Domains of Concern in Software Architectures and Architecture Description Languages

Nenad Medvidovic and David S. Rosenblum

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

Abstract

Software architectures shift the focus of developers from lines-of-code to coarser-grained elements and their interconnection structure. Architecture description languages (ADLs) have been proposed as domain-specific languages for the domain of software architecture. There is still little consensus in the research community on what problems are most important to address in a study of software architecture, what aspects of an architecture should be modeled in an ADL, or even what an ADL is. To shed light on these issues, we provide a framework of architectural domains, or areas of concern in the study of software architectures. We evaluate existing ADLs with respect to the framework and study the relationship between architectural and application domains. One conclusion is that, while the architectural domains perspective enables one to approach architectures and ADLs in a new, more structured manner, further understanding of architectural domains, their tie to application domains, and their specific influence on ADLs is needed.

Keywords — *software architecture, architecture description language, domain, domain-specific language, architectural domain*

1. Introduction

Software architecture is an aspect of software engineering directed at reducing costs of developing applications and increasing the potential for commonality among different members of a closely related product family [PW92, GS93]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so on.

Many researchers have realized that, to obtain the benefits of an architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques [Gar95, GPT95, Wolf96]. Such languages are needed to demonstrate properties of a system upstream, thus minimizing the costs of errors. They are also needed to provide abstractions adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. A large number of *architecture description languages* (ADLs) has been proposed, each of which embodies a particular approach to the specification and evolution of an architecture. Examples are Rapide [LKA+95, LV95], Aesop [GAO94], MetaH [Ves96], UniCon [SDK+95], Darwin [MDEK95, MK96], Wright [AG94a, AG94b], C2 [MTW96, MORT96, Med96], and SADL [MQR95]. Recently, initial work has been done on an architecture interchange language, ACME [GMW95, GMW97], which is intended to support mapping of architectural specifications from one ADL to another, and hence provide a bridge for their different foci and resulting support tools.

There is still very much a lack of 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. This divergence has resulted in a wide variation of approaches found in this first generation of ADLs. Perhaps even more significantly, there is a wide difference of opinions as to what problems are most important to address in a study of software architecture.

In our previous research, we have provided a foundation for understanding, defining, classifying, and comparing ADLs [Med97, MT97]. In this paper, we build upon those results by identifying and characterizing *architectural domains*, the problems or areas of concern that need to be addressed by ADLs. Understanding these domains and their properties is a key to better understanding the needs of software architectures, architecture-based development, and architectural description and interchange. A study of architectural domains is also needed to guide the development of next-generation ADLs.

This paper presents a framework of architectural domains. We demonstrate that each existing ADL currently supports only a small subset of these domains, and we discuss possible reasons for that. Finally, we consider the relationship between architectural domains and application domains.

While we draw from previous ADL work and reference a number of ADLs, the most significant contribution of this paper is the framework of architectural domains. It provides structure to a field that has been approached largely in an ad-hoc fashion thus far. The framework gives the architect a sound foundation for selecting an ADL and orients discourse away from arguments about notation and more towards solving important engineering problems.

The remainder of the paper is organized as follows. Section 2 provides a short discussion of ADLs. Section 3 presents and motivates each architectural domain, while Section 4 discusses the support for architectural domains in existing ADLs. Section 5 expounds on the relationship between application domains and architectural domains. Discussion and conclusions round out the paper.

2. Overview of ADLs

To properly enable further discussion, several definitions are needed. In this section, we define software architectures, architectural styles, and ADLs.[1] We categorize ADLs, differentiate them from other, similar notations, and discuss examples of use of ADLs in actual projects. Finally, we provide a short discussion on our use of the terms "architecture" and "design."

2.1. Definitions of Architecture and Style

There is no standard definition of architecture, but we will use as our working definition the one provided by Garlan and Shaw [GS93]:

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

Architectural style is "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done" [SC96].

2.2. Definition of ADLs

Loosely defined, "an *ADL* for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [Ves93]. ADLs provide both a concrete syntax and a conceptual framework for modeling a software system's *conceptual* architecture.

The building blocks of an architectural description are
- *components* - units of computation or data stores;
- *connectors* - architectural building blocks used to model interactions among components and rules that govern those interactions; and
- *architectural configurations* - connected graphs of components and connectors that describe architectural structure.

An ADL must provide the means for their *explicit* specification; this criterion enables one 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 formally. Without this information, an architectural description becomes but a collection of (interconnected) identifiers.

