An Evaluation of ADLs on Modeling Patterns for Software Architecture

Ahmad Waqas Kamal, Paris Avgeriou

Department of Mathematics and Computer Science
University of Groningen, the Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

Abstract. Architecture patterns provide solutions to recurring design problems at the architecture level. In order to model patterns during software architecture design, one may use a number of existing Architecture Description Languages (ADLs), including the UML, a generic language but also a de facto industry standard. Unfortunately, there is little explicit support offered by such languages to model architecture patterns, mostly due to the inherent variability that patterns entail. In this paper, we analyze the support that few selected languages offer in modeling a limited set of architecture patterns with respect to four specific criteria: syntax, visualization, variability, and extensibility. The results highlight the strengths and weaknesses of the selected ADLs for modeling architecture patterns in software design.

Keywords: Software Architecture**,** Architecture Patterns, Modeling, ADLs, UML.

1 Introduction

Architecture patterns [20] [26] entail solutions to recurring architecture design problems and thus provide a systematic way to architecture design. They offer re-use of valuable architectural knowledge, understanding, and communication of software architecture and support for quality attributes [26]. Architecture patterns are usually described and therefore modeled as configurations of components and connectors [4]. The components comprise the major subsystems of a software system and they are linked through connectors, which facilitate flow of data and define rules for communication among components. Examples of connectors are shared variable accesses, table entries, buffers, procedure calls, network protocols, etc. [22]. Connectors play a major role in modeling patterns for software architecture design.

In current software engineering practice, architecture patterns have become an integral part of architecture design, and often modeled with the use of Architecture Description Languages (ADLs): specialized languages for explicit modeling and analysis of software architecture [5]. UML is also used in practice for modeling software architecture, and we shall include it in the general category of ADLs, even though it is not strictly speaking an ADL. These languages are required to not only model general architecture constructs, but also pattern-specific syntax and semantics. Indeed, few ADLs, like Aesop [4], UniCon [21], and ACME [7] provide some inherent support for modeling specific concepts of architecture patterns. However, ADLs lack explicit support for modeling patterns, and are too limited in the abstractions they provide to model the rich concepts found in patterns [2] [4] [7].

In this paper, we attempt to evaluate the strengths and weaknesses of existing ADLs for modeling architecture patterns. We establish a comparison framework that is composed of features needed in ADLs for effectively modeling architecture patterns. Using this framework, we evaluate the most popular or commonly used ADLs, with respect to four of the most significant architecture patterns. The comparison framework consists of the following criteria:

- **Syntax** – expressing pattern elements, topology, constraints and configuration of components and connectors
- **Visualization** – graphical representation for modeling patterns
- **Variability** – the ability to express not only individual solutions but the entire space of solution variants
- **Extensibility** – capability to model new patterns

Our purpose is to evaluate the capabilities of ADLs with respect to modeling architecture patterns. It is not a scorecard to compare one ADL against other ADLs; rather it facilitates architects to select ADLs that best meet their needs to model architecture patterns. The focus of this paper is on domain independent languages. For the evaluation, we have selected six languages: UML, ACME, Wright, Aesop, UniCon and xADL. To make the aforementioned criteria workable, we use four different architecture patterns, namely Layers, Pipe-Filter, Blackboard, and Client-Server. The selection of these ADLs and patterns is not exhaustive but serves the purpose for a first evaluation of ADLs w.r.t. modeling patterns.

The remainder of this paper is organized as follows. In section 2, we introduce the theoretical background of patterns and current state of the practice in modeling patterns. Section 3 explains the comparison framework, while the evaluation of the languages is presented in section 4. Section 5 contains related work and Section 6 wraps up with conclusions and future work.

2 Theoretical Background and State of the Practice

Architecture Patterns

During the last decade, there has been a considerable effort for the systematic re-use of architecture patterns as solutions to recurring problems at the architecture design level [9] [10] [18]. Numerous architecture patterns are in use and this list is growing continuously [9] [29]. Some of the research activities in the pattern community for the past few years have been: discovery of new patterns [26] [27], combined use of patterns as pattern languages [1] [14], and using patterns in software architecture design [4] [7] [8] [21].

