visualization</td>
<td>cise graphical specification;</td>
<td>sentations</td>
<td></td>
<td>languages for modeling seman-</td>
<td>configurations;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tics; supports development in</td>
<td>hampered by fixed number of roles</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>Architect-</td>
<td>explicit, con-</td>
<td>allowed; supported</td>
<td>none</td>
<td>enabled by inter-</td>
<td>aided by explicit</td>
<td>none</td>
<td>unanticipated dynamism:</td>
<td>fixed stylistic</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>tural: ex-</td>
<td>cise graphical specification;</td>
<td>via inter-</td>
<td></td>
<td>nal component</td>
<td>configurations and variable</td>
<td></td>
<td>element insertion, removal, and rewiring</td>
<td>invariants</td>
<td></td>
</tr>
<tr>
<td></td>
<td>plicit;</td>
<td>parallel type hierarchy for visualization</td>
<td>nal component</td>
<td></td>
<td>architecture; supports</td>
<td>number of connector ports;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>architecture</td>
<td></td>
<td>development in C++, Java, and Ada</td>
<td>construction of its own tool suite</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Darwin</td>
<td>Binding:</td>
<td>in-line textual specification with many con-</td>
<td>supported by lan-</td>
<td></td>
<td>allows multiple languages for modeling seman-</td>
<td>no support for par-</td>
<td>constrained dynamism:</td>
<td>provided services may only be bound to required services and vice versa</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in-line</td>
<td>nector details; provides graph-</td>
<td>guage’s composite</td>
<td></td>
<td>tics of primitive components; supports develop-</td>
<td>tial architectures;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>textual specification</td>
<td>ical notation</td>
<td>component feature</td>
<td></td>
<td>ment in C++</td>
<td>hampered by in-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with many con-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>line configurations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>nector details;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>provides graph-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ical notation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MetaH</td>
<td>Connect-</td>
<td>in-line textual specification with many con-</td>
<td>supported via macros</td>
<td></td>
<td>supports system generation; implemen-</td>
<td>no support for par-</td>
<td>none</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tory: in-</td>
<td>nector details; provides graph-</td>
<td></td>
<td></td>
<td>tation con-</td>
<td>tial architectures; hampered by in-line configurations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>line</td>
<td>ical notation</td>
<td></td>
<td></td>
<td>straining</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>with many con-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nector details;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>provides graph-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ical notation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rapide</td>
<td>Configu-</td>
<td>in-line textual specification with many con-</td>
<td>mappings relate an</td>
<td>refinement</td>
<td>supports develop-</td>
<td>no support for par-</td>
<td>constrained dynamism:</td>
<td>refined maps constraint valid refinements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tory: in-</td>
<td>nector details; provides graph-</td>
<td>architecture to an</td>
<td>maps enable</td>
<td>ment of executable</td>
<td>tial architectures;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>line</td>
<td>ical notation</td>
<td>interface</td>
<td>comparative simula-</td>
<td>simulations in</td>
<td>hampered by in-line</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>with many con-</td>
<td></td>
<td></td>
<td>tions of architectures</td>
<td></td>
<td>configurations;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>nector details;</td>
<td></td>
<td></td>
<td>at different</td>
<td></td>
<td>hampered by in-line</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>provides graph-</td>
<td></td>
<td></td>
<td>levels</td>
<td></td>
<td>configurations;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ical notation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>used in large-scale</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SADL</td>
<td>Configu-</td>
<td>explicit, con-</td>
<td>mappings relate an</td>
<td>refinement</td>
<td>supports both compo-</td>
<td>no support for par-</td>
<td>none</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tory: ex-</td>
<td>cise graphical specification</td>
<td>architecture to a</td>
<td>maps enable</td>
<td>nent and variable component types;</td>
<td>tial architectures;</td>
<td></td>
<td>players may only be attached to roles and vice versa</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>plicit</td>
<td>component specification</td>
<td>component</td>
<td>correct refinements across style</td>
<td>supports only predefined architecture and vari-</td>
<td>supported by explicit configurations;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>able number of con-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nector roles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UniCon</td>
<td>Connect-</td>
<td>explicit textual and graphical specification;</td>
<td>supported through</td>
<td>supports sys-</td>
<td>supports system generation;</td>
<td>some support for par-</td>
<td>none</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tory: ex-</td>
<td>configuration description may be distrib-</td>
<td>composite compo-</td>
<td>tem genera-</td>
<td>implementation con-</td>
<td>tial architectures;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>plicit</td>
<td>uted</td>
<td>nents and connectors</td>
<td>tion; implemen-</td>
<td>stration con-</td>
<td>supported by explicit configurations and vari-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tation; supports sys-</td>
<td>stration con-</td>
<td>able number of connector types; supports compo-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tem generation;</td>
<td>stration con-</td>
<td>nent wrappers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weaves</td>
<td>Weave:</td>
<td>explicit, con-</td>
<td>supported through</td>
<td>supports sys-</td>
<td>development in C++, Objec-</td>
<td>allows partial archi-</td>
<td>anticipated dynamism:</td>
<td>precludes direct component-to-component links</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>explicit</td>
<td>cise graphical specification</td>
<td>composite sockets</td>
<td>tem generation;</td>
<td>tive C, and For-</td>
<td>tectures; aided by explicit configurations;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tran; requires all tool</td>
<td>support for families via socket-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>frameworks to provide a set of methods</td>
<td>populated frame-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wright</td>
<td>Attach-</td>
<td>explicit, con-</td>
<td>computa-</td>
<td>none</td>
<td>supports both fine-</td>
<td>aided by explicit configu-</td>
<td>constrained dynamism:</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ment: ex-</td>
<td>cise textual specification</td>
<td>tion and glue</td>
<td></td>
<td>and coarse-</td>
<td>rations; aided by explicit configurations;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>plicit</td>
<td></td>
<td></td>
<td></td>
<td>grain elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 4
ADL Support for Modeling Architectural Configurations
4.4.3 Analysis

