Style-Based Refinement for Software Architecture

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
garlan@cs.cmu.edu

Abstract

A question that frequently arises for architectural design is "When can I implement a design in style S_1 using a design in style S_2?" In this paper I propose a technique for structuring a solution to this kind of problem using the idea of substyles. This technique leads to a two-step process in which first, useful subsets of a family of architectures are identified, and second, refinement rules specific to these subsets are established. I will argue that this technique, in combination with an unconventional interpretation of refinement, clarifies how engineers actually carry out architectural refinement and provides a formal framework for establishing the correctness of those methods.

1 The importance of style-based refinement

A key issue for architectural design is to understand when a system characterized in one style can be substituted for one described in another. That is, when can I use an instance of style S_1 in place of a system already described in style S_2? This problem arises in a number of contexts:

- **Representing the internal structure of an architectural component (or connector):** Often the elements of an architectural description are implemented internally by another, more detailed architecture. In some cases the boundary of one element serves to aggregate and encapsulate certain parts of a complex description (i.e., $S_1 = S_2$). In this case, simple name mappings between the internal representation and the external may suffice. But more generally, the internal representation may involve a change in architectural style. For example, I might have a component of a pipe-filter architecture that is internally constructed using componentry based on procedure-call.

- **Recasting an architectural description in a new style:** Often a system that is most naturally represented in one style must be implemented using the facilities that are best described in another style. In this case, a complete architectural description (not just individual elements) must be represented in a new way. For example, I might represent an event-based design in an object-oriented one, in order to pave the way for more direct implementation using an object-oriented language.

- **Applying tools developed for one style to a system written in another one:** Architectural analysis tools are often written for systems that fit within a given style. For example, a timing analyzer may apply only to pipe-filter systems, while a transaction throughput analyzer might apply only to a certain class of repository-oriented systems. However, sometimes we have a system that is characterized by a style that has enough commonality with another that it makes sense to apply tools developed for the second style. As a simple example, I should be able to apply a pipe-filter timing analyzer to an instance of a pipeline style (i.e., a linear sequence of filters).

For any specific instance of the problem it is possible to cast the solution in terms of traditional refinement. That is, system X_2 can be used in place of X_1 if we can find a suitable "abstraction map" from X_2 to X_1. This is essentially the approach taken by Rapide [3]. The events of one system are mapped to the events of the other system, and then tools check that the resulting behavior of the concrete system is consistent with that of the abstract system. (I will reexamine this simplistic notion of refinement later.)

However, architecture provides an opportunity to do much more. Instead of comparing *architectural instances* we develop a set of rules between *architectural styles.* We then prove that two systems (in the respective styles) that are related by the refinement rules represent a valid refinement relationship. For example, we might determine that any event-based system can be implemented as an object-oriented system if we transform each component so that it calls an event dispatcher, and we transform the event connectors to a dispatcher-mediated, broadcast connector. This approach to refinement can be called *style-based refinement.*

Naturally, techniques that apply at the level of styles are much more powerful than techniques that apply to instances. This is because the demonstration can be performed once for the styles and then reused many times for instances of those styles. Moreover, it permits verification to be done by specialists in style definition, while allowing regular system designers to simply use the results with confidence that they will be correct. Finally, it makes explicit the rules of thumb

SIGSOFT 96 Workshop, San Francisco CA USA
© 1996 ACM 0-89791-867-3/96/10 ..$3.50

that engineers develop to implement one kind of system abstraction in terms of another.

2 The problem of style-based refinement

The problem, however, is that style-based refinement is much more difficult to establish. To have a set of valid rules one must show that *every* instance of one style has a valid refinement in the other. Since the demonstration of substitutability must range over a possibly infinite collection of systems, it involves more general proof techniques.

One such technique, proposed by Moriconi and his associates, is to use a set of compositional refinement patterns [4]. The patterns determine local transformations for parts of an architectural description. The patterns are constructed in such a way that the local transformations compose and, if applied systematically to a system, will result in the construction of a system in another style.

This approach, when it works, is a good idea. However, in general, it may be difficult to find transformation rules that apply to all system instances in a given style. Consider, for example the problem of implementing pipe-filter systems using a shared variable style. (That is, pipe reads/writes are accomplished by reading and writing to shared variables). In the presence of cycles, a filter that needs to read more than one value before producing its result can cause the shared variable system to deadlock. Thus not all pipe-filter systems can be implemented by the simple rule of replacing pipes with procedure calls to shared variables. On the other hand, clearly *some* pipe-filter systems can.

