A Case Study in Software Architecture Interchange[*]

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
(412) 268-5056
garlan@cs.cmu.edu

Zhenyu Wang
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
(412) 268-5056
zwang@cs.cmu.edu

Abstract

An important issue for the specification and design of software architectures is how to combine the analysis capabilities of multiple architectural definition languages (ADLs) and their supporting toolsets. In this paper, we describe our experience of integrating three ADLs: Wright, Rapide, and Aesop. We discovered that it is possible achieve interoperability in ADL tools for a non-trivial subset of the systems describable by these languages, even though the languages have different views about architectural structure and semantics. To carry out the integration we used the Acme architectural interchange language and its supporting tools.

Keywords: software architecture, architecture description languages, architectural interchange, Acme, architecture analysis tools

1 Introduction

An increasingly important problem for complex software systems is the ability to specify and analyze their architectures. At an architectural level of design, one typically identifies the key computational entities and their interactions. These high-level descriptions are then used to understand key system properties such as performance (latencies, throughputs, bottlenecks), reliability, modifiability, etc.

Unfortunately, current practice relies on little more than informal diagrams and notations to support this activity. This imprecision in defining architectures limits the ability to carry out useful analyses, and even to communicate with others effectively. In response, a number of formal architectural description languages (ADLs) have been developed [MT97]. Typically, each of these ADLs supports the description of architectural *structure* together with some form of associated *semantics*. The particular semantics usually determines what kinds of useful analyses can be carried out on systems described in that ADL.

To take a few examples, Rapide [LAK$^+$95] provides semantics based on posets, and supports analyses based on (among other things) animation and simulation. Wright [AG97] provides semantics based on CSP, and supports static analyses of deadlock freedom, and interaction consistency. Darwin [MDEK95] provides semantics based on the Pi Calculus, and supports description of dynamically reconfigurable distributed systems.

For any particular system, it may be that the desired analyses are completely covered by those supported by a single ADL. However, in general it is useful to exploit capabilities of multiple ADLs. Unfortunately, each of the current ADLs stands in isolation, making it difficult to do this.

One proposal to help ameliorate the situation is to use a common *architectural interchange language* for exchanging architectural descriptions between var-

[*]Research sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031, and by the National Science Foundation under Grant No. CCR-9357792. 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.

ious ADLs.[1] Specifically, a notation called Acme, has been proposed as a candidate interchange language [GMW97]. Acme provides a simple, generic vocabulary for describing architectural structures as hierarchical graphs of components and connectors. In addition, Acme has a flexible annotation mechanism that permits each ADL to encode non-structural information (such as types, signatures, protocols, performance and reliability estimates, etc.).

To the extent that different ADLs share an interpretation of the Acme encoding, they can communicate. However, even when they do not, it may be possible to massage the Acme representation to make a given description accessible by other ADLs.

In order for such a scheme to work, however, two assumptions must hold. First, it must be possible to translate sufficient semantic content of an architectural description from one ADL to another. Otherwise, it will not be possible to exploit the analysis capabilities of each. Second, the use of Acme must provide advantages over pairwise translation (i.e., direct ADL-to-ADL). Otherwise, there would be no reason to go through an *intermediate* form.

In this paper we present a case study that sheds light on these two assumptions. Specifically, we describe our experience using Acme to integrate three ADLs: Wright, Rapide and Aesop. With respect to semantic translation issues, we will focus primarily on Wright and Rapide, since these were the most problematic. As we will illustrate, it is possible to map a substantial subset of Wright descriptions into Rapide, even though the two languages have somewhat different views about architectural structure and the semantics of architectural behavior. We also briefly consider the cost effectiveness in using Acme to carry out the translation.

2 Wright, Rapide, and Aesop

Before describing our approach to integration, we first briefly describe the three ADLs that we attempted to integrate.

