Bridging the gap between Acme and UML for CBD

Miguel Goulão
Departamento de Informática
Faculdade de Ciências e Tecnologia - UNL
2825 Monte de Caparica, Portugal
+351 212948536 (ext. 10731)
miguel.goulao@di.fct.unl.pt

Fernando Brito e Abreu
Departamento de Informática
Faculdade de Ciências e Tecnologia - UNL
2825 Monte de Caparica, Portugal
+351 212948536 (ext. 10707)
fba@di.fct.unl.pt

ABSTRACT
Architecture Description Languages (ADLs) provide formality in the description of software architectures, but are not easily reconciled with day-to-day development concerns, thus hampering their adoption by a larger community. UML, on the other hand, has become widespread throughout industry and academia and can be used as a bridge from architectural to design elements. In this paper we use the new UML 2.0 modeling abstractions to contribute with a mapping from Acme (a second generation ADL which contains the most common ADL constructs) to UML thus allowing architectures to be expressed in a mainstream notation. The mapping uses UML light-weight extension mechanisms and includes OCL well-formedness rules for the mapped models. The feasibility of this mapping is demonstrated through several examples. Ultimately, our proposal helps in the transition from architecture to implementation, using UML design models as a middle tier abstraction.

Keywords
Software component specification, ADLs, Acme, UML.

1. INTRODUCTION
Software architectural descriptions provide an abstract representation of the components of software systems and their interactions. There are three main streams of architectural description techniques: ad-hoc, OO techniques and ADLs.

As pointed out in [1], the use of ad-hoc, informal and idiosyncratic notations that combine the use of box and line diagrams, along with textual descriptions, is still widespread. These descriptions cannot be analyzed for consistency or completeness and cannot be traced back and forward to actual implementations. All these drawbacks are related to the lack of formality of these ad-hoc notations.

To overcome those drawbacks, one can use formal notations known as architecture description languages (ADLs), such as Aesop [2], Adage [3], C2 [4], Darwin [5], Rapide [6], SADL [7], UniCon [8], MetaH [9], or Wright [10]. There are toolsets that support them, such as graphical editors, consistency checkers and performance analyzers. Although with a considerable overlap on the core, each ADL focuses on different aspects of software architecture. While some ADLs, such as C2, are particularly suited for a particular architectural style – as user interface intensive systems, in this case – others, such as Wright, are more general purpose. This diversity provides us with different approaches to solve specific families of problems. However, the interchange of information between different ADLs becomes a major drawback. Developing a single ADL providing all the features of the various ADLs would be a very complex endeavor. Instead, an ADL called Acme [11] emerged as a generic language which can be used as a common representation of architectural concepts in the interchange of information between architectural descriptions originally developed with different formalisms. Acme includes constructs for the common core elements of other ADLs. Rather than developing mappings for all combinations of ADLs, we only need to provide mappings from other ADLs to Acme and vice-versa [12].

A shortcoming of ADLs is that although they allow for architecture in-depth analysis, the rigor of formal methods is not easily reconciled with day-to-day development concerns. Object-oriented approaches to modeling, on the other hand, are more widely accepted and implemented in the industry. In particular, the UML [13] has become both a de jure and de facto standard. Using it to describe software architectures could bring economy of scale benefits, with better tool support and interoperability, as well as lower training costs.

As we will review in the related work section, there have been several attempts to bridge the gap between special-purpose modeling languages (ADLs) and mainstream OO modeling notations, namely UML. One motivation for these attempts is to bring architectural modeling to a larger community, through the use of mainstream modeling notations. Another is to provide automatic refinement mechanisms for architectures. UML can be used as a bridge from architectural to design elements [14]. However, the question arises for the UML adequacy for this purpose. Several roads can be pursued to answer it:

- map the ADL’s concepts to “plain vanilla” UML;
- use the UML extension mechanisms such as stereotype and profile specifications to constrain the semantics of UML’s meta-classes, so that the constrained meta-classes offer a closer match to the ADL’s constructs;
- extend the UML meta-model, in order to include new meta-classes to represent ADL’s constructs.