An ADL's conceptual framework 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 Petri nets [Pet62], Statecharts [Har87], partially-ordered event sets [LVB+93], communicating sequential processes (CSP) [Hoa85], model-based formalisms (e.g., *CH*emical *A*bstract *M*achine [IW95], Z [Spi89]), algebraic formalisms (e.g., Obj [GW88]), and axiomatic formalisms (e.g., Anna [Luc87]).

Finally, even though the suitability of a given language for modeling architectures is independent of whether

1. This section is condensed from a detailed exposition on ADLs given in [Med97] and [MT97], where we provided a definition of ADLs and devised a classification and comparison framework for them.

and what kinds of *tool support* it provides, an accompanying toolset will render an ADL both more usable and useful. Furthermore, capabilities provided by such a toolset are often a direct reflection of the ADL's intended use.

2.3. Categorizing ADLs

Existing languages that are commonly referred to as ADLs can be grouped into three categories, based on how they model configurations:
- *implicit configuration languages* model configurations implicitly through interconnection information that is distributed across definitions of individual components and connectors;
- *in-line configuration languages* model configurations explicitly, but specify connector information only as part of the configuration, "in line";
- *explicit configuration languages* model both components and connectors separately from configurations.

The first category, implicit configuration languages, are, by definition given in this paper, *not* ADLs, although they may serve as useful tools in modeling certain aspects of architectures. An example of an implicit configuration language is ArTek [TLPD95]. 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 module interconnection languages (MILs) [DK76, PN86], programming languages, and object-oriented notations and languages (e.g., Unified Method [BR95]). MILs typically describe the *uses* relationships among modules in an *implemented* system and support only one type of connection [AG94a, SG94]. Programming languages describe a system's implementation, whose architecture is typically implicit in subprogram definitions and calls. Explicit treatment of connectors also distinguishes ADLs from OO languages, as demonstrated in [LVM95].

It is important to note, however, that there is less than a firm boundary between ADLs and MILs. Certain ADLs, e.g., Wright and Rapide, model components and connectors at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation. We refer to these languages as being *implementation independent*. On the other hand, several ADLs, e.g., UniCon and MetaH, enforce a high degree of fidelity of an implementation to its architecture. 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.

2.4. Applications of ADLs

ADLs are special purpose notations whose very specific foci render them suitable for powerful analyses, simulation, and automated code generation. However, they have yet to find their place in mainstream software development. Although current research is under way to bridge the gap that separates ADLs from more widely used design notations [RMRR97], only a small number of existing ADLs have been applied to large-scale, "real-world" examples to date. What these examples do demonstrate is the potential for effective use of ADLs in software projects.

Wright was used to model and analyze the *Runtime Infrastructure* (RTI) of the Department of Defense (DoD) *High-Level Architecture for Simulations* (HLA) [All96]. The original specification for RTI was over 100 pages long. Wright was able to substantially condense the specification and reveal several inconsistencies and weaknesses in it.

SADL was applied to an operational power-control system, used by the Tokyo Electric Power Company. The system was implemented in 200,000 lines of Fortran 77 code. SADL was used to formalize the system's reference architecture and ensure its consistency with the implementation architecture.

Finally, Rapide has been used in several large-scale projects thus far. A representative example is the X/Open Distributed Transaction Processing (DTP) Industry Standard. The documentation for the standard is over 400 pages long. Its reference architecture and subsequent extensions have been successfully specified and simulated in Rapide [LKA+95].

2.5. Architecture vs. Design

Given the above definition of software architectures and ADLs, an issue worth addressing is the relationship between architecture and design. Current literature leaves this question largely unanswered, allowing for several interpretations:
- architecture and design are the same;
- architecture is at a level of abstraction above design, so it is simply another step (artifact) in a software development process; and
- architecture is something new and is somehow different from design (but just how remains unspecified).

All three interpretations are partially correct. To a large extent, architectures serve the same purpose as design. However, their explicit focus on connectors and configurations distinguishes them from traditional software design. At the same time, as a (high level) architecture is refined, connectors lose prominence by becoming distributed across the (lower level) architecture's elements. Such a lower level architecture may indeed be considered to be a design. Keeping this relationship in mind, for reasons of simplicity we will simply refer to architectures as "high level," "low level," and so forth, in the remainder of the paper, while "design" will only refer to the process that results in an architecture.

3. Architectural Domains

ADLs typically share syntactic constructs that enable them to model components and component interfaces, connectors, and configurations.[2] A much greater source of divergence are the different ADLs' conceptual frameworks, and, consequently, their support for modeling architectural semantics. ADL developers typically have decided to focus on a specific aspect of architectures, or an *architectural domain*, which guides their selection of an underlying semantic model and a set of related formal specification notations. These formal notations, in turn, restrict the types of problems for which the ADL is suitable.