Wright

Wright models system structures using the abstractions of components, connectors, ports, roles and configurations [AG97, All97]. *Components* represent processing elements and *connectors* describe interactions between them. Each component and connector has an associated specification described using a variant of CSP. These specifications describe the abstract behavior of the element in terms of the events that it can engage in. Additionally, both components and connectors have interfaces. Component interfaces are called *ports*, while connector interfaces are called *roles*. These interfaces are described by protocol specifications (also in CSP). System descriptions, called *configurations* are defined by attaching roles of connectors to ports of components.

Wright provides a number of useful static analyses of architectural descriptions [All97], including:

- Port-component consistency: checks whether a port protocol is a valid projection of the component's internal behavior.
- Port-role compatibility: checks whether a port's behavior meets the requirements imposed by a connector to which it is attached.
- Connector deadlock-freedom: checks whether a connector represents an interaction that cannot deadlock.
- Attachment completeness: checks that any unattached port or role makes no assumptions about the behavior of its environment.

These checks (and others) are carried out by the Wright toolset using FDR [FDR92], a commercial model-checker for CSP.

Rapide

Rapide also describes an architecture as a composition of components [LAK+95]. Each component (called a *module*) has a set of *interfaces* that describe patterns of events that can take place. Component behavior is specified in terms of the way outgoing events are produced in response to incoming events.

Rapide provides a fixed form of connection: essentially, a connector indicates how output events produced at one interface appear as input events at other interfaces. Rapide also provides a bundling facility for connectors, called services.

Rapide's primary form of analysis is based on tool-supported examination of system runtime behavior. Thus Rapide can function as a kind of architecture simulation language: sets of traces (technically, *posets*) can be examined for satisfaction of desirable ordering relations. Additionally, Rapide provides run-time animation capabilities with its "Raptor" tools.

[1]Here and elsewhere, when we refer to an ADL we will mean that ADL *and* its associated toolset.

2.1 Aesop

Aesop provides a toolkit for describing and enforcing architectural *styles* [GAO94]. An architectural style is a set of component and connector types, together with rules for how they can be legally combined. For example, a Pipe-Filter style might describe a filter component type and a pipe connector type, together with rules that indicate how pipes must connect output ports of one filter to input ports of another. Or, a client-server style might describe client and server component types and a client-server connector type, with rules that govern how many clients can communicate with a given server, and whether servers can communicate directly with other servers.

Aesop includes a graphical editor that can be specialized with visualizations appropriate to different styles. It also serves as a harness for analysis tools that can be invoked on architectural descriptions.

Viewed as an ADL, Aesop represents architectures as a system of objects. It uses an object-oriented language for describing both the types of components and connectors in a style as well as the semantic "behavior" of architectural instances.

3 Integration Scheme

The three ADLs described above have complementary capabilities. With its graphical editing capabilities, and support for domain-specific architectural design (using styles), Aesop is a good front end for architectural design. To carry out deeper semantic analyses on these designs, Wright provides capabilities for statically checking the consistency and completeness of the design. With its support for simulation, runtime analysis and animation, Rapide provides other important capabilities for evaluating an architectural description.

Clearly it would be beneficial to harness all three in a single environment. However, a number of difficulties present themselves. First, each ADL has a somewhat different view of the structure of architectures. Second, and more importantly, there are some significant differences in the way the languages specify architectural properties, such as abstract behavior. How can one bridge these mismatches?

In an attempt to answer this question we created a prototype environment in which all three ADLs were integrated using Acme as an interchange language, together with several Acme-based architectural transformation tools that we developed to handle the mismatch problems.

Acme

As noted earlier, Acme provides a simple structural framework for representing architectures, together with a liberal annotation mechanism [GMW97]. Acme does not impose any semantic interpretation of an architectural description, but simply provides a syntactic structure on which to hang semantic descriptions, which can then be interpreted by tools. The Acme language is a simple textual notation, designed for ease of tool manipulation.