The first approach has the advantage of using UML as is. By doing so, we can use all the standard UML tools. The tradeoff is that we must use UML model elements to represent concepts that often do not semantically match what we wanted to represent in the original ADL. This often leads to the second option, which may require some sort of adaptation of existing UML tools. However, the latter helps to achieve a higher level of similarity between the concepts expressed by the ADL and UML representations. In this paper, we will follow this second approach. Finally, the third approach would result into a
language that is not fully compatible with standard UML and, as such, without readily available tool support.

OO methods have a few advantages in the representation of component-based systems, when compared to ADLs. There is a widespread notation, an easier mapping to implementation, commercial tools support and well defined development methods. But they also have some shortcomings. OO connections are represented through method invocation which makes them considerably less expressive than their ADL’s counterparts – in ADLs connections are first class design elements. OO methods usually provide a weak support to hierarchical decomposition of architectures, so that the views with a progressive level of detail are harder to create and synchronize. They offer little or no support to the definition of system families, and to the definition of non-functional properties. The latter problem was one of the motivations for aspect-oriented design [15]. The relatively weak support to the integration of different views remains a problem in OO notions. Hofmeister et al [22] used UML to express multiple architectural views and concluded that it was more suited to describe the structural aspects than the dynamic ones. They complemented UML with ROOM [23] to describe protocols and used tables to express correspondences between the different views.

Instead of using the current UML 1.5 version, we will represent the concepts covered by Acme using the candidate UML 2.0 metamodel, which has been partially approved by the OMG recently [16]. This increases our modeling power due to the new features of the upcoming version standard, mainly in what concerns component representation and hierarchical decomposition of architectures.

This paper is organized as follows. Related work is discussed in section 2. Section 3 briefly presents the concepts of Acme that we intend to represent with UML. Section 4 provides an overview of the new UML concepts that play a significant role in this mapping. Section 5 contains a formal specification of the mapping between Acme and UML. Section 6 includes a discussion of the virtues and limitations of that mapping. Section 7 summarizes the conclusions and identifies further work.

2. RELATED WORK

A number of mappings between the concepts expressed in ADLs and their representation with UML have been attempted. In [17], UML is used “as is” to express C2 models. A problem with this approach is the conceptual mismatch between the architectural concepts and the semantics of the UML metamodel elements.

The mismatch problem becomes more evident in [1], where Garlan presents several UML metamodel elements as valid options to express each of the structural constructs defined in Acme. If different design elements seem valid options to express the same architectural concept, then this is evidence that there is not an adequate, biunivocal match. Each mapping becomes the best candidate depending on the goals of the translation from Acme to UML. While some mappings allow for more expressiveness, others foster the understandability of the resulting design model at the cost of loosing some syntactic sugar and semantic information.

In [14], Egyed and Medvidovic describe a development environment, called SAAGE, which allows expressing a C2SADEL architecture by means of a UML model consistent with the original architectural model.

In [18] UML’s extension mechanisms are used to show how to express architectural models defined in C2 and Wright. The basic idea is to create stereotypes for the UML metamodel elements, using OCL (Object Constraint Language) [13] to express the constraints on the original metamodel elements. These constrained metaclasses are then used to express the software architecture concepts. In order to avoid the use of other UML metamodel elements in the architectural descriptions, invariants of the system architecture include OCL clauses (ie, well-formedness rules for the translated models). A shortcoming of this approach is that one mapping from the ADL to UML is required for each ADL. Therefore, a different set of stereotypes is required for mapping each ADL.

Another alternative is to use one of UML’s standard profiles as a starting point for the mapping. An example of this can be found in [19], where the UML-RT profile is used to express Acme’s architectures. Although mismatches still occur, they are mitigated when compared to the generic UML notation. In particular, the notion of connector in UML-RT is much closer to the notion of connectors in ADLs than generic UML.