Among a number of software patterns that exist in the literature, architectural patterns, and design patterns [23] are the most widely known and used. It is difficult to draw a clear boundary between both types of these patterns, because it depends on the way these patterns are perceived and used by software architects. The work in POSA [26] lists some traditional architectural patterns, while work in GOF [27] lists 23 specific solutions to design problems. GOF is more concerned about object-oriented issues of the system design, while the work in POSA is more concerned about architecture issues, i.e. high-level components and connectors. In this paper we focus on the latter.

Another terminological difference that often causes confusion is that between architecture patterns [26] and architecture styles [33]. These two terms come from two different schools of thoughts. Their commonality lies in that both patterns and styles specify a certain structure, e.g. the 'Layers' pattern/style decomposes system into groups of components at a particular level of abstraction and enforces communication rules. Their differences are the following:

- In the architecture patterns perspective, patterns specify a problem-solution pair, where problem arises in a specific context and a proven general solution addresses that problem. A context depicts one or more situations where a problem addressed by the pattern may occur. Moreover, the patterns capture common successful practice and at the same time, the solution of the pattern must be non-obvious [1].

- In the architecture styles perspective, styles are defined as a set of rules that identify the types of components and connectors that may be used to compose a system [18]. Architecture styles are more focused on documenting solutions in the solution domain [18]. The problem and the rationale behind a specific solution receive little attention [1].

These two schools of thought have more or less converged admitting that they are indeed referring to the same concepts [26] [34]. We concur with this trend. For the sake of simplicity, we shall use only the term 'architecture pattern' in this paper.

Modeling Architecture Patterns

Many researchers have focused on using the inherent as well as the extensible support of ADLs to model architecture patterns [2] [4] [6] [15]. Many of these ADLs focus on the use of components and connectors as architecture building blocks [13] and some provide built-in support to model patterns in software design. For instance, ACME supports templates that can be used as recurring patterns, Aesop allows pattern-specific use of vocabulary, and UniCon provides syntax and graphical icons support for a limited set of patterns. While describing architectures using ADLs, the architects mostly focus on the components as a central locus of computation for decomposing system functionality and use connectors as communication links between components. Furthermore, in an effort to bring ADLs closer to each other, some researchers are working with integrative approaches among ADLs [7], and among ADLs and UML [6]. However, these practices are still in an experimental phase, and there is yet no proven approach to model architecture patterns effectively. Unfortunately, the current practice of modeling architecture patterns is still un-systematic and ad-hoc.

3 Evaluation Framework

The framework elements defined in this section are used to assess the support offered by ADLs to model patterns. Four elements make up this evaluation framework: syntax, visualization, variability, and extensibility.

- Syntax. We define syntax as pattern-specific elements and rules that govern the modeling of architecture patterns e.g. grouping in Layers, communication links, topology in Client-Server, etc.
- Visualization. Graphical support for modeling patterns can be helpful in visual composition of pattern elements and graphical icons to represent pattern elements.
- Variability. Architecture patterns are characterized by an inherent variability, as they not only provide a unique solution to a problem but an entire solution space. The chosen variants in the different variation points affect the design, and quality attributes of the system. For instance, bypassing Layers in the layered pattern can affect maintainability. An important aspect of our work is to see how the variability in modeling patterns is addressed by ADLs.
- Extensibility. Discovery of new patterns and inclusion in the existing list of patterns requires extensibility of the ADLs. It is possible that the introduction of new patterns may entail new modeling elements, may introduce new constraints and rules etc. Therefore ADLs need to be extended to be able to model newly discovered patterns

4 Modeling Patterns in ADLs and UML

To evaluate the suitability of ADLs for modeling architecture patterns, we have selected UML [3] [6] and five ADLs for evaluation: ACME [7], Wright [8], Aesop [4], UniCon [21], and xADL [30]. Each of these languages provide unique support for modeling certain concepts of architecture patterns. UML provides explicit extensibility support for expressing pattern elements. ACME is used as an ADL and as an interchange platform between different ADLs and provides templates for capturing common recurring solutions. Wright provides enriched communication protocols. Aesop has a generic vocabulary of extensible architecture elements for expressing patterns. UniCon supports abstractions for a limited set of traditional architecture patterns. Finally, xADL uses XML tags and schemas to provide extensibility support for expressing pattern elements. The selection of these ADLs is based on: a) their popularity for designing software architectures [19]; b) their maturity for modeling patterns [16]; c) their capability for describing software architectures [5]; and d) their generalized nature and independence of specific domains.