This relationship between an architectural domain and candidate formal notations is rarely straightforward or fully understood. In the absence of objective criteria, ADL researchers are forced to base their decisions on intuition, experience, and biases arising from past research accomplishments. Unfortunately, intuition can often be misleading and experience insufficient in a young discipline such as software architectures.

In this paper, we attempt to fill this void. The remainder of this section motivates and formulates a framework for classifying the problems on which architectural models focus (architectural domains), shown in Figure 1. Architectural domains represent broad classes of problems and are likely to be reflected in many ADLs and their associated formal specification language constructs. Their proper understanding is thus necessary. Furthermore, heuristics may be developed over time that will enable easier interchange of architectures modeled in ADLs that focus on particular architectural domains. Finally, such a framework can be used as a guide in developing future ADLs.

2. One can think of these syntactic features as equivalent to a "boxes and arrows" graphical notation with little or no underlying semantics.

```
Representation
Design Process Support
Analysis
   Static
   Dynamic
Evolution
   Specification-Time
   Execution-Time
Refinement
Traceability
Simulation/Executability
```

Figure 1: Architectural domains.

3.1. Representation

A key role of an explicit representation of an architecture is to aid understanding and communication about a software system among different stakeholders. For this reason, it is important that architectural descriptions be simple, understandable, and possibly graphical, with well understood, but not necessarily formally defined, semantics.

Architectural models typically comprise multiple views, e.g., high level graphical view, lower level view with formal specifications of components and connectors, conceptual architecture, one or more implementation architectures, corresponding development process, data or control flow view, and so on. 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 arrows" description, the developers may want detailed component and connector models, while the managers may require a view of the development process.

3.2. Design Process Support

Software architects decompose large, distributed, heterogeneous systems into smaller building blocks. In doing so, they have to consider many issues, make many decisions, and utilize many design techniques, methodologies, and tools.

Modeling architectures from multiple perspectives, discussed in the previous subsection, is only one way of supporting software architects' cognitive processes. Others include delivering design guidance in a timely and understandable fashion, capturing design rationale, and revisiting past design steps.

3.3. Analysis

Architectures are often intended to model large, distributed, concurrent systems. The ability to evaluate the properties of such systems upstream, at the architectural level, can substantially lessen the number of errors passed downstream. Given that unnecessary details are abstracted away in architectures, the analysis task may also be easier to perform than at source code level.

Analysis of architectures may be performed statically, before execution, or dynamically, at runtime. Certain types of analysis can be performed both statically and dynamically.

3.3.1. Static Analysis

Examples of static analysis are internal consistency checks, such as whether appropriate components are connected and their interfaces match, whether connectors enable desired communication, whether constraints are satisfied, and whether the combined semantics of components and connectors result in desired system behavior. Certain concurrent and distributed aspects of an architecture can also be assessed statically, such as the potential for deadlocks and starvation, performance, reliability, security, and so on. Finally, architectures can be statically analyzed for adherence to design heuristics and style rules.

3.3.2. Dynamic Analysis

Examples of dynamic analysis are testing, debugging, assertion checking, and assessment of the performance, reliability, and schedulability of an executing architecture. Saying that an architecture is executing can mean two different things:

- the system built based on the architecture is executing, or
- the runtime behavior of the architecture itself is being simulated.

Clearly, certain analyses, such as performance or reliability, are more meaningful or even only possible in the former case. However, an implementation of the system may not yet exist. Furthermore, it may be substantially less expensive to perform dynamic analyses in the latter case, particularly when the relationship between the architecture and the implemented system is well understood.

3.4. Evolution

Support for software evolution is a key aspect of architecture-based development. Architectures evolve to reflect evolution of a single software system; they also evolve into families of related systems. As design elements, individual components and connectors within an architecture may also evolve.

Evolution of components, connectors, and architectures can occur at specification time or execution time.

3.4.1. Specification-Time Evolution

If we consider components and connectors to be types which are instantiated every time they are used in an architecture, their evolution can be viewed simply in terms of subtyping. Since components and connectors are modeled at a high level of abstraction, flexible subtyping methods may be employed. For example, it may be useful to evolve a single component in multiple ways, by using different subtyping mechanisms (e.g., interface, behavior, or a combination of the two) [MORT96].

At the level of architectures, evolution is focused on incremental development and support for system families. Incrementality of an architecture can further be viewed from two different perspectives. One is its ability to accommodate addition of new components and the resulting issues of scale; the other is specification of incomplete architectures.

3.4.2. Execution-Time Evolution

Explicit modeling of architectures is intended to support development and evolution of large and potentially long-running systems. Being able to evolve such systems during execution may thus be desirable and, in some cases, necessary. Architectures exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or subarchitectures during execution.

