

ADLs and Dynamic Architecture Changes

Nenad Medvidovic

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425
nen@ics.uci.edu

Abstract

Existing ADLs typically support only static architecture specification and do not provide facilities for the support of dynamically changing architectures. This paper presents a possible solution to this problem: in order to adequately support dynamic architecture changes, ADLs can leverage techniques used in dynamic programming languages. In particular, changes to ADL specifications should be interpreted. To enable interpretation, an ADL should have an architecture construction component that supports explicit and incremental specification of architectural changes, in addition to the traditional architecture description facilities. This will allow software architects to specify the changes to an architecture *after* it has been built. The paper expands upon the results from an ongoing project -- building a development environment for C2-style architectures.¹

I. Introduction

Architecture description languages (ADLs) are the means by which software architectures are defined. ADLs enable software architects to express high level system structure by describing its coarse-grained components and connections among them. Such specifications reduce the cognitive load on designers and enable system-level analysis and code generation. At the same time, current ADLs are static. They support a linear process in which an architecture is specified and then “compiled” into a running system. In general, existing ADLs do not provide any support for dynamically changing architectures.

This paper briefly motivates the need for dynamic architectural changes. It then provides a closer examination of the current inability of existing ADLs to support such changes. Finally, it proposes a possible solution to this problem. This solution is based on both past experiences and lessons learned from an ongoing project.

II. Dynamism in Software Architectures

The promise of software architectures is that they will reduce the costs and improve the quality of software development by enabling

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

component-based construction of large-scale software. Architectures shift the focus of developers from lines of code and one-of-a-kind systems to coarse-grained components and system families. Explicit focus on architectures gives developers more flexibility, as reflected in the potential for component reuse and substitutability.

Another aspect of flexibility, and desirable trait of software architectures, is dynamism. Dynamic architectures can be modified after the system has been built. Being able to do so at such a high level of abstraction gives developers a lot of power and provides extensibility, customizability, and evolvability of large software systems. Furthermore, dynamic changes of software architectures are essential to the evolution of long running and mission-critical systems.

Some common ways in which an architecture may be modified after it has been built are:

- addition of new components,
- upgrading existing components (e.g., performance tuning),
- removal of unnecessary components (both temporary and permanent),
- reconfiguration of application architecture (reconnection of components and connectors), and
- reconfiguration of system architecture (e.g., modifying the mapping of components to processors).

III. Characteristics of Existing ADLs

ADLs are formalisms that allow specification of architectural structure and its operational semantics. ADLs can also specify other aspects of an architecture, such as communication protocols, design rationale, and the mapping from components in the architecture to source code modules that implement them. ADLs are not very well understood and there is no consensus in the research community on what is and is not an ADL [6]. [3] recognizes that an ADL can have elements of CASE environments, programming languages, and requirements specification languages. Therefore, ADLs are currently used in a variety of ways: some are simply a convenient design notation, while others serve as simulation, and even architecture programming, languages.

An ADL is usually accompanied by a set of tools that assist the developers in generating a running system from an architecture. The level of assistance provided by such tools varies: some support automatic code generation from architectures (e.g., MetaH [1] and LILEANNA [11]), while others (currently) do not provide any support beyond design (e.g., ArTek [9]). Even in those environments that support code generation, the level of support will differ. Also, the details of the generation process may vary across ADLs, architectural styles, and toolsets. For example, some may require a 1-to-1 structural correspondence between an architecture and the resulting system, while others (e.g., the C2 style [10]) allow multiple conceptual components to be implemented by a single off-the-shelf (OTS) source module and vice versa [5].

ADLs typically provide a static description of an architecture. They provide a snapshot of the architecture *before* the system is built and generally lack facilities for changing that architecture at run-time. Rapide’s **where** connection conditions [4] and LILEANNA’s **make** statements [11] can be viewed as (rare) exceptions to this. However, even in these cases, all possible dynamic changes must be known and programmed into the architecture beforehand. This static approach to ADLs is also reflected in the common perception of ADL specifications as something that is always compiled (e.g., [2] and [8]).