We have selected four patterns for the evaluation: Layers, Pipe-Filter, Blackboard, and Client-Server. We selected these patterns because they are the most commonly used in practice and they represent a number of different domains and concerns. Layers demands grouping of components, Pipe-Filter handles streams of data, Client-Server is frequently used in distributed systems, and Blackboard is for dynamic configurations. Although we limit ourselves to only four patterns, we emphasize that our study is not meant to be exhaustive. However, an analysis of the most representative patterns is able to highlight the pros and cons of the different ADLs in modeling patterns. In the following sub-sections, we use the evaluation criteria defined in the previous section to evaluate each of these languages.

4.1 Syntax

UML: UML is intended as a modeling language for many different areas and it lacks considerably in expressing pattern elements. For instance, pipes in a Pipe-Filter pattern do not match with UML connectors, since UML connectors cannot have an associated state or interface. Such shortcomings can be solved through the extension mechanism of UML, where its metamodel can be extended with **profiles** to express architecture patterns. In specific, a UML profile is comprised of tagged values, metaclasses, and stereotypes, that may be defined to support pattern-specific syntax [6]. UML also provides explicit support through the Object Constraint Language (OCL) to express constraints for modeling pattern elements. Thus, to fully express most of the architecture patterns and to define interactions among pattern elements, the UML metamodel elements must be extended.

ACME: In addition to the core ontology of seven basic architecture design elements, ACME provides a template mechanism, which can be used for abstracting common reusable architectural idioms and patterns [7]. ACME allows defining user specified constraints on architecture elements to model patterns. Violations of these constraints are automatically checked by ACME studio. To apply constraints on architecture elements, ACME allows two kinds of rules specification: *invariants,* violations of which are errors and *heuristics,* violations of which generate warnings [15]. For instance, a *heuristic* rule can be defined to flag a warning message if a particular filter has more than two ports.

Wright: Rich connector support in Wright makes it a good option for patterns that heavily rely on connector and protocol specifications. Wright provides connector protocols as roles and glues, where glues can be used to define and constrain the behavior of interacting components. In Client-Server pattern, this role and glue specification for connectors allows constraining which clients can communicate to which server at architecture level. The glue specification can be used further to describe how clients and servers fit into the configuration [8] [17]. Furthermore, Wright constructs can be used in modeling dynamic systems [28], which is helpful in

Blackboard and Client-Server patterns. For instance, clients can be aware of the state of a server at run-time to use the services more efficiently [29].

Wright provides support for constraints checking with the use of accompanying tools. For instance, with use of the FDR tool [24], Wright syntax can be checked for deadlocks in Client-Server, cyclic graphs in Pipe-Filter, and compatibility checking, etc. However, Wright demands conversion of its description into CSP [24] first so that the CSP compatible tools [24] like FDR can work for automated checking.

Aesop: Aesop is a system for developing pattern-specific architecture design environments for specifying pattern elements, topology, and constraints, etc. [4]. It provides a generic list of seven elements (i.e. components, connectors, ports, roles etc.), which can be customized to represent pattern-specific elements. This customization is based on the principal of sub-typing: a pattern-specific vocabulary of design elements by providing subtypes of basic architecture elements [4]. For example, in the Pipe-Filter pattern, a port class can be sub-typed as Input and Output, and a role class can be sub-typed as Source and Sink. In addition, Aesop provides first class connector support, thus connectors can literally perform the same computation as done by components. This gives an advantage to Aesop in modeling patterns that require complex communications e.g. TCP/IP and Remote Procedure Call (RPC) in Client-Server.

UniCon: UniCon provides support for a limited set of *built-in* types of abstractions (i.e. specialized set of architecture elements) to represent pattern elements. In specific, UniCon supports connector abstractions of type Pipe, ProcedureCall, RPC, RealTimeScheduler, DataAccess, and PLBundler [21]. For instance, when modeling a Pipe-Filter pattern, the connector abstraction for the pipe provides support for specifying the number of connections, input ports, output ports, source roles, and sink roles, etc. Thus, only the existing abstractions available in UniCon can be used to specify constraints and to represent pattern elements. This makes it a weak option for modeling patterns, which are not supported by existing abstractions in UniCon.