Dynamic changes of an architecture may be either planned at architecture specification time or unplanned. Both types of dynamic change must be constrained to ensure that no desired architectural properties are violated.

3.5. Refinement

The most common argument for creating and using formal architectural models is that they are necessary to bridge the gap between informal, “boxes and arrows” diagrams and programming languages, which are deemed too low-level for designing a system. Architectural models may need to be specified at several levels of abstraction for different purposes. For example, a high level specification of the architecture can be used as an understanding and communication tool; a subsequent lower level may be analyzed for consistency of interconnections; an even lower level may be used in a simulation. Therefore, correct and consistent refinement of architectures to subsequently lower levels of abstraction is imperative. Note that, in this sense, code generation is simply a special case of architectural refinement.

3.6. Traceability

As discussed above, a software architecture often consists of multiple views and may be modeled at multiple levels of abstraction (Figure 2). We call a particular view of the architecture at a given level of abstraction (i.e., a single point in the two-dimensional space of Figure 2) an “architectural cross-section.” It is critical for changes in one cross-section to be correctly reflected in others. A particular architectural cross-section can be considered “dominant,” so that *all* changes to the architecture are made to it and then reflected in others. However, changes will more frequently be made to the most appropriate or convenient cross-section. Traceability support will hence need to exist across all pertinent cross-sections.

One final issue is the consistency of an architecture with system requirements. Changes to the requirements must be appropriately reflected in the architecture; changes to the architecture must be validated against the requirements. Therefore, even though system requirements are in the problem domain, while architecture is in the solution domain, traceability between the two is crucial. For purposes of traceability, requirements can be considered to be at a very high level of architectural abstraction, as shown in Figure 2.

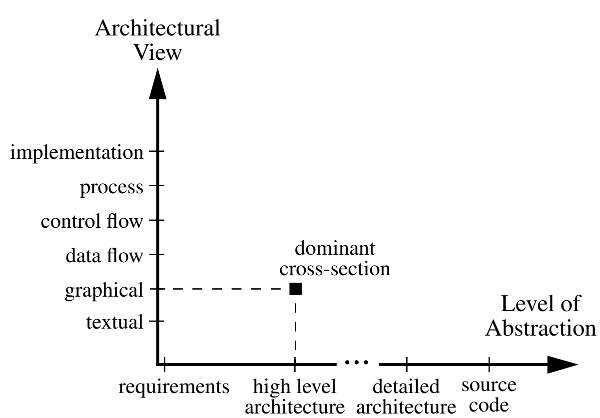

Figure 2: Two-dimensional space of architectural views and levels of abstraction. The vertical axis is a set of discrete values with a nominal ordering. The horizontal axis is a continuum with an ordinal ordering of values, where system requirements are considered to be the highest level of abstraction and source code the lowest. One possible dominant cross-section (graphical view of the high level architecture) is shown.

3.7. Simulation/Executability

Static architectural models are useful for establishing *static* properties of the modeled system. Certain dynamic properties may also be predicted with static models, but only if specific assumptions hold. For example, if the architect can correctly predict execution time and criticality of each component, then schedulability of the encompassing architecture can be evaluated.

On the other hand, other dynamic properties, such as reliability, may by definition require a running system. Also, developers may want to produce an early prototype to, e.g., attempt allocation of architectural elements to components of the physical system. Other stakeholders (e.g., customers or managers) may want to verify early on that the architecture conforms to their wishes. Simulating the dynamic behavior of a high level architecture may thus be preferred to implementing the system: it is a quicker, cheaper, and more flexible way of arriving at the desired information.

A special case of architectural simulation is the execution of the complete implemented system. The ultimate goal of any software design and modeling endeavor is to produce such a system. An elegant and effective architectural model is of limited value, unless it can be converted into a running application. A simulation can only partially depict the final system’s dynamic behavior. Manually transforming an architecture into a running system may result in many, already discussed problems of consistency and traceability between the architecture and its implementation. Techniques, such as refinement and traceability discussed above, must be employed to properly accomplish this task.

4. ADL Support for Architectural Domains

In the previous section, we motivated and described different architectural domains in terms of their characteristics and needs of software architectures. Another way of viewing architectural domains is in terms of modeling languages and specific language features needed to support different domains. At the same time, a useful way of understanding and classifying architecture modeling languages is in terms of architectural domains they are intended to support. For these reasons, this section studies the kinds of language facilities that are needed to support each architectural domain, as well as the specific features existing ADLs employ to that end. Our hope is that this discussion will shed light on the relationships among different architectural domains (and their resulting ADL features) and point out both where they can be effectively combined and where we can expect difficulties.