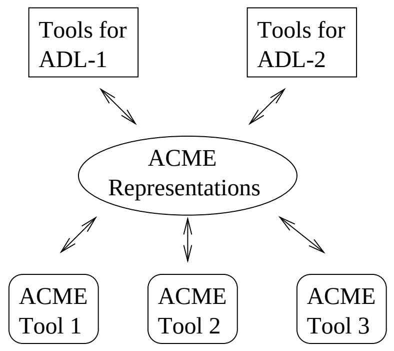

Figure 1: Acme-based Integration

An architectural design is shared among several ADLs by first translating the design into an Acme representation (see Figure 1). This representation can then be read by other ADLs that understand Acme, or it can be manipulated by tools that operate on Acme directly. Acme comes with a rich library for parsing, unparsing, and manipulating the representations, together with a growing corpus of tools for performing graphical layout, Web generation, and analysis.

Using Acme

To integrate the three ADLs we used the integration scheme illustrated in Figure 2. In this ensemble Aesop acts a graphical editor, exploiting its visualization and style-enforcement capabilities. Aesop descriptions are initially annotated with Wright specification fragments.[2] To carry out *static analyses* on these designs we translate Aesop into Acme, and then to native Wright, on which Wright's analysis tools can be invoked. To carry out *dynamic analyses* we use an intermediary tool that transforms Acme with Wright annotations into Acme with Rapide annotations. This transformed description can then be

[2]Alternatively, as we illustrate later, the user can start with a Wright specifications, which can be imported into Aesop (via Acme) for graphical viewing.

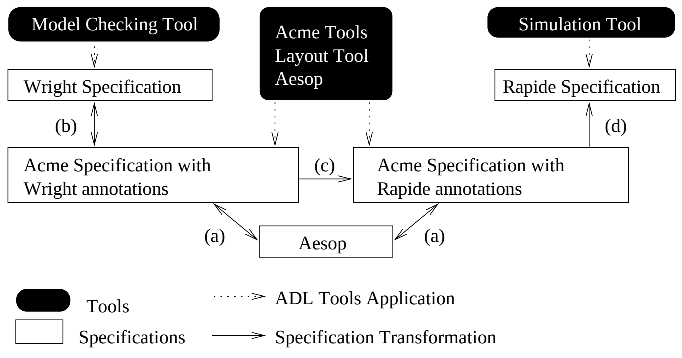

Figure 2: Integration of Aesop, Wright, and Rapide

mapped into native Rapide on which Rapide's behavior analyzer and animator can be invoked.

In this scheme Acme plays the role of an intermediary representation, that can be operated on by the various translation tools. While Acme provides neutral ground through which ADL translation can take place, clearly it does not make the differences between different ADLs disappear. In particular, although Wright, Aesop, and Acme are quite similar in their treatment of structure, there are significant differences between Wright and Rapide both structurally and semantically.

With respect to architectural structure, the most significant difference is that Wright permits the definition of new kinds of connectors (and their semantics), while Rapide provides a fixed set. Another key difference is that Rapide permits the creation of new architectural elements at runtime, while Wright describes only static architectures.

With respect to semantics, both Wright and Rapide describe the behavior of architectures in terms of patterns of events. However, there are three significant differences. First, Wright's behavior definitions are functional (with heavy reliance on recursion), whereas Rapide's are largely imperative. Second, Rapide imposes some restrictions on the use of nested parallelism in its descriptions, whereas Wright does not. Third, non-determinism can be made explicit in Wright specifications (using the CSP's internal choice operator), while it is implicit in Rapide.

In the next section we briefly describe how we handle these and other problems.

4 Integration Details

As illustrated in Figure 2, there are four key translations that contribute to the overall integration (labeled (a) – (d)).

(a) Aesop ↔ Acme

Translation between Acme and Aesop is relatively straightforward because Aesop and Acme have a similar notion of structure. In particular, in Aesop connectors are first class entities, which map well to Acme connectors. In the reverse direction Acme components and connectors can be mapped to components and connectors in the "generic" Aesop style. Further, Aesop permits arbitrary properties to be associated with architectural objects, providing a natural home for Acme property lists. (Although we don't have space to discuss it here, styles in Aesop are easily represented using the "families" of Acme.)

