Toward User-Defined Element Types and Architectural Styles

Robert DeLine
Computer Science Department
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213-3891
rdeline@cs.cmu.edu

Abstract

When considering the design of an architectural description language (ADL) to be used as part of a software developer's daily practice, two goals merit attention. First, the language should support the easy definition of new element types and architectural styles. Second, it should play a central role in system construction. A proposed ADL, called UniCon-2, addresses these goals with its flexible type system, its **duty** construct, and its extensible compiler architecture based on OLE. Such an ADL provides a good starting point for exploring the architectural description of families of systems and flexible componentry.

Introduction

A number of researchers in the field of software architecture are centering their work on new notations, called architectural description languages (ADLs). When considering the eventual adoption of ADLs into standard software development practice, two language features seem worthy goals. First, the ADL should support the definition of new types of components, new types of connectors, and new architectural styles. Second, the ADL should play a central role in system construction. Designing an ADL that meets both these goals is the focus of this paper.

The world is populated with a diversity of software architectures both across and within software systems. This diversity is not a historical accident or a practice to be denigrated: certain architectural styles are simply better suited than others to solve certain problems [3]. As long as there is a variety of problems to solve, there will be a variety of architectural styles. To support this diversity, an ADL should allow architects to define their own architectural styles, along with the new types of components and connectors that are a part of that style. Moreover, the definition of new types and styles should not require exotic skills. The more ordinary the skill set required to define a new type or style, the more the average architect can participate.

The second goal is for the ADL to play a central role in system construction. The marriage between system description and system construction is natural, since both at heart are about the pieces make up a system and how those pieces fit together. The compiler or environment for an ADL can provide a natural home for the construction methods of a given architectural style, as well as many of the niggling details that underlie architectural connections. Typical details include: the input format for the RPC stub generator and the knowledge of how to invoke it; methods for creating pipe and filter systems with arbitrary topologies, without causing deadlock; and the right list of kernel calls to initialize a system under a real-time operating system. An ADL that encapsulates such details and makes system construction easier would be especially attractive to practitioners.

An ADL that meets these goals makes a good starting point for exploring architectural descriptions of families of systems. Many developers work not on a single product, but on a group of products, each of which is a small variation on a common theme. The architectures of these variants may have many, but not all, parts in common and may connect those parts in slightly different ways. Current ADLs are inadequate to describe architectural styles that have this kind of prescribed envelope of variability. Further, a cousin of the family of systems problem is the problem of describing flexible componentry, for example, a component that features an Open Implementation [4]. Here too the description of such a component needs to capture its envelope of possible behaviors.

Several research projects have begun to work toward the two goals mentioned above. Aesop [2] allows new element types and architectural styles to be defined as subclasses of existing types and styles stored in an object-oriented database; refinements are made by overriding the methods that govern that type or style. Moriconi et al [5] allow user-defined types and styles whose emphasis is on provably correct architectural refinement; hence, their type and style definitions require mathematical skills absent in many current practitioners. Neither of these works place an emphasis on system construction. One ADL with this emphasis is UniCon [6]. However, the current version of UniCon presents a closed set of element types and architectural styles, although we have recently taken steps towards user-defined connectors with the notion of connector experts [7].

Quick Tour of UniCon

To design an ADL that meets both these goals, I propose a new version of UniCon, called UniCon-2, with three major changes: a rich type system; a new construct, called a **duty**; and an extensible compiler architecture. Since the first two of these new features focus largely on syntactic issues, a quick review of UniCon's terminology and syntax is needed to understand them.