The remainder of this paper examines some of the issues that must be addressed by ADLs and their accompanying tools in order to support dynamism in architectures and particularly the types of dynamic architectural changes discussed in Section II. The paper assumes an idealized architecture-based development process: an architecture specified in an ADL is input into the ADL’s development toolset for automated system generation. The above-discussed issues of the level of support for and the exact nature of code generation are important, but are beyond the scope of this paper.

IV. Role of ADLs in Dynamic Architectural Changes

In order for designers of ADLs to adequately support dynamically changing architectures, they should try to leverage techniques that have already been successfully applied to dynamic programming languages and, at the same time, avoid their drawbacks (large executables, inadequate performance, etc.). In particular, while architectures specified in ADLs are compiled, run-time changes to those architectures should be interpreted. Since those changes are occurring at a level of granularity above programming language statements, it is possible to interpret them but still compile the appropriate source code into the executable version of the system [7].

Another technique, particularly useful when system resources are limited, is “just-in-time” component loading. Just-in-time loading can be viewed as a generalization of the problem of dynamic architecture modification: if we can load components during execution because of limited system resources, we can also do it when the architecture itself changes.

Another issue, and the particular focus of this paper, is the set of features an ADL should have to support dynamically changing architectures. Existing ADLs typically foster declarative specifications of the overall architectures. Any (local) changes to an architecture are performed in the context of this complete definition and it is the definition itself that is manipulated. This means that, in order to determine which part of the architecture has changed, the entire architecture must be recompiled/reinterpreted after the simplest of modifications.

The entire system does not necessarily have to be generated anew in this case. With the appropriate tool support, it may be possible to isolate the modified parts of the architecture, interpret them, and perform the necessary changes to the running system. Such an approach has several drawbacks, however. The architect acts directly on the architecture specification (the data structure) itself; there is no abstraction or information hiding.² In addition, the approach may be slow if the architecture is very large and is modified, and subsequently reinterpreted, many times. Finally, it is error-prone. If, for example, a large number of components in the architecture must be modified to use event registration, it is up to the architect to manually alter the specification of each component’s communication protocol.

2. A possible exception to this are ADLs, such as MetaH [1] and UniCon [8], which provide a graphical, in addition to the textual, notation. Though equivalent, the graphical notation can be thought of as being at a higher level of abstraction than its textual counterpart.

A solution preferable to this is to extend the ADL with architecture construction notation (ACN). Such a facility would allow the architect to explicitly specify the changes made to the architecture, making those changes more suitable for interpretation. It would provide a kind of an application programmable interface (API) to the architecture specification. Note that the ADL would still allow for direct manipulation of the architecture specification, described above.

In a recent project, we implemented a simplified version of C2 SADL, an ADL (with its accompanying ACN) for architectures built according to the C2 style [10].³ The discussion of how C2 SADL can express dynamic changes in C2-style architectures follows. For the purpose of this discussion, the assumption is made that all components and connectors adhere to C2 style rules and that any OTS components or interconnection technologies will be appropriately modified (“wrapped”) to adhere to those rules.

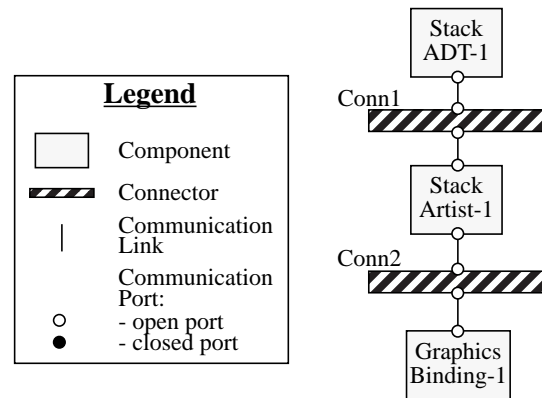


Fig. 1. Simple stack visualization architecture.