(b) Wright ↔ Acme$_{Wright}$

The translation from Wright to Acme consists of mapping Wright's components and connectors to Acme's, and then adding properties to those elements that correspond to the Wright behavior descriptions. That is, each component and connector is annotated with a property that specifies its Wright behavior. Since Wright and Acme have a similar view of structure this step is likewise straightforward.

(c) Acme$_{Wright}$ → Acme$_{Wright+Rapide}$

In this step the properties that characterize Wright behavior are transformed to create additional properties that characterize Rapide behavior. To do this we must find a way to create Rapide module code fragments from the Wright "process" annotations stored in the Acme representation.

Wright processes are defined using a subset of CSP that includes operators for event sequencing ($\rightarrow$), external choice ($\Box$), internal choice ($\sqcap$), and parallel composition ($\|$).[3] In addition, unlike CSP, Wright distinguishes between initiated and observed events: the former is indicated with an overbar.

Figure 3 shows how each of these constructs is mapped into a Rapide specification. Most of the translations are straightforward: Wright initiated events correspond to Rapide output events, while observed events correspond to Rapide input events. Top-level parallelism can be translated directly to

[3]Space does not permit a detailed description of these operators: however, a detailed understanding is not necessary to understand the main ideas of the paper.

```
Map ($\overline{e} \rightarrow P$) = e(); Map(P);

Map($e \rightarrow P$) = await e(); Map(P);

Map ($P_1 || \cdots || P_n$) =
    parallel
        Map($P_1$)
        ||
        ...
        ||
        Map($P_n$)
    end parallel;

Map ($e_1 \rightarrow P_1 \Box \cdots \Box e_n \rightarrow P_n$) =
    await
        $e_1 \Rightarrow P_1; \ldots ; e_n \Rightarrow P_n$;
    end await;

Map ($P_1 \sqcap \cdots \sqcap P_n$) =
    $\tau$();
    await
        $\tau() \Rightarrow P_1; \ldots ; \tau() \Rightarrow P_n$;
    end await;
    where $\tau$ is defined as an internal event of the module
```

Figure 3: Acme$_{Wright}$ to Acme$_{Wright+Rapide}$

the "parallel" construct of Rapide. Guarded internal choice simply becomes the Rapide event-trigger case statement.

The only three non-obvious parts are the handling of non-determinism, nested parallelism, and recursion.

To handle non-determinism that arises from an internal choice in CSP ($\sqcap$) we introduce a local Rapide event τ (invisible to other modules) that serves as a guard for each of the processes in the choice. Since multiple choices with the same event guard in Rapide is treated as non-deterministic choice, we achieve the desired effect.

To handle nested parallelism we could have used CSP's algebraic operators to eliminate parallelism before performing the translation. However, as it turns out, nested parallelism is rarely used in Wright, and so instead we constrain our translation to apply only to Wright descriptions that don't use this feature. (Use of nested parallelism can be easily checked for, and a warning can be issued to the user, who can perform the transformation by hand if desired.)

To handle recursion in CSP processes, we first map CSP processes into a state machine and then generate Rapide code to simulate this state machine. This is accomplished by assigning a state number to each process and defining a Rapide variable that records the current state of execution. When one process (recursively) transitions another process, the translation arranges for the state variable to change correspondingly.

Acme$_{Wright+Rapide}$ $\rightarrow$ Rapide

In this step, the goal is to translate Acme structures (i.e., components and connectors) annotated with Rapide fragments into native Rapide structures such as interfaces and modules. The basic idea behind the translation is that component interfaces in Acme become interfaces in Rapide. Component behavior (described by Wright property specifications) become module implementations in Rapide. The key difficulty is how to handle the fact that in Acme (as well as Wright and Aesop) new connectors may be defined and those connectors may have complex specifications, whereas in Rapide connectors are primitives.