xADL: xADL provides five XML (Extensible Markup Language) based *tags* to represent architecture elements, namely <Architecture>, <Component>, <Connector>, <ComponentType>, and <ConnectorType> [30]. xADL contains the inherent features of XML, which allow to extend tags for expressing pattern elements. Each tag can be enforced with pattern elements specific constraints. For instance, in a pipe-filter pattern, ComponentType defines nature of filter (e.g. message passing, data computation, data conversion etc.), and ConnectorType defines nature of pipe (e.g. input and output type of parameters). xADL supports type of connections using XML DTDs (Document Type Definitions) [33], which means different kinds of connections to express pattern elements can be used by specifying DTDs. Furthermore, these DTDs can be used to constrain the behavior of interacting pattern elements. For instance, a filter port can define the type of messages it receives using DTDs. Since tags in xADL represent general concepts to express architecture elements, manual work with these tags is required to fully express patterns.

Table 1 provides a brief description of the syntax support offered by each ADL for modeling patterns.

Table 1. Syntax Support for Patterns in the ADLs

Patterns ADLs	**LAYERS**	**PIPE-FILTER**	**BLACK BOARD**	**CLIENT-SERVER**
UML 2.0	**Strength:** Package metaclass support in UML can be used to group components **Weakness:** UML Aggregation, Composition and Package structure are not suitable to model all concerns of a layered pattern	**Strenght:** Connector metaclass in UML can be extended to express pipes **Weakness:** weak support for pipe representation	**Weaknes:** Connectors have fixed interfaces which affects dynamic configuration	**Strength:** UML profile can be extended to express client-server components and to define constraints on client-server topologies **Weakness:** UML profile in itself provides weak connector support for complex communication
ACME	**Strength:** ACME templates can be used to express grouping among components	**Strength:** Templates can be defined in ACME to express filters, pipes and data flow links	**Weakness:** Dynamic composition of components and connectors is weakly supported	**Strength:** Templates can be used to express client-server components and configuration constraints for defining communication links and topologies
Wright	**Strength:** Roles and glue specification can be used to express layered information flow constraints	**Strength:** Wright provides roles and glue support for expressing pipes and to define data flow connections among filters	**Strength:** provides constructs to describe dynamics of the components and provides events support to notify the state change of the components	**Strength:** Compatibility checking of clients and server is well supported, Deadlock detection is addressed by the use of roles and glues, allows complex topologies, reconfiguration supported, dynamism supported **Weakness**. Topological constraints not explicitly addressed
AESOP	**Strength**: Pattern-specific elements and constraints can be expressed by defining and extending sub-types of the generic elements: components, connectors, configuration, ports, roles, bindings, etc. **Weaknes***:* Configuration rules not very well supported for dynamic composition			

Patterns ADLs	**LAYERS**	**PIPE-FILTER**	**BLACK BOARD**	**CLIENT-SERVER**
UniCon	**Weakness***:* Fixed pattern elements specific abstractions is a problem to express layererd pattern specific constraints	**Strength**: Implicit abstractions support for expressing pipes and filters	**Strength**: Dynamic configuration and analysis supported **Weakness***:* Fixed set of abstractions to represent pattern elements is a problem to define *flexible* configuration rules	**Strength:** Rich abstractions to represent communication links supported e.g. connecor abstractions for procedure call, RPC, RealTimeScheduler for real-time communication, etc.
xADL	**Strength:** Grouping structure can be extended to express Layers	**Stength:** Tags can be extended to express pipes and filters	**Stength:** Dynamic configuration of architecture elements supported, Events can be used to inform the connected elements about the state change	**Stength:** A varity of communication protocols can be specified by specifying new kinds of DTDs and tags

4.2 Visualization

UML: A number of UML tools have been developed with explicit support for visual software designing e.g. IBM Rational Rose, Rational Software Architect, ArgoUML, etc. However, none of the tools developed for UML specifically focus on modeling architecture patterns. As a solution, few of the UML tools provide visual support to extend UML metamodel elements. For instance, Rational Rose allows user to create stereotypes, which are extensions to UML metaclasses, to model pattern elements. Still, UML tools are weak in providing explicit visualization support to model patterns and it largely depends on the way these UML tools are used to configure architecture elements for modeling patterns.