The simple stack visualization architecture shown in Fig. 1 is used for illustration purposes. This architecture is built according to C2 style rules and is expressed below in the simplified C2 SADL.

```

architecture StackArchitecture is {
  components {
    StackADT1;
    StackArtist1;
    GraphicsBinding1;
  }
  connectors {
    Conn1;
    Conn2;
  }
  topology {
    connector Conn1 connections {
      top_ports {
        StackADT1 filter no_filtering;
      }
      bottom_ports {
        StackArtist1 filter no_filtering;
      }
    }
    connector Conn2 connections {
      top_ports {
        StackArtist1 filter no_filtering;
      }
      bottom_ports {
        GraphicsBinding1 filter no_filtering;
      }
    }
  }
}

```

3. SADL is pronounced “saddle” and stands for Software Architecture Description Language. The main features of C2 SADL are described in [5] and [6].

A. Addition of New Components

A component is added to the StackArchitecture and placed between its two connectors after the system has been built as follows:⁴

```
StackArchitecture.AddComponent (StackArtist2);
StackArchitecture.Weld (Conn1, StackArtist2);
StackArchitecture.Weld (StackArtist2, Conn2);
```

The message filtering policy can optionally be specified as the third parameter of the **Weld** function. The default filtering policy is **no_filtering**. The resulting architecture is shown in Fig. 2.

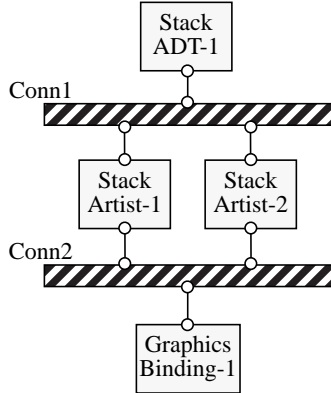


Fig. 2. A second stack visualization component is added to the architecture.

B. Removal of Unnecessary Components

A component in a C2-style architecture is effectively removed from the architecture by “turning off” the ports of the connectors to which the component is attached. In other words, the ports’ filtering policies are changed to **message_sink**. The component is not physically removed from the architecture at this point, even though its functionality is unreachable from the rest of the architecture. For example, StackArtist1 is disconnected from the two connectors (Fig. 3a) in the following manner:

```
Conn1.SetBottomPortFilter (StackArtist1, message_sink);
Conn2.SetTopPortFilter (StackArtist1, message_sink);
```

To “reattach” the component back to the architecture, the appropriate Conn1 and Conn2 port filters are simply set back to **no_filtering**. The component is physically (permanently) removed from the architecture (Fig. 3b) as follows:

```
StackArchitecture.Unweld (Conn1, StackArtist1);
StackArchitecture.Unweld (StackArtist1, Conn2);
StackArchitecture.RemoveComponent (StackArtist1);
```

C. Upgrading Existing Components

An existing component in a running system can be upgraded by

1. subtyping from the component,
2. modifying the subtype,
3. adding the subtype to the architecture, and
4. removing the original component from the architecture.

C2 SADL can support all four steps. The issues in subtyping C2 components, as well as the relevant C2 SADL features, are presented in [5]. Adding and removing components in an architecture was demonstrated in the preceding subsections. This approach permits us to keep the original component running, if needed, while it is being upgraded.

4. The ACN notation shown in this paper is imperative. We believe that this is a more natural way of expressing the desired modifications. However, this can be as readily accomplished by using an equivalent declarative notation.

