

Ensuring General-purpose and Domain-specific Properties

Using Architectural Styles

David S. Wile

Teknowledge Corp.

4640 Admiralty Way, Suite 231

Marina del Rey, CA 90292 USA

+1 310 578 5350 ext. 202

dwile@teknowledge.com

ABSTRACT

The use of *domain-specific* languages and development tools, such as those found in product-line development kits, have been shown to reduce programming and debugging costs considerably. At the same time, *general-purpose* Commercial, Off-The-Shelf (COTS) tools are improving rapidly. Moreover, providing security, privacy, reliability, and other Quality of Service (QoS) properties normally has little to do with the specific problem domains in which they are applied. Here, based on the observation that an architecture itself is a composition mechanism, I advocate the use of software architecture styles to provide a composition mechanism that retains the benefits of both domain-specific and general-purpose approaches.

Keywords

Domain-specific languages, architecture style, software architecture, composition

1 PROBLEM SPACE: PROPERTY SPECTRUM

Modern system design and construction policies must constantly deal with tradeoffs between domain-specific and general-purpose solution techniques. Domain-specific approaches require an initial investment in infrastructure development costs, after which considerable leverage has been demonstrated in subsequent development effort reduction and control, and in product reliability and maintainability [9]. Hence the success of customized product line development facilities and on a smaller scale, Fourth Generation Languages (4GLs). Contrasted with this is the adoption and adaptation of COTS tools to provide graphical user interfaces, presentation development tools, web-based interfaces, etc. Moreover, specific design

disciplines for development of policies and mechanisms to provide security, trust, robustness, and other QoS properties are emerging that are quite independent from the functionality provided by the specific product.

The problem is, how can the design of products in the future be controlled in such a manner that the best aspects of both of these development approaches – loosely, domain-specific and general-purpose – can be employed in the development of products?

To understand the issues more fully, it helps to identify what I call here the “Property Spectrum.” The level of properties that may be of interest to establish for a system varies from what are really very general-purpose properties to extremely domain-specific properties. Very general properties include properties like trust and certification; these are really 2nd order properties – applying to other properties, in fact. For example, one certifies that a particular property, like “valid identity of the user,” holds. Quite generic properties like security and robustness belong to a second “hue” of the property spectrum. The third is made up of properties like “adheres to protocols,” “uses appropriate datatypes,” or “meets performance promises.” And, naturally, there are myriad domain-specific properties that have to do either with implementations or the individual problem domains, such “an empty stack is never popped” or “cannot become critical” (e.g. in a nuclear reactor), respectively.

So the question becomes, how can we reason about all of these in concert? Naturally, the solutions sought always relate to “divide and conquer” or composition principles, but finding composition principles for some of these properties can be very difficult – properties in different hues of the spectrum often depend on one another. Moreover, the composition of two systems that share a property, like security, can easily violate the property.

I believe the solution to finding software properties amenable to compositional analysis, measurement, and prediction should be sought in the use of architectural

styles for describing, analyzing, refining and composing systems.

2 THE APPROACH: ARCHITECTURE STYLES

Software architectures can be described formally using so-called architecture description languages (ADLs). There are many such languages, generally specialized to particular kinds of properties that they are used to reason about [8]. For example, Meta-H allows reasoning about real-time response [10]; Wright facilitates reasoning about deadlock [2]; Rapide emphasizes reasoning about event causality [7]. The Acme language [4] was designed to exchange architecture descriptions among ADLs, but it can function as a low entry-fee ADL in itself.

To establish terminology for the following discussion, it is worth sketching the major concepts of Acme, without detailing the syntax itself. Architectures, or systems, are collections of *components* and *connectors*, attached to each other at attachment points called *ports* and *roles*, respectively, i.e. a component's port will be attached to a connector's role. Each of these *elements* – connectors, components, roles, and ports – as well as whole systems, may be characterized with *properties*, simple attributes known at specification time; a stronger form of property, called a *constraint*, is needed to reason symbolically about architectures.