4.1. Representation

Ideally, an ADL should make the structure of a system clear from a configuration specification alone, i.e., without having to study component and connector specifications. Architecture descriptions in *in-line configuration ADLs*, such as Darwin, MetaH, and Rapide tend to be encumbered with connector details, while *explicit configuration ADLs*, such as ACME, Aesop, C2, SADL, UniCon, and Wright have the best potential to facilitate understandability of architectural structure.

One common way of facilitating understandability and communication is by providing a graphical notation, in addition to the textual notation. However, this is only the case if there is a precise relationship between a graphical description and the underlying semantic model. For example, Aesop, C2, Darwin, MetaH, Rapide, and UniCon support such "semantically sound" graphical notations, while ACME, SADL, and Wright do not.

ADLs must also be able to model the architecture from multiple perspectives. As discussed above, several ADLs support at least two views of an architecture: textual and graphical. Each of these ADLs also allows both top-level and detailed views of composite elements. Aesop, MetaH, and UniCon further distinguish different types of components and connectors iconically.

Support for other views is sparse. C2 provides a view of the development process that corresponds to the architecture [RR96]. Darwin's *Software Architect's Assistant* [NKM96] provides a hierarchical view of the architecture which shows all the component types and the "include" relationships among them in a tree structure. Rapide allows visualization of an architecture's execution behavior by building its simulation and animating its execution. Rapide also provides a tool for viewing and filtering events generated by the simulation.

4.2. Design Process Support

As the above examples of C2's, Darwin's, and Rapide's support tools indicate, language features can only go so far in supporting software architects. Adequate tools are also needed. A category of tools that is critical for adequately supporting the design process are *active specification tools*; they can significantly reduce the cognitive load on architects.

Only a handful of existing ADLs provide tools that actively support specification of architectures. In general, such tools can be proactive or reactive. UniCon's graphical editor is proactive. It invokes UniCon's language processing facilities to *prevent* errors during design. Reactive specification tools detect *existing* errors. They may either only inform the architect of the error (*non-intrusive*) or also force the architect to correct it before moving on (*intrusive*). An example of the former is C2's design environment, *Argo*, while MetaH's graphical editor is an example of the latter.

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. The semantic model will largely influence whether the ADL can be analyzed statically or dynamically, or both. For example, Wright, which is based on communicating sequential processes (CSP) [Hoa85], allows static deadlock analysis of individual connectors and components attached to them. On the other hand, Rapide architectures, which are modeled with partially ordered event sets (posets) [LVB+93], can be analyzed dynamically.

4.3.1. Static Analysis

The most common type of static analysis tools are language parsers and compilers. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All existing ADLs have parsers. Darwin, MetaH, Rapide, and UniCon also have compilers, which enable these languages to generate executable systems from architectural descriptions. Wright does not have a compiler, but it uses FDR [For92], a model checker, to establish type conformance.

There are numerous other possible types of static analysis of architectures. Several examples are provided by current ADLs. Aesop provides facilities for checking for type consistency, cycles, resource conflicts, and scheduling feasibility in its architectures. C2 uses critics to establish adherence to style rules and design guidelines. MetaH and UniCon both currently support schedulability analysis by specifying non-functional properties, such as criticality and priority. Finally, given two architectures, SADL can establish their relative correctness with respect to a refinement map.

4.3.2. Dynamic Analysis

The ability to analyze an architecture dynamically directly depends on the ADL's ability to model its dynamic behavior. To this end, ADLs can employ specification mechanisms, such as event posets, CHAM, or temporal logic, which can express dynamic properties of a system. Another aspect of dynamic analysis is enforcement of constraints at runtime.

Most existing ADLs tend to view architectures statically, so that current support for dynamic modeling and analysis is scarce. Darwin enables dynamic analysis of architectures by instantiating parameters and components to enact "what if" scenarios. Similarly, Rapide *Poset Browser*'s event filtering features and *Animation Tools* facilitate analysis of architectures through simulation. Rapide's *Constraint Checker* also analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture. Finally, runtime systems of those ADLs that provide architecture compilation support can be viewed as dynamic analysis facilities.

4.4. Evolution

An architecture can evolve in two different dimensions:

- evolution of individual components and connectors, where the structure of the architecture is not affected, although its behavior may be; and
- evolution of the entire architecture, which affects both the structure and behavior of an architecture.