In the remainder of this paper I sketch an alternative (but complementary) technique that advocates the principled use of substyles for dealing these issues of refinement. The basic idea is to identify subsets of the systems in a given style that are amenable to specialized treatment. This approach improves our capabilities for defining style-based refinement, because we are freed from the requirement of carrying it out on all systems within a given style. I argue that this not only provides a solid formal basis for carrying out style-based refinement, but corresponds to what engineers actually do (informally) when then they implement architectural designs. I then elaborate on both these points by enumerating the formal and methodological implications.

3 The use of substyles in refinement

In practice, systems builders often exploit idiosyncracies of a given system to provide efficient implementations. For example, a program that uses sets, but requires only sets with small numbers of binary values, may use a bit vector.

For architectures the situation is similar. If I have a system that I *know* is constructed as a linear sequence of filters, I can produce a functional implementation that simply composes the transformations of the pipeline into a single, more efficient function. Thus, as long as one is concerned with refinement of architectural instances, it is possible to demonstrate that a specialized refinement technique is appropriate.

But when we move to styles, this is no longer possible. Since we cannot exploit the particular idiosyncracies of a design we can't in general take advantage of these special cases and contexts of use. Thus there is an apparent dilemna: Style-based refinement is more powerful and generally useful, but can be applied much less effectively.

I would argue, however, that a solution can be found by applying at the style level the same technique used for instance refinement: we exploit the idiosyncracies of a subset of the systems in a style to produce specialized refinement rules for that subset. This involves a two step process. First, suitable substyles of the style are identified. A substyle is characterized as a collection of additional constraints (beyond those imposed by the style). Second for each substyle, we use the additional constraints to define specialized refinements.

To take a simple example, we might define a "pipeline" as a substyle of pipe-filter, by adding the constraint that the topology must be linear. For this substyle, we can then define a simple refinement rule from that substyle into a functional composition style (i.e., stream transformations are simply composed). A more complex substyle of pipe-filter would be, acyclic, "balanced", 1-in-1-out systems. In these systems each filter computes one output value for each input value that it consumes, and the overall graph is acyclic. For this substyle, we can define a simple refinement rule into a shared variable style. (For an example of a formal proof of a similar result, see [1].)

Formally, this approach can be characterized by two functions, as illustrated in Figure 1. The first is a partial projection function that determines whether a given system (labelled Style 1), is a member of a specialized substyle (labelled Style 1a).[1] The second function determines the refinement relationship between members of the abstract (sub)style and members of concrete style (labelled Style 2). Formally this is characterized as an abstraction function, since there may be many concrete systems that can be used to represent the abstract system. Note, however, that both functions are surjective: that is, all elements of the substyle are handled.

In terms of engineering practice, we observe that this general approach is what engineers typically do. When presented with a system that must be implemented in a different style, the engineer will typically look for characteristics of the proposed system that enable the use of certain standard techniques of implementation. The definition of substyles simply makes explicit (formally) which constraints must exist exist in order to use those techniques safely. Thus the approach of defining useful substyles provides a home for capturing the understanding of experienced designers.

4 What is refinement, anyway?

Until now, we have been vague about what refinement actually means. However, the definition of refinement is a critical issue, since it determines the correctness criteria for a set of refinement rules (or equivalently, the abstraction map).

The "classical" approach is to use the notion of behavioral substitutability. That is, the concrete representation should not produce any externally-observable behavior that the abstract representation could not have produced. This is essentially the criterion used by Rapide. It is also the criterion used by CSP[2].

But, as Moriconi and others have argued [4], behavioral substitutability may not be strong enough. In general, there may be properties other than computational equivalence that we wish to preserve in the concrete representation.

Moriconi proposes that for architectures it is sufficient to impose a requirement of "conservative extension". Formally

[1] The function may include renamings, such as "filter" to "stage".