Based on a mapping from Acme’s components to a combination of UML subsystems, components and composite classifiers, Selic suggested some adjustments to the UML metamodel [20], to increase the semantic accuracy of his mapping. Although the discussion of the details of such mapping and UML extension proposals go beyond the scope of this paper, it is worth mentioning that this sort of proposals have contributed to the body of knowledge that lead to the development of the new UML metamodel. Table 1 summarizes the mentioned proposals:

<table>
<thead>
<tr>
<th>Table 1 – ADL to UML Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping Approach</td>
</tr>
<tr>
<td>Use UML “as is”</td>
</tr>
<tr>
<td>Proposals: [1][17]</td>
</tr>
<tr>
<td>Use UML lightweight extension mechanisms</td>
</tr>
<tr>
<td>Proposals: [14][18][19]</td>
</tr>
<tr>
<td>Extend UML metamodel</td>
</tr>
<tr>
<td>Proposal: [20]</td>
</tr>
</tbody>
</table>

In what concerns the potential completeness of this mapping approaches, Hilliard argued that for some architectural viewpoints, such as reliability, performance or security modeling, it might be more reasonable to use existing notations and techniques rather than to somehow map them to UML [21]. However this makes the integration of the several viewpoints harder.

The approaches discussed in this paper try to bridge the gap between software architecture and design using an OO modeling notation. All of the above mentioned mappings were performed
with UML 1.x, whereas in our paper we use the new UML 2.0 metamodel elements, which enhance the language’s suitability for component-based design. Other approaches try to unify software architecture with implementation. For example, ArchJava is a Java extension that integrates software architecture with Java implementation code [24]. It provides traceability between architecture and code. It also guarantees that direct communication between components in the architecture’s implementation requires their interconnection in the specification.

3. ACME

Acme is a generic ADL which supports four different aspects of architecture: structure, properties, constraints and styles.

From a structural point of view, a system consists of a graph where nodes represent components and interactions between nodes are represented by connectors. Each component is a computational element or data store in the architecture and has several interfaces, known in Acme as ports. Like components, connectors also have interfaces called roles, which can be attached to the participating components. Each role is attached to a component’s port. Both components and connectors can be further detailed with a representation. The binding between the higher and lower levels of abstraction on the architecture description is achieved through a rep-map. All structural elements described above can have extra-structural properties attached to them. Properties are used to document relevant architectural details and consist of a name, an optional type and a value. Acme allows the specification of design constraints, expressed as predicates, which can be used to validate the corresponding architecture. Constraints may be either defined as invariants or as heuristics. A violation of an invariant should be treated as an error, whereas a violation of a heuristic should generate a warning, in Acme supporting tools. Finally, Acme allows the definition of styles of systems. These are used in the definition of domain or application-specific design vocabulary. The style definition also includes constraints on the application-specific design elements. To illustrate some of these concepts, we provide a small extract of an Acme specification of a client-specific design elements. To illustrate some of these concepts, we provide a small extract of an Acme specification of a client-server architecture, adapted from [11].

The style definition also includes constraints on the application-specific design elements. To illustrate some of these concepts, we provide a small extract of an Acme specification of a client-server architecture, adapted from [11]. This case study is also the base for our mapping examples.

4. UML 2.0

The UML 2.0 metamodel is divided in two complementary specifications: the infrastructure [25] (already approved by OMG) defines the foundational language constructs; the superstructure [26] (currently under evaluation) defines the user level constructs. We will use the UML 2.0 proposal by the U2 Partners (http://www.u2-partners.org/). When compared to versions 1.x, the new version of UML has some advantages which are relevant to this research, such as:

- formal definition of its underlying meta-model;
- increased support for representing software components, ports, and interfaces (provided or required), as well as for the hierarchical decomposition of components.

The UML 2.0 will also include OCL 2.0 [27].

5. MAPING ACME INTO UML

Throughout this section, when appropriate, we will include OCL code using the courier font, always preceded by a natural language explanation of its rationale both for auxiliary OCL functions and well-formedness rules included in the proposed mapping. We start by defining IsAcmeComponent().

Predicate 1: IsAcmeComponent() is a predicate that evaluates to true if the model element is a component stereotyped with <<AcmeComponent>>.

class: IsAcmeComponent(): Boolean =
self.isOclType(Component) and
self.extension->exists(e | e.ownedEnd.type.name = “AcmeComponent”)

More predicates are also defined for similar Acme constructs:
IsAcmeConnector(), IsAcmePort(), IsAcmeRole(), IsAcmeProperty(), IsAcmeSystem().

