An Architecture-Based Approach to Software Evolution

Nenad Medvidovic, Richard N. Taylor, and David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425, U.S.A.
{neno,taylor}@ics.uci.edu

1. Introduction

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are as high as 60% of the overall development costs [6]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus supporting only limited reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

In this paper, we posit that an explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability, for two reasons:

- connectors are often not treated explicitly or, when they are, they are too rigid and do not accommodate modification of their attached components easily; and
- no specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [13]. Each of them may evolve. Our work to date has focused on the evolution of individual components and architectural configurations. In the future, we intend to investigate the proper techniques for evolving connectors. For evolving individual components, our approach expands the traditional techniques for supporting evolution (e.g., modularity, typing). We introduce explicit, flexible connectors to aid the evolution of architectural configurations.

The following section discusses our approach to component evolution and introduces an architectural type theory on which the approach is based. Section 3 discusses the role of software connectors in the evolution of architectural configurations. Conclusions and a discussion of ongoing work round out the paper.

2. Component Evolution

Researchers in software architectures, and particularly in architecture description languages (ADLs), can learn from extensive experience in the area of PLs. For example, an existing software module can evolve in a controlled manner via subtyping. Our approach to component evolution is indeed based on type theory. We treat each component in an architecture as a type and support its evolution via subtyping. However, while PLs (and several existing ADLs [4, 5, 7]) support a single subtyping method, architectures may require multiple subtyping methods, many of which are not commonly supported in PLs. Therefore, an extension to PL type theory is needed.

A useful overview of PL subtyping mechanisms is given by Palsberg and Schwartzbach [15]. They describe a consensus in the object-oriented (OO) typing community regarding the definition of a range of OO typing mechanisms. *Arbitrary subclassing* allows any class to be declared a subtype of another, regardless of whether they share a common set of methods. *Name compatibility* demands that there exist a shared set of method names available in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass must preserve the interface of the superclass. *Behavioral conformance* allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. *Strictly monotone subclassing* also demands that the subtype preserve the particular implementations used by the supertype.

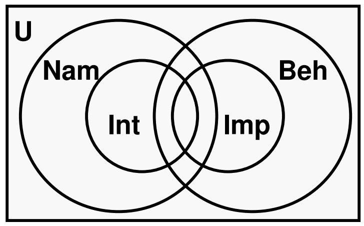

Fig. 1. A framework for understanding OO subtyping mechanisms as regions in a space of type systems.

We have developed a framework for understanding these subtyping mechanisms as regions in a space of type systems, shown in Fig. 1. The entire space of type systems is labeled *U*. The regions labeled *Int* and *Beh* contain systems that demand that two conforming types share interface and behavior, respectively. The *Imp* region contains systems that demand that a type share particular implementations of all supertype methods, which also implies that types preserve the behavior of their supertypes. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance. Each subtyping mechanism described in [15] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the Int and Beh regions and is expressed as *int* **and** *beh*.

We have demonstrated the utility of such a flexible subtyping mechanism in our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [10, 12]. Several examples are shown in Fig. 2.

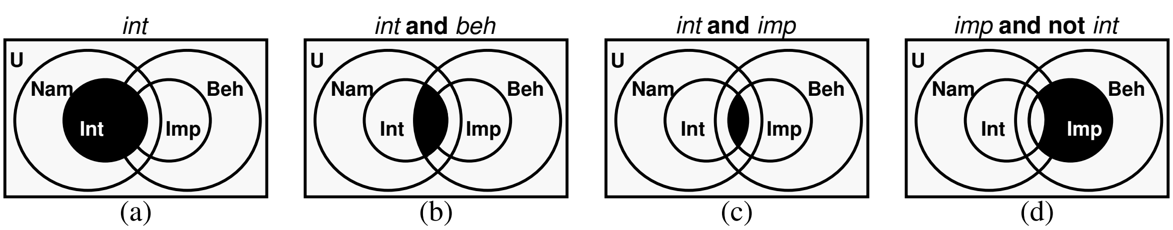

Fig. 2. Examples of component subtyping mechanisms we have encountered in software architectures. *(a)* interface conformance; *(b)* behavioral conformance; *(c)* strictly monotone subclassing; *(d)* implementation conformance with a different interface (e.g., software adaptors [18]).