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 uses CSP to analyze individual connectors and components attached to them for deadlocks; Aesop and C2 ensure style-specific topological constraints and type conformance among architectural elements; MetaH and UniCon support schedulability analysis by specifying nonfunctional properties, such as criticality and priority; finally, SADL can establish relative correctness of two architectures with respect to a refinement map.

Another set of analysis techniques involves simulation of the behavior described in an architecture. Examples are Rapide’s, C2’s, and Weaves’ event monitoring and filtering tools. Similarly, Darwin allows instantiation of parameters and dynamic components to enact “what if” scenarios. A related technique, commonly employed in Weaves, is to insert into the architecture a “listener” component whose only task is to analyze the data it receives from adjacent components.

Language parsers and compilers are another kind of analysis tool. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All of the surveyed languages have parsers. Several (Darwin, MetaH, and UniCon) also have “compilers,” enabling them to generate executable systems from architectural descriptions, provided that component implementations already exist. Rapide’s compiler generates executable simulations of Rapide architectures. C2’s DRADEL environment, on the other hand, provides a tool that generates executable implementation skeletons from an architectural model; the skeletons are completed either by developing new or reusing OTS functionality.

Another aspect of analysis is enforcement of constraints. Parsers and compilers enforce constraints implicit in type information, nonfunctional attributes, component and connector interfaces, and semantic models. Rapide also supports explicit specification of other types of constraints, and 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. C2’s constraint checker currently focuses only on the topological rules of the style; an initial integration with the architecture constraint checking tool, Armani [45], allows specification and enforcement of arbitrary constraints.

4.4.4 Refinement

Several ADLs support direct refinement of architectural models to executable code via “compilation.” 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 4.3.3. Rapide, on the other hand, provides an executable sublanguage that contains many common programming language control structures. C2 goes beyond linking existing modules, but not as far as to provide executable language constructs: An architecture is refined into a partial implementation which contains completion guidelines for developers derived from the architectural description. For example, each method is accompanied by specifications of its precondition and postcondition, as shown in Fig. 21; the developer must only ensure their satisfaction when implementing the method and need not worry about the rest of the system.

Only SADL and Rapide provide tool support for refining 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 more concrete architectural style. Such a proof is performed only once; thereafter, SADL provides a tool that automatically checks whether any two architectures described in the two styles adhere to the mapping. Rapide, on the other hand, supports event maps between individual architectures. The maps are compiled by Rapide’s Simulator so that the Constraint Checker can verify that the events generated during simulation of the concrete architecture satisfy the constraints in the abstract architecture.

4.4.5 Implementation Generation

A large number of ADLs, but not all, support generation of a system from its architecture. Exceptions are SADL, ACME, and Wright, which are currently used strictly as modeling notations and provide no implementation generation support. It is interesting to note that, while SADL focuses on refining architectures, it does not take the final refinement step from architectural descriptions to source code.

Several ADLs employ architectural “compilers,” as already discussed above. Aesop adopts a different approach: It provides a C++ class hierarchy for its concepts and operations, such as components, connectors, ports, roles, and attachments. This hierarchy forms a basis from which an implementation of an architecture may be produced; the hierarchy is, in essence, a domain-specific language for implementing Aesop architectures.