ACME: ACME has the advantage that with the introduction of ACME studio, which is an extension to Eclipse, it provides explicit visualization support to model specific patterns. The ACME studio editor provides three views: overview of the files in the project, textual source of the architecture and architecture diagrams with visibility of modeled patterns. To model specific patterns, ACME studio allows one to directly associate pattern elements with their corresponding architecture elements. For example, a component can be created by selecting a pattern type as filter, server, etc. In addition, ACME studio provides visualization support to view pattern elements at both abstract and detail level. For instance, a selected filter can be expanded to view its internal structure of pipes and filters.

Wright: Wright does not provide specific visualization support for modeling patterns.

Aesop: For visual modeling of patterns, Aesop supports a palette of pattern specific architecture elements and an *interface* that allows tools to manipulate architecture descriptions [4]. The graphical palette represents pattern-specific customized architectural elements for the modeling of architecture patterns. For example, pattern-specific graphical icons can be included in the palette e.g. pipes, filters, server, etc. In addition, Aesop stores architecture descriptions as objects in its object base and external tools can access this *object base* to provide visual editors for modeling patterns, creation and manipulation of objects, etc. [4]. Furthermore, Aesop provides a coloring scheme to identify mismatched connections. For instance, in a Pipe-Filter pattern, a color can be used to highlight incorrectly attached pipes [25].

UniCon: UniCon provides a specialized set of graphical icons to support traditional patterns like Pipe-Filter, Client-Server, etc. These graphical icons are provided in UniCon's default listing of component and connector types e.g. cloud for abstract binding, pipe, clock for real time communication, etc. For compatibility checking, UniCon provides graphical support to identify mismatched connections. For example, when a connection with mismatched signature is proposed, the editor facilitates including a connector that can translate the calling signature to the declared signature [21].

xADL: xADL benefits from associated XML compliant tools that can be used for visual description of software architecture (e.g. XSV [31] and XML Spy [32]). However, xADL does not provide specific visualization support for modeling architecture patterns and it mainly depends the way these tools are manually used by the architects to express architecture patterns.

Table 2 gives a brief description of visualization support offered by each ADL for modeling patterns in general.

Table 2. Visualization Support for Patterns in ADLs

UML 2.0	**Strength***:* UML tools support visual composition of components and connectors, which can be used for modeling specific concepts of architecture patterns **Weakness***:* UML does not provide explicit support for modeling architecture patterns
ACME	**Strength:** ACME studio provides explicit visualization support to model few selected patterns
Wright	**Weakness:** No specific visualization support provided for modeling architecture patterns
AESOP	**Strength:** Pattern-specific architecture elements with distinctive colors can be visually created. Pattern elements can be composed for modeling specific patterns.
UniCon	**Strength:** For a specific list of patterns, UniCon provides good graphical support for modeling patterns and to convert graphical diagrams into textual description

	Weakness: UniCon provides graphical support for modeling only few patterns.
xADL	**Weakness:** No specific visualization support provided for modeling architecture patterns

4.3 Variability

UML: Fixed interfaces, weak connector support and lack of explicit support to express pattern elements is a problem for modeling variability in patterns. Extending UML, as discussed in previous sections, is an explicit way to model pattern variability. However, even the extension to UML metamodel can address a limited variability in patterns. First, because pattern variability at detail level of design is not addressed at a higher level of abstraction to represent architecture elements as done in UML. Secondly, OCL constraints need to be explicitly addressed for each specific variability issue for modeling patterns. For instance, an OCL constraint restricting no more than two ports attached to a filter will always fail in the operation to add a third port to a specific filter and some sort of extension to OCL description is required.

ACME: ACME defines a weak typing system with a fixed set of types e.g. seven architectural elements of its core ontology and data types of Integer, Boolean, and String [11]. This provides ACME both an advantage and disadvantage in modeling patterns variability. An advantage is that being a standard interchange platform between ADLs, ACME provides a generalized support to represent architecture elements, which is extensible to model variability in patterns. For instance, a filter in Pipe-Filter pattern resembles a generic ACME component with input and output ports. This allows using a filter in all required contexts by considering it as a mere component endowed with the properties of a filter. However, this flexibility in the language has a negative impact on the analysis of modeled variability as no explicit type checking support is provided in ACME.