2.1. Architectural Type Theory

In [10] we discussed the types of syntactic constructs needed in an ADL in order to support this approach. In this section we present a brief overview of the underlying type theory. The two possible applications of an architectural type theory are type checking of architectural descriptions and evolution of existing components by software architects. Each is briefly discussed below.[1]

1. We omit some details for simplicity.

Every component is an *architectural type*. An architectural type, AT, has a name, a set of interface elements, an associated behavior, and (possibly) an implementation:

$$AT = \langle nam, int^*, beh, imp \rangle$$

Each interface element has a direction indicator (provided or required), a name, and a set of parameters. Each parameter, in turn, has a name and a type.

$$int = \langle dir, int\_nam, param^* \rangle$$
$$param = \langle param\_nam, param\_type \rangle$$

A component's behavior consists of an invariant and a set of operations. The invariant is used to specify any protocol constraints on the use of the component. Each operation has (a set of) preconditions and postconditions and (possibly) a result.

$$beh = \langle inv, oper^* \rangle$$
$$oper = \langle pre, post, result \rangle$$

Finally, since we separate the interface from the behavior, we define a function, f, that maps every interface element to an operation of the behavior. This function is "onto": each operation exports at least one interface.

A subtyping relationship, $\leq$, between two components, C_i and C_j, is defined as a conjunction of *nam*, *int*, *beh*, and *imp* relationships (see Fig. 1):

$$(\forall C_i, C_j : AT)(C_j \leq C_i \;\Leftrightarrow\; C_j \leq_{nam} C_i \;\vee\; C_j \leq_{int} C_i \;\vee\; C_j \leq_{beh} C_i \;\vee\; C_j \leq_{imp} C_i)$$

We consider *int* and *beh* subtyping relationships in more detail below. *Nam* is a trivial relationship; we have encountered it in practice only as part of the stronger *int* relationship. Similarly, although useful in practice for evolving components, *imp* is not a particularly interesting relationship from a type-theoretic point of view. Implementation conformance can be established with a simple syntactic check: the operations of the subtype must have the same implementation as the corresponding operations of the supertype.

Component C_j is an *interface subtype* of C_i if and only if it provides at least (but not necessarily only) the interface elements provided by C_i with identical names and direction indicators, and *matching* parameters for each interface element. Two parameters belonging to the two components' interface elements match if and only if they have identical names and the parameter type of C_i is a subtype of the parameter type of C_j (*contravariance of arguments*). Note that, as with interface elements, the subtype must provide at least (but not necessarily only) the parameters that match the supertype's parameters:[2]

$$\begin{aligned}
&(\forall C_i, C_j : AT)(C_j \leq_{int} C_i \;\Leftrightarrow\; (\forall M \in C_i.int)(\exists N \in C_j.int) \\
&\quad ((M.dir = N.dir) \;\wedge\; (M.int_nam = N.int_nam) \;\wedge\; ((\forall P_m \in M.param)(\exists P_n \in N.param) \\
&\quad\quad ((P_m.param_nam = P_n.param_nam) \wedge (P_m.param_type \leq P_n.param_type))))
\end{aligned}$$

A *behavioral subtyping* relationship between two components is specified as follows:

$$\begin{aligned}
&(\forall C_i, C_j : AT)(C_j \leq_{beh} C_i \;\Leftrightarrow\; (C_j.beh.inv \Rightarrow C_i.beh.inv) \;\wedge \\
&\quad (\forall P \in C_i.beh.oper)(\exists Q \in C_j.beh.oper) \\
&\quad\quad ((P.pre \;\Rightarrow\; Q.pre) \wedge (Q.post \;\Rightarrow\; P.post) \;\wedge\; (Q.result \leq P.result))
\end{aligned}$$

This definition requires that invariant of the supertype be ensured by that of the subtype. Furthermore, each operation of the supertype has a corresponding operation in the subtype, where the subtype's operation has the same or weaker preconditions, same or stronger postconditions, and the type of its result is a subtype of the supertype's result type (*covariance of result*).