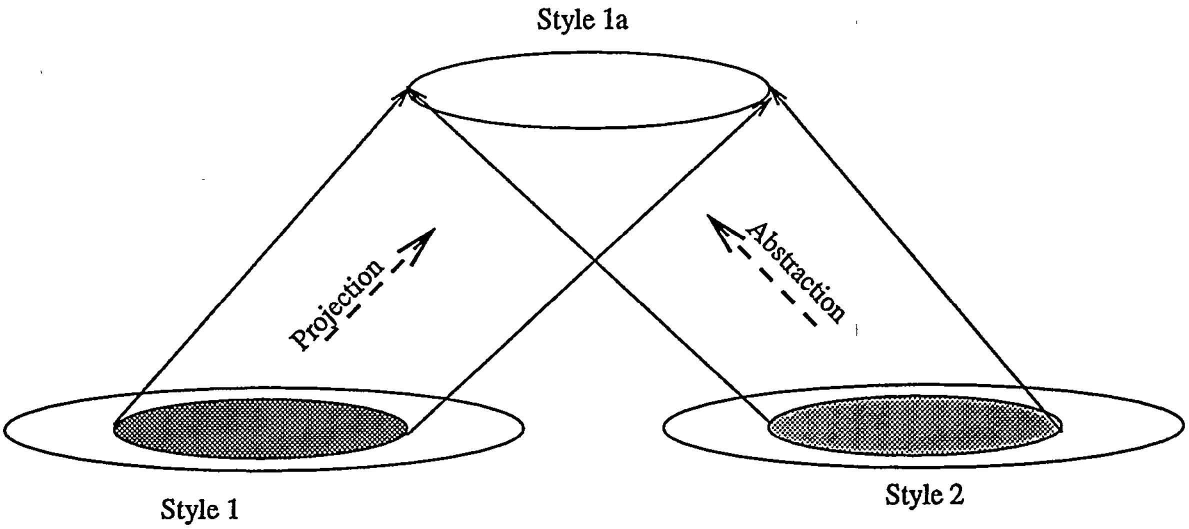

Figure 1: Substyles and Refinement

this means that results not provable about the abstraction are considered to be false. In other words, if we don't explicitly include a particular feature, then we are implicitly claiming that it does not exist. Thus refinements can only elaborate (or restrict) what was already present in the abstraction. The basic idea here is to avoid a situation in which an architectural refinement violates the structural integrity of the system it is refining—even though it might still produce the same computations.

As a simple example, if we do not include a connector between two components in an abstract architecture, then we can conclude that they do not interact. Hence, a refinement could not introduce a connector that caused them to interact (within the style of the abstract system).

We would argue, however, that this restriction is much too restrictive. Since architectural descriptions are by their nature abstract, we may choose to leave details out of the description, because we would like the implementor to bind them in a lower-level description. That is, I may use the absence of information to delay the binding of certain design decisions. In such cases, I certainly don't want the system to preempt that choice for me. Moreover, if I have a tool that can automatically generate an implementation from an architectural specification, why should I care if the representation style is completely different from the one used in the abstract architecture?

But if "conservative extension" is too strong, and "classical" refinement is too weak, what is needed? We argue that what is at issue here is to capture the notion of "relative" substitutability. By this we mean that a refinement must be substitutable for the system it is refining, *with respect to a set of properties of interest.* In the case that the property of interest is externally-visible computational behavior, then we get classical refinement. However, we can pick other kinds of properties. For example, if an abstract architecture is evolvable along certain dimensions (by localizing certain design decisions, for example), we may wish our refinement to preserve that property. As another example, if we are primarily concerned with performance, we might make refinement relative to the target performance of a system: the concrete system must perform at least as well as the abstract system.

In short, we claim that there is no single definition of refinement. Rather, refinement rules must be explicit about what kinds of properties they are preserving in the refined design.

5 Implications

We believe that the approach outlined above has both formal and methodological implications. Formally, it suggests that a critical issue for refinement is to understand the formal relationship between naturally-occurring constraints on the use of an architectural style, and the ability to exploit them when changing representations. The job then is to provide formal guarantees that the constraints of those styles are sufficient to guarantee correctness of the refinement rules. Furthermore, formalists must be explicit about the refinement criteria that are being employed: rather than assume a single view of substitutability, they must recognize that refinement is relative to the properties of interest for that class of system.

Methodologically, the approach suggests that when architects identify styles, they should also classify substyles that will permit natural representations in other common styles. Further, as systems designers build understandings about practical representation techniques for representing one architectural style with another, they should make explicit the constraints that they are exploiting to carry out that change in representation, as well as the properties that they believe are preserved across the change.

6 Acknowledgements

The position argued in this paper grew out of numerous discussions about refinement with Rob Allen, Mark Moriconi, Mary Shaw, and Zhenyu Wang—all of whom I would like to thank.

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9109469; and by a grant from Siemens Corporate Research. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, the National Science Foundation, or Siemens Corporation.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

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

[3] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[4] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.