Acme has several abstraction mechanisms that facilitate more concise specification of and reasoning about architectures. First, a *type* specification may be given for any of the four kinds of element. So, for example, one can describe a “filter” type of component or a “pipe” type of connector. Types are technically just a set of predicates that must be determined to hold on elements they purport to describe, but one can readily construct elements that have such properties, using the *new* construct.

A second form of abstraction is involved in *refining* elements. This states that one element is implemented, or logically equivalent to, a whole substructure of elements. One must *bind* the relevant interfaces from the implemented element to the implementing elements. For example, one would bind the roles of a connector that was being implemented by a set of components and connectors to roles within the subarchitecture.

Architecture *families*, or styles, are the final form of abstraction Acme provides. Families are simply collections of elements and types. A system can then be specified to be a member of the family. As with types, this is simply a predicate that the system satisfies the predicates implied by the family.

A crucial observation here is that *domain-specific languages are analogous to architectural styles*. In fact, we are migrating our old syntax-based tools that provide

support for the design and development of domain-specific languages to tools that support domain-specific architectural style development and analysis[6].

Fortunately, the Acme view of architecture types and styles is entirely *compositional*. In particular, a system can be in more than one family; the conjunction of the predicates from the two families must hold.¹ Hence, to deal with the spectrum of domain-specific to generic properties, the composition of domain-specific or generic styles seems to be very promising.

Hence, it is my position here that one should seek composition principles for software architecture styles rather than reason about properties on an element-by-element basis. The hope is to have a nearly direct analogy with hardware architectures. The oft-touted plug-and-play nature of hardware seems mostly to rely on what might be called “hardware styles,” characterized by such properties as voltage levels, grounding conventions, physical connector designs and wiring conventions. I can imagine analogies for impedance matching devices and other transformations as well, but I do not want to dwell on the analogy. I merely allude to it to give a feel for what I am (admittedly) grasping at, here

3 OPEN PROBLEM: A COMPOSITION STYLE

The desire to use the combination of architecture styles as a composition mechanism is not novel; postulating that it act as a “cynosure” for community consensus, is.

In [1], compatibility of architectural styles was studied for a small set of styles; [3] expanded the set in a slightly less formal setting. Both of these were based on finding a core set of cross-cutting conceptual features that characterized more than one architectural style. Concepts such as dynamism, support for data transfer, triggering, concurrency, and distribution were analyzed. Styles were described in Z in terms of a base set of features; one could define a base style in Acme that captured the same ideas.

Then, architectural composition was described as grouping systems using any of a set of style independent connectors. Some examples of these connectors are calls, spawns, transfers data, triggers call, and shares data. The two cited efforts produced tools for detecting inconsistencies of the conceptual features when such compositions were employed. So, for example, use of a triggered call to connect two systems may have a data transfer problem by referring to a subsystem that forbids data connectors. The work is quite usable for detecting some potential mismatches as is.

¹ Naturally, families can be inconsistent with one another, so the compositionality of reasoning does not guarantee compositionality of systems themselves.

However, one does not get the feeling that this work, even as extended by Gacek, is “complete,” in the sense that the spectrum of properties identified above could be covered in any sense. I think an expanded approach should be taken based on the observation that: *an architecture itself is a composition mechanism*. Hence, I can imagine designing an architecture style in which we can reason about composing systems in heterogeneous architecture styles. Abd-Allah-Gacek’s composition mechanism is a starting point for such a style, but looking at the problem this way may open up other avenues for composition. Properties of the styles themselves may be reasoned about, or second order properties of the systems being composed.

For example, consider the property of system “closedness” by which I mean that all elements of an architecture have been described [11]. Hence, for example, no new components or connectors can be formed. Then any property that depends on closedness, e.g. performance, cannot be guaranteed to hold when composed with another system using a sequential control connector, whereas it may be when using a parallel connector. Stated more positively, one might be able to characterize the performance of a system formed by the *closed* composition (via sequential connectors) of subsystems, using a particular kind of composition style – a composition style substyle!