The subtyping relationship expressed by the combination of these two definitions and the mapping function, f, results in the region depicted in Fig. 2b and is similar to other researchers' notions of behavioral subtyping (e.g., America [2], Liskov and Wing [8], Leavens et al.[3]). However, in these approaches type correctness is characterized as either legal or illegal. In software architectures, various *degrees* of type conformance may be acceptable, so that, for example, the interfaces of two communicating components may match up only partially. Additionally, by separating interface from behavior (and adding the

2. In our notation, "X.Y" denotes X's constituent Y. For example, "C_i.int" denotes component C_i's interface.

nam and *imp* relationships), we give a software architect more latitude in choosing the direction in which to evolve a component. Such a flexible type system allows some potentially undesirable side effects (e.g., a supertype and its subtype may not always be interchangeable in a given architecture). However, it is left up to the architect to decide whether he wants to preserve architectural type correctness or simply enlarge his palette of design elements, which could then be used in the future.

3. Configuration Evolution

We employ flexible connectors to support the evolution of architectural configurations. Connectors remove from components the responsibility of knowing how they are interconnected. Connectors also introduce a layer of indirection between components. The potential penalties paid due to this indirection (e.g., performance) should be outweighed by other benefits of connectors, such as their role as facilitators of evolution. To facilitate architectural evolution, connectors must be flexible, i.e., they must easily accommodate changes to their attached components. At a minimum, these changes include component addition, removal, replacement, and reconnection.

Existing approaches tend to sacrifice the potential flexibility introduced by connectors in order to support more powerful architectural analyses. For example, Wright [1] and UniCon [16] require the architect to specify the types of component *ports* and *players*, respectively, that can be attached to a given connector *role*. Furthermore, although some variability is allowed in specifying the number of components that a given connector will be able to support (parameterized number of roles in Wright; potentially unbounded number of players with which each role may be associated in UniCon), once these variables are set at architecture specification time, neither approach allows their modification.

Our approach to configuration evolution is based on our experience with the C2 architectural style [17]. In the C2 style, connectors are communication message routing devices. To provide an added degree of freedom in composing components, C2 connectors support implicit invocation, which minimizes component interdependencies. We have demonstrated that C2 connectors provide strong support for evolution of architectural configurations both at specification time [11, 12] and at runtime [9, 14].

A unique aspect of C2 connectors, and a direct facilitator of architectural evolution, are their *context-reflective interfaces*. A connector does not export a specific interface. Instead, it acts as a communication conduit which, in principle, supports communication among any set of components. The number of connector ports is not predetermined, but changes as components are attached or detached. The "interface" exported by a C2 connector is thus a function of the attached components' interfaces. This allows any C2 connector to support arbitrary addition, removal, replacement, and reconnection of components or other connectors.

Clearly, it is not always the case that two components can communicate (e.g., due to mismatched interfaces, or message filtering), even though they may be attached to the same connector. At the architectural level, this can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into implementation, implicit invocation guarantees that, in the worst case, communication messages will be lost (*partial-* or *no-communication* [12, 17]), but the rest of the system's architecture will be able to perform at least in a degraded mode.

4. Conclusions and Future Work

Software architectures show great potential for reducing development costs while improving the quality of the resulting software. Architectures also provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools which also must be supported with specific techniques to achieve desired properties. In this paper, we have outlined two such techniques for supporting evolution, one for components and the other for architectural configurations.

We have already put a subset of these ideas into practice in the context of the C2 style and its accompanying ADL. We are currently developing a set of tools to support architectural subtyping, type checking, and mapping of architectural descriptions to the C2 implementation infrastructure [11]. We are also expanding C2 connectors to support more complex message passing protocols, as well as existing middleware technologies (e.g., CORBA and Java's RMI). A number of issues remain items of future work. These include investigation of techniques for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, further research of issues in adapting and adopting legacy components into architectures using the subtyping approach, automating the evolution of existing components to populate partial architectures, and assessment of the applicability of the properties of C2 connectors described in Section 3 to other architectural approaches.

5. Acknowledgements

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

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

6. References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, pages 60-90, Springer-Verlag, 1991.

[3] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.

[4] 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.

[5] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

[6] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

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

[8] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[9] N. Medvidovic. ADLs and Dynamic Architecture Changes. In Alexander L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24-27, San Francisco, CA, October 14-15, 1996.

[10] 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 the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 16-18, 1996.

[11] 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.

[12] N. Medvidovic and R.N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, pages 237-248, October-December 1997.

[13] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 60-76, Zurich, Switzerland, September 22-25, 1997.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. To appear in *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 19-25, 1998, Kyoto, Japan. Also available as Technical Report, UCI-ICS-97-39.

[15] J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992.

[16] 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.

[17] 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.

[18] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, pages 176-190, Portland, OR, USA, October 1994.