There appeared to be two ways to handle the situation. One was to limit the class of architectural descriptions to just those that use the connector types provided by Rapide. The second was to represent complex connectors as modules in Rapide. We chose the latter, because it allows a much larger class of systems to be handled by the translation. (We will return to this issue in section 6.)

Thus to complete the translation, Acme connectors are translated into Rapide modules, and Acme attachments (between connectors and components) are translated into event bindings between the Rapide interfaces of modules representing Acme components and the Rapide interfaces of modules representing Acme connectors. Finally, we map an Acme top-level "System" into a Rapide top-level "Architecture".

5 Example

To illustrate how the pieces fit together for the integration of Wright and Rapide, consider the following deliberately simple (partial) Wright specification of a single reader attached to a pipe.

System Reader-Pipe:

Component Reader
  Port read = $read?x \rightarrow$ read
  Computation = $read.read?x \rightarrow$ Computation

Connector Pipe
  Role read = $\overline{read?x} \rightarrow$ read
  Role write = $\overline{write!x} \rightarrow$ write
  Glue = $write.write?x \rightarrow \overline{read.read!x} \rightarrow$ Glue

Instances
  reader1 : Reader;
  pipe1 : Pipe;

```
    Attach pipe1.read to reader1.read
```

After the first translation step (b) we obtain an Acme description annotated with Wright processes. (The Acme description appears in Figure 4 at the end of this paper.) After the second translation step (c), the Acme description is almost the same, except that each component or connector now has a new Rapide property generated from the Wright properties. From this, we synthesize a native Rapide specification (d) to get:

```
type reader is interface
    action
        in          read_read(x : Params);
end interface reader

type pipe is interface
    action
        in          write_write(x : Params);
    action
        out         read_read(x : Params);
end interface pipe

module new_reader() return reader is

    while true
        await read_read(?x : Params);
        end await
end module

module new_pipe() return pipe is
    while true
        await write_write(?x : Params)
        ⇒ read_read(?x);
        end await
end module

Architecture system() return root is
    reader1 is new_reader();
    pipe1 is new_pipe();
connect
    (?x in Params) pipe1.read_read(?x)
to
    reader1.read_read(?x);
end system
```

The application of the translation rules can be seen by comparing the Rapide specification that is obtained from the original Wright specification. First, component *reader* is mapped into interface *reader*. Since its port process *read* only has a single observed event *read*, the reader interface only has an in event *read_read*, (the naming scheme is *port name_event name*). Second, we generate module instances *reader1* and *pipe1* by calling module generators we get from component *reader* and connector *pipe*. Since we only have one attachment from port *read* to role *read* both of which only have one event, we need only one basic connection.

6 Discussion and Conclusion

Let us now return to the two key issues raised earlier: (1) How do we integrate diverse ADLs even in the presence of structural and semantic mismatches? (2) Does Acme help?

With respect to the problem of mismatched features in the ADLs, given the differences between the three ADLs (and especially between Wright and Rapide), it was clear from the outset that complete integration would not be possible. That is, we recognized that there would be certain kinds of system descriptions possible in one ADL that had no counterpart in one of the others. However, rather than attempting to get complete coverage, we attempted to find a scheme that would allow a significant subset of the systems to be mutually accessible in the integrated system. The key challenge was to make that subset as large as possible. To accomplish this we used three techniques that have general applicability to any integration effort of this kind.

1. **Limit the class of systems:** We avoided some of the difficulties in translation by excluding certain systems. In particular, we excluded Wright specifications that use nested parallelism. Ideally (as is the situation here), this does not impose a serious limitation on the class of system that can be handled.
2. **Limit the directionality:** A key factor in making the integration work was an early decision not to attempt to map Rapide specifications back to Wright. By avoiding the reverse translation problem, we avoided problems of handling features of Rapide that had no counterpart in Wright. For example, we did not have to handle the Rapide features for creating new architectural structures at runtime.
3. **Provide a semantics-based translator:** We were able to convert most Wright specifications into Rapide specifications using straightforward mappings. This helped us deal with a number of representational discrepancies that are likely to appear when attempting to integrate two arbitrary ADLs. These include: use (or not) of first class connectors; different treatments of nondeterminism; different control structures (e.g., functional versus imperative descriptions); and different constraints on nesting (e.g., nested parallelism).