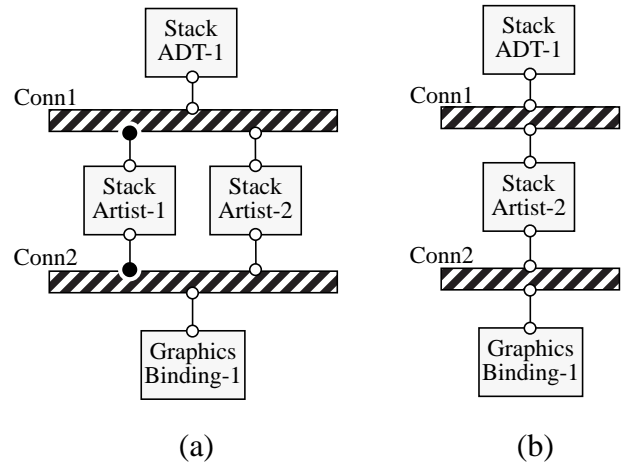


Fig. 3. A component is removed from an architecture either (a) temporarily, by blocking message traffic at the ports to which it is attached, or (b) permanently, by physically removing it from the architecture.

D. Reconfiguration of Application Architecture

Components and connectors in an architecture can be “rewired” simply by using the **Weld** and **Unweld** functions of the ACN. If a large portion of the architecture needed to be reconfigured, it may actually be more effective to specify the new architecture in the traditional manner (i.e., by using the declarative part of the ADL). Since C2 SADL supports both ways of modifying an architecture, this decision is left up to the architect.

V. Conclusion

The problem of dynamically changing architectures is a difficult one. Specific ADL features can alleviate some aspects of this problem. Currently existing ADLs only provide means for declaratively specifying the structure of an architecture. As such, they are not well suited to support dynamic architectural changes. One possible solution is to provide an architecture construction facility (ACN) in an ADL. Coupled with ACN interpretation and appropriate code generation tools, this is a promising approach.

An important set of issues in architectural dynamism is trying to establish under what circumstances it is safe to remove and/or add a component to an architecture, change the filtering policy on a connector port, and “rewire” the architecture. Some problems inherent in doing this are potential loss of messages if a component is removed during processing, establishing the initial state of a component added during system execution, and ensuring the well-formedness of the newly-created architecture. However, these problems will most likely be handled by the architecture’s analysis tools and its run-time system. As such, they are outside the scope of an ADL and have thus not been discussed in this paper. A more in-depth treatment of these issues is given in [7].

Another issue that is beyond the scope of this paper is the role of a particular architectural style in facilitating dynamic architecture changes. On the one hand, the ability to express such changes in an ACN is independent of a style. On the other hand, some styles may be more amenable to enacting those changes at runtime than others. Our particular focus in software architectures is on C2, a message-based style. We have had initial success in using the approach discussed in this paper with C2-style architectures. Clearly, additional experiments are needed to establish its applicability to other styles, but we expect this initial experience to be of great value to us and other researchers attempting to do so.

VI. Acknowledgements

Thanks to Dick Taylor for his insightful comments on an earlier draft of the paper. Special thanks to Peyman Oreizy for his work on dynamic software architectures and, in particular, his help with several aspects of this work.

VII. References

- [1] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. Submitted to *IEEE Transactions on Software Engineering*, January 1994.
- [2] D. Garlan, R. Monroe, and D. Wile. ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219, Carnegie Mellon University, November 1995.
- [3] P. Kogut and P. Clements. Features of Architecture Description Languages. Draft of a CMU/SEI Technical Report, December 1994.
- [4] D. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.
- [5] 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: The Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 16-18, 1996.
- [6] 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 17, 1996.
- [7] Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report, UCI-ICS-96-35, University of California, Irvine, August 1996.
- [8] 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.
- [9] A. Terry, R. Hayes-Roth, L. Erman, N. Coleman, M. Devito, G. Papanagopoulos, and B. Hayes-Roth. Overview of Teknowledge's Domain-Specific Software Architecture Program. *ACM SIGSOFT Software Engineering Notes*, pages 68-76, October 1994.
- [10] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.
- [11] W. Tracz. Parameterized Programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing SAC'93*, February 1993.