Evolution in these two dimensions can occur both at architecture specification time and while the architecture is executing.[3]

4.4.1. Specification-Time Evolution

ADLs can support specification-time evolution of individual components and connectors with subtyping. Only a subset of existing ADLs provide such facilities, and even their evolution support is limited and often relies on the chosen implementation (programming) language. The remainder of the ADLs view and model components and connectors as inherently static.

Aesop supports behavior-preserving subtyping of components and connectors to create substyles of a given architectural style. Rapide allows its interface types to inherit from other types by using OO methods, resulting in structural subtyping. ACME also supports structural subtyping via its *extends* feature. C2 provides a more sophisticated subtyping and type checking mechanism. Multiple subtyping relationships among components are allowed: name, interface, behavior, and implementation subtyping, as well as their combinations [MORT96].

Specification-time evolution of complete architectures has two facets: support for incremental development and support for system families. Incrementality of an architecture can be viewed from two different perspectives. One is its ability to accommodate addition of new components to the architecture. In general, *explicit configuration ADLs* can support incremental development more easily and effectively than *in-line configuration ADLs*; ADLs that allow variable numbers of components to communicate through a connector are well suited for incremental development, particularly when faced with unplanned architectural changes [Med97].

Another view of incrementality is an ADL's support for incomplete architectural descriptions. Incomplete architectures are common during design, as some decisions are deferred and others have not yet become relevant. However, most existing ADLs and their supporting toolsets have been built to prevent precisely these kinds of situations. 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.

Still another aspect of static evolution is support for application families. In [MT96], we showed that the number of possible architectures in a component-based style grows exponentially as a result of a linear expansion of a collection of components. All such architectures may not belong to the same logical family. Therefore, relying on component and connector inheritance, subtyping, or other evolution mechanisms is insufficient. An obvious solution, currently adopted only by ACME, is to provide a language construct that allows the architect to specify the family to which the given architecture belongs.

4.4.2. Execution-Time Evolution

There are presently two approaches to supporting evolution of architectures at execution time. The first is what Oreizy calls "constrained dynamism": all runtime changes to the architecture must be known a priori and are specified as part of the architectural model [Ore96].

Two existing ADLs support constrained dynamism. Rapide supports conditional configuration; its *where* clause enables a form of architectural rewiring at runtime, using the *link* and *unlink* operators. Darwin allows runtime replication of components using the *dyn* operator.

The second approach to execution time evolution places no restrictions at architecture specification time on the

3. Saying that an architecture is "executing" can mean either that the architecture is being simulated or that the executable system built based on that architecture is running.

kinds of allowed changes. Instead, the ADL has an architecture modification feature, which allows the architect to specify changes while the architecture is running.

Darwin and C2 are the only ADLs that support such "pure dynamism" [Ore96]. Darwin allows deletion and rebinding of components by interpreting Darwin scripts. C2 specifies a set of operations for insertion, removal, and rewiring of elements in an architecture at runtime [Med96]. C2's *ArchShell* tool enables arbitrary interactive construction, execution, and runtime-modification of C2-style architectures by dynamically loading and linking new architectural elements [Ore96, MOT97]. An issue that needs further exploration is constraining pure dynamic evolution to ensure that the desired properties of architectures are maintained.

4.5. Refinement

ADLs provide architects with expressive and semantically elaborate facilities for specification of architectures. However, an ADL must also enable correct and consistent refinement of architectures to subsequently lower levels of abstraction, and, eventually, to executable systems.

An obvious way in which ADLs can support refinement is by providing patterns, or maps, that, when applied to an architecture, result in a related architecture at a lower level of abstraction. SADL and Rapide are the only two ADLs that provide such support. SADL uses maps to enable correct architecture refinements across styles, while Rapide generates comparative simulations of architectures at different abstraction levels. Both approaches have certain drawbacks, indicating that a hybrid approach may be useful.

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

Several ADLs take a different approach to refinement: they enable generation of executable systems directly from architectural specifications. These are typically the *implementation constraining languages*, such as MetaH and UniCon. These ADLs assume the existence of a source file that corresponds to a given architectural element. This approach makes the assumption that the relationship between elements of an architectural description and those of the resulting system will be 1-to-1. Given that architectures are intended to describe systems at a higher level of abstraction than source code modules, this can be considered only a limited form of refinement.

4.6. Traceability

While the problem of refinement essentially focuses only on one axis of Figure 2 (the horizontal axis) and one direction (left to right), traceability may need to cover a large portion of the two-dimensional space and is applicable in both directions. This presents a much more difficult task, indicating why this is the architectural domain in which existing ADLs are most lacking.