Wright: Flexible glue specification provides support for modeling pattern-specific variability. The glue specification for connectors allows pattern elements of same type to be represented as logically separate type of entities. For instance, each pipe in a pipe-filter pattern can express its own glue specification to connect with filters. Therefore, a pipe can be connected on one end to a filter and on the other end to a file, while other pipes in the same chain may be connected on both ends to filters. This strong representation of connections among architectural elements gives advantage to Wright in modeling specific variability by providing each architectural connection specific glue specification. Furthermore, rich specification of connector allows distinctively identifying variants of connectors e.g. pipes, procedure calls, etc.

Aesop: As discussed in previous sections, Aesop facilitates creation of *environments* to define patterns. These pattern definitions are compiled during environment creation time. While modeling patterns in software design, it does not support any kind of variability, which is not included in the original definition of the pattern. For instance, in the pipeline pattern, a filter is always initialized with only one input and one output port. A variability requirement to add a fork in pipeline will always fail in adding a new port. Furthermore, pattern-specific customization of classes requires architect to handle variability constraints at its own with least help from language.

UniCon: UniCon provides a limited set of abstractions to represent pattern elements and connections. This puts a huge constraint in modeling specific type of variability in UniCon as it allows representing connections from only existing types of abstractions. For instance, a procedure call can be replaced with a different connection from only available types of connections.

xADL: xADL defines schemas named: options (optional components, connectors, and links), variants (variant component and connector types), versions (versions in the form of graphs for components, connectors, and interfaces) [30]. These features supported by each schema can be used to model limited variability in patterns. The use of options and variants gives architects freedom to specify pattern elements of different types (e.g. different types of filters) in a single xADL document, and then instantiate any of the pattern elements during architecture design. Furthermore, xADL supports a programming language style type system for specifying pattern elements [30]. Thus, architects can define different variants of the pattern elements as types of component, connector, and interfaces. For instance, a filter type can be extended to specify one or more filters with different properties.

4.4 Extensibility

UML: UML is considered weak to represent elements of architecture patterns, which is a drawback to model new patterns as well. However, UML's metamodel can be extended to model new patterns. Medvidovic et al. [6] provides UML extension mechanism with the use of UML metaclasses, which can be effectively used to provide extensibility support to model new patterns. For instance, new stereotypes can be created and constraints specific to new patterns can be applied on these stereotypes.

ACME: ACME allows templates to specify recurring patterns, which is helpful in modeling patterns that come-up even with new syntax definitions. These templates are quite flexible supporting new definition of components and connectors. Furthermore, it allows defining new constraints for interaction among components. However, defining architecture elements in ACME requires following the typing discipline applied in ACME as discussed in previous sections. Its typing discipline with a fixed set of data types has the disadvantage that it does not support connections that require new data types.

Wright: Enriched connector support and flexible properties specification makes Wright a preferable extensible language to model new patterns that heavily rely on communication specification. For instance, the Remoting Error pattern [29] can benefit from glue and protocol specification to detect and handle network failures, server crashes, and un-reliable networking objects, etc.

Aesop: Aesop provides a generic list of elements that can be customized to fulfill the requirements to model new patterns. The principal of sub-typing introduced in Aesop can be used to express new pattern elements as sub-types of generic architecture elements. This makes Aesop an attractive option to model new patterns by defining new pattern specific design environments.

UniCon: UniCon provides support for only built-in component types like module, computation, shared data, filter, process, general etc., and built-in connector types like Pipe, ProcedureCall, DataAccess, etc. [21]. It specifies type 'general' for all other types of components that are not supported by it and provides no extension facility to specify new kind of connectors. This puts a huge constraint on modeling new patterns that demand new compositional elements. The benefit that UniCon offers by providing implicit support for modeling few patterns is questioned by its rigidness to support new type of components and connectors.

xADL: xADL, also called 'extension ADL' [33], shows high promises for extensibility to express newly discovered architecture patterns. xADL use of schemas supports extension to express new types of components, connectors, interfaces, connections and configuration rules. Similar to UML stereotyping extensions described in previous sections, xADL supports extensibility by new tags and attributes. However, extension mechanism of XML itself imposes some restrictions to express pattern elements as it offers a weak support in applying constraints on new pattern elements.

5 Related Work

The idea to compare ADLs for their suitability to design software architectures has already been investigated from different viewpoints [5], [19], etc. However, none of the approaches presented so far have specifically focused on comparison of the ADLs for their support to model architecture patterns. Most of the work to date, has focused on the use of mere components and connectors to design software architecture, neglecting the pattern rules for the composition of architecture elements. In our work, we have specifically focused on modeling patterns in few selected ADLs to analyze their support to model patterns.