5.1 Components

A component in Acme corresponds to a computational element or data store in the architecture. It may have several ports, several properties, a representation with several bindings (defined as rep-maps) and a set of design rules. An UML component represents a modular part of a system, which encapsulates its contents and whose manifestation is replaceable within its environment. The component’s behavior is defined through its required and provided interfaces. To avoid mixing Acme’s components with other concepts that we will also represent with
UML components, we created a stereotype for Acme components named <<AcmeComponent>>, using Component as the base class.

**Invariant 1:** All ports attached to <<AcmeComponent>> are either <<AcmePort>> or <<AcmeProperty>>. No other required or provided interfaces are allowed.

```plaintext
class Component inv:
  self.IsAcmeComponent() implies
  self.ownedPort->forAll(ap| ap.IsAcmePort() or ap.IsAcmeProperty()) and self.IsAcmeComponent() and
  self.ownedPort->forAll(ap| ap.IsAcmeRole() or ap.IsAcmeProperty()) and self.IsAcmeComponent() and
class Component inv:
  self.IsAcmeComponent() implies
  self.ownedPort->forAll(ap| ap.IsAcmePort() or ap.IsAcmeProperty()) and self.IsAcmeComponent() and
  self.ownedPort->forAll(ap| ap.IsAcmeRole() or ap.IsAcmeProperty()) and self.IsAcmeComponent() and
```

**Predicate 2:** Evaluates true if no required or provided interfaces are allowed for a component.

```plaintext
Invariant 1 and others that will follow use the predicate HasNoOtherInterfaces().

Component::HasNoOtherInterfaces() : Boolean =
  self.required->isEmpty() and self.provided->isEmpty()
```

### 5.2 Ports

Acme’s ports identify points of interaction between a component and its environment, so they can be viewed as the component’s interfaces to the rest of the system. These can be as simple as operation signatures, or as complex as collections of procedure calls with constraints on the order in which they should be called. UML ports are features of classifiers that specify distinct points of interaction between the classifier (in this case, the component) and its environment (in this case, the rest of the system). UML ports have required and provided interfaces. A UML interface represents a set of coherent public features along with their rights and obligations. Rights and obligations are expressed as constraints (such as pre and post conditions) or protocol specifications which may impose ordering restrictions on interactions through the interface. We use a combination of UML port and corresponding required and provided interfaces to express Acme’s port concept. However, the combination of UML’s ports and interfaces is not an exact match to Acme’ ports. UML ports can be used with other classifiers besides components. Interfaces can be used in contexts other than defining points of connection between components and other model elements. For instance, a class may implement several interfaces. Therefore, we need to restrain their usage.

**Invariant 2:** <<AcmePort>> can only be a port to an <<AcmeComponent>> and is typed by exactly one provided and one required interface.

```plaintext
context Port inv:
  self.IsAcmePort() implies
  self.owner.IsAcmeComponent() and (self.required->size()==1) and (self.provided->size()==1)
```

Ports are represented in UML diagrams as rectangles on the edge of the component and can be adorned with a name. See, for example, Figure 1: send_request and receive_request are ports declared for the client and server components, respectively. In this example, both ports are typed by a required and a provided interface.

### 5.3 Connectors

Acme connectors represent interactions among components. Since they provide the glue for architectural designs, they are viewed as first class elements in the architecture community. Several options can be considered for their representation with UML. A first one would be to represent them using UML’s assembly connector, as in Figure 1.

![Figure 1 - Using UML’s assembly connectors](image)

In this example, the components client and server are connected directly. This is visually appealing but has a few drawbacks. Connectors in Acme may be much more complex than a simple interfaces’ match. They can be, for example, a protocol, or a SQL link between two components (a client and a database). With this representation, we would lose some architectural expressiveness. Another drawback is that when using components built by different teams it is normal that their interfaces do not match exactly. The connector may provide the necessary glue between the components and this must be made explicit in the design. In order to represent the concept of connector, has no semantic equivalent in UML, we use a stereotyped component named <<AcmeConnector>>.