An architecture in UniCon consists of a set of components whose interaction is mediated by a set of connectors. A component's interface exports a set of players that engenders that component's possible interactions with the outside world. A connector's protocol exports set of roles that engender the ways it mediates interactions. An architectural configuration is created by instantiating a set of components and connectors and then hooking them together by associating the components' players with the connectors' roles. Here are two example declarations, a filter component, and a pipe connector.

```
component Sort
  Filter interface
    Stream-In player input
    Stream-Out player output
    Stream-Out player error
      unix-port = Stderr
    end error
  end interface
  external
    impl = Executable("sort", [])
  end external
end Sort


connector Pipe
  Unix-Pipe protocol
    Pipe-Source role source
    Pipe-Sink role sink
  end protocol
  external
    impl = Expert("pipe")
  end external
end Pipe
```

The component Sort exports an interface with three players: one of type Stream-In, named input; and two of type Stream-Out, named output and error. The error player is annotated with a property whose name is unix-port and whose value is Stderr. UniCon, as a property-based notation, relies on such properties to provide most of the information about an element. Sort's implementation is external, i.e. given in some notation outside the scope of UniCon; this implementation is annotated with a property named impl. The connector Pipe's definition is analogous to Sort's.

Type System

UniCon-2's first change to support user-defined element types and architectural styles is to make the values of properties be strongly typed. The value of a property in UniCon is given in a syntax that's idiosyncratic to the particular property; this makes adding new properties tedious. UniCon-2's type system, borrowed largely from ML, instead provides a rich language for describing the values of properties. This type system includes a number of base types, such a bool, int, real, and string, as well as a new type called uninterpreted. The type uninterpreted by design supports no operations whatsoever and allows UniCon-2 to harmlessly carry information that outside tools are intended to consume. The type system supports polymorphic types such as lists and tuples (but not functions) and also allows user defined types, such as these:

```
type unix-port    = Stdin | Stdout | Stderr
type impl-rep     = Source of string
                  | Object of string
                  | Executable of string * string list
                  | ObjectLibrary of string
                  | Expert of string
```

A type definition provides a list of constructors, separated by vertical bars. Each constructor consists of a name and an optional type and is used to denote values of the new type. Based on the definitions above, the expression Stdin denotes a value of type unix-port, and the expression Source("foo.c") denotes a value of type impl-rep. For a property-based language, this ability to express precisely the type of a property's value is an important first step toward user-defined element types and architectural styles.

Duty Construct

Given the flexibility to write down properties with appropriate values, the question arises of what properties *ought* to be written down (or in general what sub-constructs are expected to be recorded within a given construct). For example, how did I know to include Stream-In and Stream-Out players in the Filter interface above or that a Stream-Out player can be meaningfully annotated with a unix-port property? In UniCon the answer comes from reading the Language Reference Manual, while UniCon-2 introduces a new construct, called a **duty**, to address this concern.

A **duty** describes what information should be supplied for a given type of player, interface, role, protocol, or configuration. It includes a **requires** clause with a list of patterns that must be matched and an optional **provides** clause with a list of constructs to be added. These patterns consist of phrases from the language, possibly including a dollar sign wildcard. Here's an example:

```
interface duty Filter
  requires
    ( Stream-In player $ | Stream-Out player $ ) +
  provides
    gui-icon-size = (120, 80)
    gui-icon = [ …omitted… ]
end Filter
```

According to this duty, any interface that declares itself to be a Filter, such as Sort's interface above, must include at least one Stream-In player (whose name doesn't matter) or one Stream-Out player (whose name doesn't matter). The duty also adds two default properties, gui-icon-size and gui-icon, to the interface.

One duty may specialize another duty and may also require the absence—not just the presence—of certain

constructs. Here's an example of this:

```
interface duty Strict-Filter
  includes Filter
  requires
    (Stream-In player $) +
    (Stream-Out player $) +
  closes
    $ player $
end Strict-Filter
```

An interface that declares itself to be a Strict-Filter interface must not only uphold the Filter duty but must also have at least one Stream-In player and at least one Stream-Out player. Further, no *other* types of players are allowed—specifically, no players that don't match the **requires** pattern but that do match the **closes** pattern. Finally, although these examples show interface duties in particular, the type **duty** is polymorphic, which allows for player duties, role duties, protocol duties, etc. A configuration duty captures UniCon-2's notion of an architectural style.