Medvidovic et al. [5] provide a comparison framework to compare architecture modeling features and tool support offered by a number of ADLs. Their work is focused on components, connectors and their configuration. They highlight the inconsistency with which different ADLs specify semantics to configure components and connectors, and the problems for specifying non-functional properties. Our work is complimentary to this general survey of ADLs, as we focus on the use of patterns to design software architecture.

Shaw et al. [18] analyze patterns for their topology, configuration, data, and control issues. Their work is based on the feature selection among patterns to guide the architects to choose a pattern that is best suited to solve the *problem* at hand. The framework they propose accommodates patterns in the categories of communicating processes and dataflow networks. They also specify association of specific patterns with their description languages. However, their work is more focused on the selection of patterns to solve the *problems*, with little attention on challenges to model these patterns in ADLs. Our work is different in the sense that we specifically focus on patterns to relate them with different ADLs to provide a comparison among ADLs for their support in modeling patterns.

In our previous work [2], we have used architecture primitives as an extension to UML metamodel elements to model patterns. Although this work is focused on UML 2.0, the same approach can be used for other ADLs as long as the selected ADL supports the extension mechanism to handle the semantics of the primitives. The key idea in this approach is that the languages that can be extended to facilitate syntactic and semantic of architecture primitives can be used to model pattern variability.

6 Conclusion and Future Work

We have evaluated a few selected ADLs for their support to model architecture patterns. An evaluation framework that looks into syntax, visualization, variability, and extensibility was used to serve this purpose. We find that most of the ADLs specify strong notational, analysis and tool support to design software architectures. Furthermore, some of these ADLs provide inherent support to model patterns but at a detailed level, nearly all of the ADLs fail to capture the rich concepts found in patterns. Furthermore, ADLs differ largely in their scope to model patterns. Few ADLs are popular for modeling patterns due to their specialized nature for providing abstraction support to represent pattern elements. However, none of the ADLs deal with the variability issues for modeling patterns in general. For each ADL discussed in this paper, some of the strong and weak points were highlighted for their support to model patterns.

UML claims to be a standard design language and provides good tool support, but in many aspects, we find UML to lack significantly from other ADLs in modeling patterns. Specifically, the pattern elements do not match with the UML notations to design software architecture and some sort of UML extension is required to fill this gap.

We find extension mechanism for some of the ADLs as an effective way for modeling patterns. ADLs, like UML and Aesop, provide a generic list of customizable elements to express pattern specific elements. However, the shortcoming of this approach stems from the use of the ADLs itself. Specifically, the extension mechanism of UML is awkward to use because the extended classes are neither a part of metamodel nor are they model elements [2].

Other than simple pattern representation, ADLs are weak for their accompanying visualization and tool support. Some languages like Aesop, Wright, and UniCon provide tools for type and constraint checking, but their support is limited for the specific use of tools, such as FDR for Wright and RMA for UniCon.

Our work leaves several open questions in modeling patterns with the existing ADLs. It highlights the need for a paradigm to model patterns that can work independently of hard rules inherent in ADLs. In addition, ADLs differ extensively in their syntax and graphical notations and ADLs still lack the presence of a widely accepted generalized vocabulary of elements for modeling patterns.