The relationships among architectural views (vertical axis) are not always well understood. For example, ADLs commonly provide support for tracing changes between textual and graphical views, such that changes in one view are automatically reflected in the other; however, it may be less clear how the data flow view should affect the process view. In other cases, changes in one view (e.g., process) should never affect another (e.g., control flow). An even bigger hurdle is providing traceability support across *both* architectural views and levels of abstraction simultaneously. Finally, although much research has been directed at methodologies for making the transition from requirements to design (e.g., OO), this process is still an art form. Further research is especially needed to understand the effects of changing requirements on architectures and vice versa.

Traceability is particularly a problem in the way implementation constraining languages approach code generation, discussed in the previous subsection. These ADLs provide no means of guaranteeing that the source modules which are supposed to implement architectural components will do so correctly. Furthermore, even if the specified modules currently implement the needed behavior correctly, there is no guarantee that any future changes to those modules will be traced back to the architecture and vice versa.

4.7. Simulation/Executability

As with dynamic analysis (Section 4.3.2), simulating an architecture will directly depend upon the ADL's ability to model its dynamic behavior. Currently, Rapide is the only ADL that can simulate the architecture itself, by generating event posets. Other ADLs enable generation of running systems corresponding to the architecture.

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

C2 and Aesop provide class hierarchies for their concepts and operations, such as components, connectors, and interconnection and message passing protocols. These hierarchies form a basis from which an implementation of an architecture may be produced. Aesop's hierarchy has been implemented in C++, and C2's in C++, Java, and Ada.

4.8. Summary

Existing ADLs span a broad spectrum in terms of the architectural domains they support. On the one hand, languages like SADL and Wright have very specific, narrow foci. On the other, C2, Rapide, and Darwin support a number of architectural domains. Certain domains, e.g., evolution, refinement, and traceability are only sparsely supported, indicating areas around which future work should be centered. A more complete summary of this section is given in Table 1 below.

Table 1: ADL Support for Architectural Domains

Arch. Domain / ADL	Represent.	Design Process Support	Static Analysis	Dynamic Analysis	Spec-Time Evolution	Exec-Time Evolution	Refinement	Trace.	Simulation/ Executability
ACME	explicit config.; “weblets”	none	parser	none	application families	none	rep-maps across levels	textual <-> graphical	none
Aesop	explicit config.; graphical notation; types distinguished iconically	syntax directed editor; specialized editors for visualization classes	parser; style-specific compiler; type, cycle, resource conflict, and scheduling feasibility checker	none	behavior-preserving subtyping of components and connectors	none	none	textual <-> graphical	*build* tool constructs system glue code in C for pipe-and-filter style
C2	explicit config.; graphical notation; process view; simulation; event filtering	non-intrusive, reactive design critics and to-do lists in *Argo*	parser; critics to establish adherence to style rules and design heuristics	event filtering	multiple subtyping mechanisms; allows partial architectures	pure dynamism: element insertion, removal, and rewiring	none	textual <-> graphical	class framework enables generation of C/C++, Ada, and Java code
Darwin	implicit config.; graphical notation; hierarchical system view	automated addition of ports; propagation of changes across bound ports; property dialogs	parser; compiler	“what if” scenarios by instantiating parameters and dynamic components	none	constrained dynamism: runtime replication of components and conditional configuration	none	textual <-> graphical	compiler generates C++ code
MetaH	implicit config.; graphical notation; types distinguished iconically	intrusive, reactive graphical editor	parser; compiler; schedulability, reliability, and security analysis	none	none	none	none	textual <-> graphical	compiler generates Ada code (C code generation planned)
Rapide	implicit config.; graphical notation; animated simulation; event filtering	none	parser; compiler; constraint checker to ensure valid mappings	event filtering and animation	inheritance (structural subtyping)	constrained dynamism: conditional configuration and dynamic event generation	refinement maps enable comparative simulations of architectures at different levels	textual <-> graphical; constraint checking across refinement levels	simulation by generating event posets; system construction in C/C++, Ada, VHDL, and Rapide
SADL	explicit config.	none	parser; relative correctness of architectures w.r.t. a refinement map	none	component and connector refinement via pattern maps	none	maps enable correct refinements across levels	refinement across levels	none
UniCon	explicit config.; graphical notation	proactive GUI editor invokes language checker	parser; compiler; schedulability analysis	none	none	none	none	textual <-> graphical	compiler generates C code
Wright	explicit config.	none	parser; model checker for type conformance; deadlock analysis of connectors	none	type conformance for behaviorally related protocols	none	none	none	none

5. Architectural vs. Application Domains

Over the past decade there has been interest in relating architectures, which are in the solution domain, to the problem (or application) domain, leading to the notion of *domain-specific software architectures (DSSAs)* [Tra95]. A DSSA provides a single (generic) *reference architecture*, which reflects the characteristics of a particular problem domain, and which is instantiated for each specific application in that domain. *Architectural styles*, discussed in Section 2, provide another way of relating the problem and solution spaces. Styles are largely orthogonal to DSSAs: a single style may be applicable to multiple application domains; on the other hand, a single DSSA may use multiple styles.