While each of these techniques is useful, it is worth noting that each has a downside that must be carefully considered. Limiting the class of systems makes the translation easier, but may exclude the very systems that one is hoping to analyze. Limiting the directionality clearly restricts the ability to incorporate native descriptions from ADL's that aren't mapped back. (For example, our system cannot handle native Rapide specifications directly – we can only exploit Rapide tools on descriptions that are initially generated from Wright and Aesop.) Providing a translator helps bridge semantic gaps, but it also may make it difficult to relate the analyses produced by one ADL to the descriptions viewed in another.

With respect to the cost-effectiveness of Acme, we found that Acme reduced costs in two significant ways. The first was by providing useful infrastructure on which to build: The parsing and manipulating libraries substantially reduced the amount of code we had to write, and provided a convenient framework in which to apply the property translation tool. The second benefit was in providing a single representation that more than two ADLs could share. For example, we got Aesop compatibility (and hence a graphical front end) almost for free, since Aesop was already able to read and write Acme descriptions.

In conclusion, we believe that while it is important not to generalize too far beyond the specific examples of this case study, the goal of integrating diverse ADLs through Acme shows considerable promise. Acme does not magically make semantic differences between ADLs disappear, but it does provide a framework within which one can achieve significant benefits by combining the toolsets of different ADLs. The primary challenge in doing this is to understand what classes of systems can be handled and what kinds of translation are required to map between the various ADL-specific tools. While the specific techniques will differ from ADL to ADL, there appear to be several general strategies for accomplishing the translation. We have illustrated several in this paper, but expect the repertoire to grow as others attempt similar integration efforts.

References

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[All97] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[FDR92] *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, 1.2β edition, October 1992.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Ontario, Canada, November 1997.

[LAK$^+$95] 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.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[MT97] Nenad Medvidovic and Richard N. Taylor. Architecture description languages. In *Software Engineering – ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, Zurich, Switzerland, September 1997. Springer. Also published as Software Engineering Notes, Vol 22, No 6, November 1997.

Acme Specification of the Reader-Pipe Example

```
System acmespec = component {
    ports : {
    };
    representations : {
        acmespec = configuration {
            components : {
                reader1 = component {
                    ports : {
                        read = port {
                            representations : {
                            external : CFamWrightRep = “ {{ Name {WrightSpec1}}
                                {Data {spec1.wright}} {Tool {Component_editor}}}”;
                            };
                        };
                    };
                    representations : {
                        external : CFamWrightRep = “ {{ Name {WrightSpec1}}
                                {Data {spec1.wright}} {Tool {Component_editor}}};”
                    };
                };
            };
            connectors : {
                pipe1 = connector {
                    roles : {
                        read = role {
                            representations : {
                                external : CFamWrightRep = “ {{ Name {WrightSpec1}}
                                {Data {spec2.wright}} {Tool {Connector_editor}}}”;
                            };
                        };
                        write = role {
                            representations : {
                                external : CFamWrightRep = “ {{ Name {WrightSpec1}}
                                {Data {spec2.wright}} {Tool {Connector_editor}}}”;
                            };
                        };
                    };
                    representations : {
                        external : CFamWrightRep = “ {{ Name {WrightSpec1}}
                                {Data {spec2.wright}} {Tool {Connector_editor}}}”;
                    };
                };
            };
            Attachments : {
                pipe1.read to reader1.read
            };
        };
        bindings : {
        };
    };
};
```

Figure 4: Acme Reader-Pipe System