References

1. Paris Avgeriou and Uwe Zdun: Architectural Patterns Revisited - A Pattern Language, In proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLOP), pp. 1-39, Irse, Germany, (2005)
2. Paris Avgeriou and Uwe Zdun: Modeling Architecture Patterns using Architecture Primitives, OOPSLA' 05, ACM (October 2005)
3. Morgan Bjorkander and Cris Kobryn: Architecting Systems with UML 2.0, IEEE Computer Society, 0740-7459/03, IEEE (July 2003)
4. David Garlan, Robert Allen and John Ockerbloom: Exploiting Style in Architectural Design Environments, In Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering, New Orleans, LA (December 1994)
5. Nenad Medvidovic and Richard N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, no. 1, (January 2000)
6. Nenad Medvidovic, David S. Rosenblum, David F. Redmiles and Jason E. Robbins: Modeling Software Architectures in the unified modeling language, ACM Transactions on Software Engineering and Methodology, vol. 11, no. 1, pp. 2-57, (January 2002)
7. David Garlan, Robert Monroe and David Wile: ACME: An Architecture Description Interchange Language, Proceedings of CASCON 97, Toronto, Ontario, pp. 169-183, (January 1997)
8. Robert Allen and David Garlan: A Formal Basis For Architectural Connection, ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pp. 213-249, (July 1997)
9. Michael Kircher and Prashant Jain: Pattern-Oriented Software Architecture, Volume 3, Wiley Series in Software Design Patterns, ISBN 0-470-84525-2
10. Robert T. Monroe, Andrew Kompanek, Ralph Melton and David Garlan: Architectural Styles, Design Patterns and Objects, Carnegie Mellon University, IEEE Software, 0740-7459/97, (January 1997)
11. Robert Allen and David Garlan: A Case Study in Architectural Modelling: The AEGIS System, Computer Science Department Carnegie Mellon University, Pittsburgh, PA 15213
12. Richard P. Draves, Michael B. Jones and Mary R. Thompson: MIG – The Mach Interface Generator, Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, 15213 (November 1989)
13. Eoin Woods and Rich Hilliard: Architecture Description Languages in Practice Session Report, Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pp. 243-246, (November 2005)
14. The 1998 Pattern Languages of Program Conference, August 11-14 1998, Monticello, Illinois, USA
15. ACME Studio, http://www.cs.cmu.edu/~acme/AcmeStudio/tutorials.html
16. Architecture Description Languages – A technology roadmap, http://www.sei.cmu.edu/str/descriptions/adl.html
17. Robert Allen and David Garlan: Formalizing Architectural Connection, 16th International Confernence on Software Engineering, Sorrento, Italy, 0270-5257/94, IEEE (May 1994)
18. Mary Shaw and Paul Clements: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, Computer Science Department and Software

Engineering Institute, Carnegie Mellon University, Pittsburgh, IEEE, 0730-3157/97, PA 15213 (1997)

19. Paul C. Clements: A Survey of Architecture Description Languages, Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD’96), 1063-6765/96, IEEE (1996)
20. Mary Shaw: Some Patterns for Software Architectures, Pattern Languages of Program Design, pp.255-269, Vol. 2, Addison-Wesley Longman Publishing Co., Inc, MA, USA, ISBN 0-201-895277, (1996)
21. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik: Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, Vol. 21, no. 4, pp. 314-335, (April 1995)
22. Nikunj R. Mehta, Nenad Medvidovic and Sandeep Phadke: Towards a Taxonomy of Software Connectors, Proceedings of the 22nd international conference on Software engineering, pp. 178-187, Limerick, Ireland, ISBN 1-58113-206-9, (2000)
23. Tommi Mikkonen: Formalizing Design Patterns, Proceedings of the 20th international conference on Software engineering, Kyoto, Japan, pp. 115-124, ISBN 0-8186-8368-6, (1998)
24. Tools for CSP, http://web.comlab.ox.ac.uk/oucl/publications/books/concurrency/tools
25. Ralph Melton: The Aesop System: A Tutorial, The Able Project, Carnegie Mellon University, Pittsburgh PA, 15213
26. Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal.: Pattern Oriented Software Architecture: A System of Patterns, John Wiley & Sons, ISBN 0 471 95869 7
27. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley Professional Computing Series (1995)
28. Robert Allen, Remi Douence and David Garlan: Specifying and Analyzing Dynamic Software Architecture, Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal, (March 1998)
29. Markus Volter, Michael Kircher and Uwe Zdun: ‘Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware, Wiley Series in Software Design Patterns, ISBN 0-470-85662-9, (2004)
30. Eric M. Dashofy, Andre van der Hoek and Richard N. Taylor: A Highly-Extensible, XML-Based Architecture Description Language’, Department of information and computer science, University of California Irvine, CA 92697, USA
31. World Wide Web Consortium, Validator for XML scchemas,http://www.w3.org/2000/09/webdata/xsv, September, 2000.
32. Altova GmbH: XML Spy Software, http://www.xml-spy.com, January, 2001
33. Zhang Jingjun, Zhang Yang and Li Furong: Combinatorial Model and Aspect-Oriented Extension of Architecture Description Language, 0-7803-8932-8, IEEE, (2005)
34. Len Bass, Paul Clements and Rick Kazman: Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, ISBN 0321154959, (2003)