A similar approach is used in C2, which provides a framework of abstract classes for C2 concepts [42]. Components and connectors used in C2 applications are subclassed from the appropriate framework classes. The framework has been implemented in C++, Java, and Ada; several OTS middleware technologies have been integrated with the framework to enable interactions between C2 components implemented in different languages [10]. Application skeletons produced by C2’s code generation facilities discussed above result in instantiated, but partially implemented, framework classes.
4.4.6 Dynamism

The limited support for modeling dynamism in existing ADLs, discussed in Section 4.3.7, is reflected in the limited tool support for dynamism. 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 and Weaves toolsets support dynamism more extensively. Weaves provides a visual editor, Jacquard, which uses the provided API to the architectural model to dynamically manipulate a weave in an arbitrary fashion. C2’s ArchStudio tool [52] enables arbitrary interactive construction, execution, and runtime-modification of C2-style architectures implemented in Java. ArchStudio supports modification of an architecture at runtime by dynamically loading and linking new components or connectors into the architecture. Both C2 and Weaves exploit their flexible connectors (see Section 4.2) to support dynamism.

4.4.7 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, SADL’s toolset consists primarily of a refinement consistency checker, and Weaves has focused on interactive specification and manipulation of architectures. On the other hand, Darwin, Rapide, and UniCon provide powerful architecture modeling environments; C2 and Darwin are the only ADLs that provide tool support in all classification categories. 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 5.
5 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 ways 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, procedural vs. OO). Similarly, formal specification languages have been grouped into model-based, state-based, algebraic, axiomatic, and so forth. 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 languages 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 (CHAM 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 implementations than “mainstream” (implementation independent) ADLs. Thus, for example, although C2 is implementation independent, we assumed this one-to-one relationship in building the initial prototype of our implementation generation tools [42].

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 nonfunctional 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 two ADLs have even attempted to achieve unanticipated 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). C2 has provided the initial demonstration of the feasibility of implementing complex connectors by employing existing research and commercial connector technologies, such as Polylib [56] and CORBA [53]. However, 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 are currently considering several issues: providing a clearer distinction between descriptive languages (e.g., ACME) and those that primarily enable semantic modeling (e.g., Wright); comparing software ADLs to hardware ADLs; and expanding the framework to include other criteria (e.g., support for extensibility). We have 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 advance toward 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.

Acknowledgments

The authors would like to thank the following people for their insightful comments on earlier drafts of this paper: R. Allen, K. Anderson, P. Clements, R. Fielding, D. Garlan, M. Gorlick, W. Griswold, D. Hilbert, A. van der Hoek, P. Kammer, J. Kramer, D. Luckham, J. Magee, R. Monroe, M. Moriconi, K. Nies, P. Oreizy, D. Redmiles, R. Riemschneider, J. Robbins, D. Rosenblum, R. Selby, M. Shaw, S. Vestal, J. Whitehead, and A. Wolf. We also thank the referees of Transactions on Software Engineering for their helpful reviews.

This effort was sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References


Nenad Medvidovic received the BS degree in computer science from Arizona State University, where he graduated summa cum laude in 1992. He received the MS degree in information and computer science and the PhD degree from the Department of Information and Computer Science at the University of California, Irvine in 1995 and 1998, respectively. He is an assistant professor in the Computer Science Department at the University of Southern California in Los Angeles. His research interests focus on software architectures, specifically, architectural styles, modeling and analysis of architectures, architecture-based software evolution, software reuse, and interoperability. His WWW home page can be accessed at http://sunset.usc.edu/~neno/.

Richard N. Taylor received the PhD degree in computer science from the University of Colorado at Boulder in 1980. He was a 1985 recipient of a Presidential Young Investigator Award and, in 1998, was recognized as an ACM Fellow. He is a professor of information and computer science at the University of California at Irvine, where he has been since 1982. His research interests are centered on software architectures, hypermedia, the World Wide Web protocols, and workflow technologies. Professor Taylor is the director of the Institute for Software Research (ISR), an alliance between the corporate community and the University. He was the chairman of ACM’s Special Interest Group on Software Engineering, SIGSOFT (1989-1993). He served as program cochair of the 1997 International Conference on Software Engineering (ICSE ’97) and general chair of the 1999 International Joint Conference on Work Activities, Coordination, and Collaboration. Dr. Taylor is a member of the IEEE Computer Society.