As an aside, it is worth noting that there is a strong relationship between domain-specific languages and closed architectural styles (styles whose usage is restricted to insist that systems comprise elements described by types in the style). In both cases, certain properties may be guaranteed to hold without determining exactly how much one is relying on the closedness. In other words, one does not have to characterize the assumptions that use of elements solely in the style guarantees. Efforts to understand how to specify an open version of a style correspond to the descriptions of domain-specific languages in terms of general-purpose ones. (For example, one might find the assumptions “side-effect free” or “not affecting variable X” useful in such attempts.)

One needs to be able to reason about various levels of composition. For example, the performance of the closed composition just mentioned will vary depending on how the connector is implemented. So mappings between architectures for implementations will have to affect the reasoning process. But this is just like the reasoning that goes on in architectural styles today. A property of the systems being combined is their implementation style. Hence, properties such as performance can be contingent on such other properties. A more refined set of properties would include those cross-cutting concepts identified in [1, 3].

Finally, previous work has not been concerned with repair of architectural mismatches. In fact, repairs such as

introducing protocol-transforming connectors or wrapping exception-throwing systems with graceful reporting mechanisms, could also be introduced into a style for architectural composition.

4 WORKSHOP RELEVANCE

Synergy within the community

The approach taken by this community should probably be to develop a composition style that emphasizes one or two particular (well-understood) properties, at first. For example, performance and security might be reasonable candidates. These can be determined by the more industrial attendees of the workshop. By understanding what style elements are useful for describing compositions that make such properties analyzable, we can look toward generalizing to arbitrary properties. Obviously, the attendees of the workshop interested in formal methods can be invaluable here.

Positions

The following questions were asked in the CFP.

1. What do developers want to predict about component assemblies? Any of a wide variety of properties I labeled the “Property Spectrum” above.
2. What compositional reasoning techniques are available to support prediction? This is a proposal to develop a style that answers just this question.
3. Which of these techniques benefit from knowledge of component internals and what do they need to know? Here the components have been moved up to the system level, and the properties of these systems that will be necessary to understand will be of the nature of the cross-cutting properties identified by Abd-Allah and Gacek.
4. What can be known about component properties in the absence of knowledge of the context in which it will be deployed and used? Almost nothing. The architectural style for composition will be used to characterize the necessary context on a case-by-case basis.
5. How do we measure those properties and what degree of precision is required? These more abstract properties tend to be Boolean and there is usually a worst case scenario, but you always get out at most the amount of precision you put in.
6. How is this information made available by the component? Hopefully, we can devise formulas that derive the system’s property values in terms of the properties of the components from which it is composed along with those of its implementation architecture.

ACKNOWLEDGEMENTS

This work was sponsored by DARPA contract numbers F30602-96-2-0224 and F30602-00-C-0200.

REFERENCES

1. A. Abd-Allah. *Composing Heterogeneous Software Architectures*. Ph.D. Dissertation. Computer Science, University of Southern California. 1996.
2. R. Allen. *A Formal Approach to Software Architecture*. Ph.D. Thesis. Carnegie Mellon University CMU Tech. Report CMU-CS-97-144, May, 1997.
3. C. Gacek. *Detecting Architectural Mismatches during Systems Composition*. Ph.D. Dissertation. Computer Science, University of Southern California. 1998.
4. D. Garlan, R. Monroe, and D. Wile. Architectural descriptions of component-based systems. In *Foundations of Component-Based Systems*, Gary Leavens and Murali Sitaraman, ed.s Kluwer, 2000. (See also: [http://www.cs.cmu.edu/~acme/.](http://www.cs.cmu.edu/~acme/))
- 5 D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *IEEE Software*. Nov. 1995.
6. N. Goldman and R. Balzer. The ISI Visual Design Editor Generator. *Conference on Visual Languages*. 1999.
7. D. Luckham, et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21(4) April, 1995. 336-355.
8. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions of Software Engineering*. 26(1) January, 2000
9. C. Ramming and D. Wile, eds. *IEEE TSE Special Issue on Domain-Specific Languages*, May/June 99
10. Vestal. *MetaH Programmer's Manual, Version .14*. Technical Report Honeywell Technology Center, 7 Mar 1997.
11. David Wile. Modeling architecture description languages using AML. In *Automated Software Engineering*. Kluwer. (8) 2001. pp. 63-88.