**Invariant 3:** All ports attached to <<AcmeConnector>> are either <<AcmeRole>> or <<AcmeProperty>>. No other required or provided interfaces are allowed.

```plaintext
context Component inv:
  self.IsAcmeConnector() implies
  self.ownedPort->forAll(ap| ap.IsAcmeRole() or ap.IsAcmeProperty()) and self.HasNoOtherInterfaces()
```

Although representing a connector with a stereotyped component clutters the outcome design, it offers the ability to represent the connector as a first class design element, with flexibility in the definition of any protocols it may implement. Consider the example in Figure 2, where the components client and server have interfaces that do not match (PSendRequest and RSendRequest, on the client side, vs. RReceiveRequest and PReceiveRequest, on the server side), but the rpc connector implements a protocol to make the components interact. We have included provided and required interfaces in both ends of the connector, to illustrate that it provides bi-directional communication abilities. The Acme concept of connector can provide bi-directional glue between the attached components, unless it is constrained.

![Figure 2 – Using the <<AcmeConnector>>](image)
5.4 Roles

Roles are related to connectors (in Acme) the same way as ports are related to components. Thus, it makes sense to represent Acme roles as constrained UML ports, through the use of the <<AcmeRole>> stereotype.

**Invariant 4:** <<AcmeRole>> can only be a port to an <<AcmeConnector>> and is typed by exactly one provided and one required interface.

```plaintext
context Port inv:
  self.IsAcmeRole() implies
  self.owner.IsAcmeConnector() and
  (self.required->size()==1 and
   self.provided->size()==1)
```

5.5 Systems

An Acme system represents a graph of interacting components. The UML’s concept of package (with the standard <<subsystem>> stereotype) would be a suitable candidate for representing a system but has at least two disadvantages:

- a package represents a set of elements, rather than a structure containing them;
- a package is not suitable for defining system-level properties.

To mitigate those problems we use the constrained component stereotype <<AcmeSystem>>.

**Invariant 5:** All the components used in an <<AcmeSystem>> are either <<AcmeComponent>>, or <<AcmeConnector>>.

```plaintext
context Component inv:
  self.IsAcmeSystem() implies
  self.contents()->select(el|el.IsKindOf(Component))->asSet()
  ->forAll(cmp|cmp.IsAcmeComponent() or
      cmp.IsAcmeConnector())
```

**Invariant 6:** All ports used in an <<AcmeSystem>> can be <<AcmePort>>, <<AcmeRole>> or <<AcmeProperty>>.

```plaintext
context Component inv:
  self.IsAcmeSystem() implies
  self.contents()->select(el|el.IsKindOf(Port))->asSet()
  ->forAll(prt|prt.IsAcmePort() or
      prt.IsAcmeRole() or
      prt.IsAcmeProperty())
```

**Invariant 7:** All ports attached to <<AcmeSystem>> are either <<AcmePort>>, or <<AcmeRole>>. A system has no other interfaces.

```plaintext
context Component inv:
  self.IsAcmeSystem() implies
  self.ownedPort->forAll(ap|ap.IsAcmePort() or
      ap.IsAcmeProperty()) and
  self.HasNoOtherInterfaces()
```

5.6 Representations

Acme’s representations provide the mechanism to add detail to components and connectors. Rep-maps are used to show how higher and lower-level representations relate to each other. Since we have used stereotyped components to represent both components and connectors, we will use the features for packaging components of UML 2.0 to express representations. UML provides two wiring elements (in the UML specification, they are referred to as “specialized connectors”): assembly and delegation. The former provides a containment link from the higher level component to its constituent parts, while the latter provides the wiring from higher level provided interfaces to lower level ones, and from lower level required interfaces to higher level ones. A delegation corresponds to Acme’s rep-map concept.