Here then are the rules to check whether some construct *c* upholds a duty. First, for each **includes** clause, the body of the named duty is textually included in situ. For each **provides** clause, the given set of constructs is textually added to *c*. For each **requires** clause, the given pattern is compared the constructs in *c*. Any pattern for which no matching construct is found generates an error. Finally, for each **closes** clause, if any of the previously unmatched constructs in *c* matches the given pattern, an error is generated.

Extensible Compiler Architecture

Together the new type system and **duty** construct allow the definer of a new element type or architectural style to state what information must (or must not) be included in an instance of that type or style. However, these new languages features are not expressive enough to capture the idiosyncratic semantic checks and construction information particular to some type or style. For example, the **duty** construct will allow an architect to insist that a Procedure-Call player have a property called signature with a value whose type is expressed in the type system. But how can the architect express the intention that, whenever there's a procedure-call connection between a Procedure-Call player and a Procedure-Def player, their signature properties should be compatible? Moreover how is she to express what constitutes this notion of signature compatibility?

By augmenting UniCon-2 with ever more powerful notations (logics, relational calculi, programming constructs, etc.) one might be able to create a language capable of capturing all of this type/style-specific information. However, some of this information will doubtless not be natural to express in this *über*-notation, and any notation this powerful is also likely to be inscrutable to many practitioners. A better strategy is to open up the UniCon-2 compiler architecture to allow the insertion of new parts to handle these type/style-specific checks and construction information.

UniCon has taken a step in this direction by encapsulating the information associated with a connector type into a connector expert. This expert is made up of a set of code fragments, literals, and table entries that are injected into the UniCon compiler's source code before compilation [7]. The UniCon-2 compiler design improves on this, not only by adding component and style experts, but by using OLE [1] to encapsulate the expertise. OLE offers a number of advantages for a type or style expert: the expert is isolated from compiler internals and other experts, and its interface is explicit; it can be produced using any one of the growing set of programming languages and platforms OLE supports; and it can be linked into the compiler at run-time. To return to our example, the previously mentioned architect would express the algorithm for ensuring signature compatibility by creating an expert for the procedure-call connector and implementing the algorithm in the appropriate expert function.

In summary, the type system and duty construct allow the definer of a new element type or architectural style to express precisely what information is to be included in an instance of that type or style. How to use that information to determine the semantic legality of the instance or to build the instance is expressed in a standard programming language and encapsulated in an OLE component.

Looking Ahead

Once the dust settles from this redesign, the next goal for UniCon-2 will be the ability to specify architectural elements that export some amount of flexibility in their interface or behavior. Note that some initial support for this appears in UniCon-2. In the example above, the Filter duty defines an envelope of possible players, which the Strict-Filter duty restricts. Surveying the kinds of flexibility that elements and architectures need to express and finding notational ways to express that flexibility are the next big research challenges.

References

[1] K. Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[2] D. Garlan, R. Allen, and J. Ockerbloom. "Exploiting style in architectural design environments," in *Proc. ACM SIGSOFT '94: Symp. Foundations of Software Eng.*, Dec. 1994.

[3] M. Jackson. *Software requirements and specifications: A lexicon of practice, principles, and prejudices*. Addison-Wesley, 1995.

[4] G. Kiczales. "Beyond the black box: Open Implementation," *IEEE Software* 13(1).

[5] M. Moriconi, X. Qian, R. A. Reimenschneider. "Correct architecture refinement," *IEEE Trans. on Software Eng.* 21(4), Apr. 1994.

[6] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik. "Abstractions for software architecture and tools to support them," *IEEE Trans. on Software Eng.* 21(4), Apr. 1994.

[7] M. Shaw, R. DeLine, G. Zelesnik. "Abstractions and implementations for architectural connections," *Proc. 3rd Int. Conf. on Configurable Distributed Systems*, May 1996.