Any attempt to further explore and perhaps generalize the relationship between architectural and application domains would be greatly aided by a classification of application domains. We are unaware of any such classification, although Jackson identified a number of *domain characteristics* that could serve as a starting point for one [Jac95]:

- *static* vs. *dynamic* domains, with the latter being application domains having an element of time, events, and/or state;
- *one-dimensional* vs. *multi-dimensional* domains;
- *tangible* vs. *intangible* domains, with the latter typically involving machine representations of abstractions (such as user interfaces);
- *inert* vs. *reactive* vs. *active* dynamic domains; and
- *autonomous* vs. *programmable* vs. *biddable* active dynamic domains.

Given these application domain characteristics, one can easily identify a number of useful relationships with architectural domains. For instance, support for evolution, executability and dynamic analysis are more important for dynamic domains than for static domains. As another example, reactive domains are naturally supported by a style of representation (e.g., Statecharts [Har87]) that is different from that in active domains (e.g., CHAM [IW95]). As we deepen our understanding of architectural domains, we will be able to solidify our understanding of their relationship with application domains.

6. Conclusions

Software architecture research has been moving forward rapidly. A number of ADLs and their supporting toolsets have been developed; many existing styles have been adopted and new ones invented. Theoretical underpinnings for the study of software architectures have also begun to emerge in the form of definitions [PW92, GS93] and formal classifications of styles [SC96] and ADLs [Med97, MT97].

This body of work reflects a wide spectrum of views on what architecture is, what aspects of it should be modeled and how, and what its relationship is to other software development concepts and artifacts. This divergence of views has also resulted in a divergence of ADLs' conceptual frameworks (as defined in Section 2). Such fragmentation has made it difficult to establish whether there exists in ADLs a notion similar to computational equivalence in programming languages. Furthermore, sharing support tools has been difficult.

ACME has attempted to provide a basis for interchanging architectural descriptions across ADLs. However, ACME has thus far been much more successful at achieving architectural interchange at the syntactic (i.e., structural) level, than at the semantic level. Although some of the ACME team's recent work looks encouraging, this still remains an open problem. One of the reasons ACME has encountered difficulties is precisely the fact that there is only limited agreement in the architecture community on some fundamental issues, the most critical of which is what problems architectures should attempt to solve.

This paper presents an important first step towards a solution to this problem. We have recognized that the field of software architecture is concerned with several domains and that every ADL reflects the properties of one or more domains from this set. Architectural domains thus provide a unifying view to what had seemed like a disparate collection of approaches, notations, techniques, and tools. The task of architectural interchange can be greatly aided by studying the interrelationships among architectural domains. Existing ADLs can be better understood in this new light and new ADLs more easily developed to solve a specific set of problems.

Much further work is still needed, however. Our current understanding of the relationship between architectural domains and formal semantic theories (Section 2) is limited. Also, we need to examine whether there exist techniques that can more effectively support the needs of particular architectural domains than those provided by existing ADLs. Finally, a more thorough understanding of the relationship between architectural and application domains is crucial if architecture-based development is to fulfill its potential.

7. Acknowledgements

We would like to thank Richard Taylor, Peyman Oreizy, Jason Robbins, David Redmiles, and David Hilbert for their participation in numerous discussions of issues concerning ADLs. We also thank the DSL reviewers for their helpful reviews.

Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0218 and F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Approved for Public Release — Distribution Unlimited.

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.

This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973.

8. References

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

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

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

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

[DK76] F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, pages 80-86, June 1976.

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

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

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

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

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

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

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

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

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

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

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

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

[Jac95] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

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

[Luc87] D. Luckham. *ANNA, a language for annotating Ada programs: reference manual*, volume 260 of

Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1987.

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

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

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

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

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

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

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

[MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 17-23, 1997.

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

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

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

[MT97] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. To appear in *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 22-25, 1997.

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

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

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

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

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

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

[RMRR97] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. Technical Report, UCI-ICS-97-35, University of California, Irvine, August 1997.

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

[SC96] M. Shaw and P. Clements. Toward Boxology: Preliminary Classification of Architectural Styles. In A. L. Wolf, ed., *Proceedings of the Second*

International Software Architecture Workshop (ISAW-2), pages 50-54, San Francisco, CA, October 1996.

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

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

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

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

[Tra95] W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, July 1995.

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

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

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