**Invariant 8:** All assembly connectors used in Acme systems wire together <<AcmeComponents>> to <<AcmeConnectors>>. In other words, all assembly connectors must be attached to an <<AcmePort>> and an <<AcmeRole>>.

```plaintext
context connector inv:
  self.kind = #assembly implies
  self.end()->exists(cp|cp.role.IsAcmePort()) and
  exists(cr|cr.role.IsAcmeRole())
```

Coming back to our case study, Figure 3 details the specification of server. The wiring between the internal structure of server – a system which contains a topology with three components and the connectors between them – and the server’s own ports is achieved with the usage of the <<delegate>> connectors. Although Acme explicitly uses the concepts of representation and system for defining subsystems, we opted to make these implicit in our mapping. Making them explicit would not improve the expressiveness of the resulting design and would only clutter the diagram by creating an extra level of indirection.

**Figure 3 - Defining subsystems**

5.7 Properties

Properties represent semantic information about a system and its architectural elements. In UML 2.0, components have an external view and an internal one. To allow automatic reasoning on the design properties, using OCL, we can make these properties available outside the component’s internal scope. Ports can be typed with a provided interface that allows the component user to access its properties.
The downsides of representing Acme properties as UML ports are that by doing so we are cluttering the design and extending the interfaces provided by the design element.

**Invariant 9:** An `<<AcmeProperty>>` port owns a single provided interface that must provide get and set operations for the property’s value and type.

```plaintext
class Port inv:
    self.IsAcmeProvided() implies
        (self.required->isEmpty()) and
        (self.provided->size()=1)
```

### 5.8 Constraints (invariants and heuristics)

Constraints allow the specification of claims on how the architecture and its components are supposed to behave. While invariants are conditions that must hold at all times, heuristics are constraints that should hold, although breaking them is possible. In UML, we can express design constraints through OCL. These constraints can be pre-conditions, post-conditions or invariants. Acme’s notion of invariant can be directly mapped to its OCL counterpart. However, there is no direct UML semantic equivalent for the notion of heuristic. This could be circumvented by creating the `<<AcmeConstraint>>` stereotype as a specialization of the UML Constraint metaclass. The former would have an enumerated attribute with two allowed values: invariant and heuristic.

### 5.9 Styles and Types

An architectural style or type defines a vocabulary of design elements and the rules for composing them. It is used in the description of families of architectures. Since we have created stereotypes for the several UML constructs used in this Acme to UML mapping, we can now specify architectural styles using these stereotyped elements.

**Figure 4: The pipe and filter family**

Figure 4 represents the pipe and filter family, an architectural style that defines two types of components, FilterT and UnixFilterT, a specialization of FilterT. The architectural style is defined by means of a UML package, as the family definition does not prescribe a particular topology. It does, however, establish an invariant that states that all the connectors used in a pipe and filter system must conform to PipeT.

### 6. DISCUSSION

The presented mapping from Acme to UML is more straightforward than previous approaches. This mainly results from the increased expressiveness provided by the new UML design elements. From a structural viewpoint, representing a topology is fairly simple when using UML. This is mainly due to the relative closeness of the sort of structural information that we want to express both at the architectural and design levels. In both cases we have to identify components and the connections among them, possibly at different levels of abstraction.

However, while a connector is regarded as a first class design element by the architecture community, it has no direct mapping in UML. Our proposal is to promote connectors to first class design elements, by representing them as stereotyped components. This seems to be a good option, if we consider that the evolution of CBD should provide us with an increasing number of off-the-shelf components and that, by doing so, the complexity of building component-based software is shifting to the production of glue code. Representing connectors as stereotyped components gives us the extra flexibility to meet this challenge.

The representation of extra-functional properties is not an easy nut to crack. Perhaps they could be more suitably defined at the meta-level, but this still requires further research.

Although Acme allows associating several representations to the same design element, it offers no support for the synchronization of these multiple views (specified as systems). Neither does our mapping, in the definition of sub-systems. We plan to use aspect-oriented design to deal with this problem.

Heuristics are also complex to map directly to UML, as UML provides no direct representation for this concept, although we can use OCL to deal with this problem.

Since Acme does not provide a direct support for component dynamics specification, in this paper we do not address it. Nevertheless, we could use properties to annotate the architectural entities with information on their expected behavior. For instance, a connector may have a property specifying its protocol with some formalism (e.g. Wright). We could use UML’s behavioral modeling features similarly, thus complementing the structural information in the mapped specification with a behavioral specification of the design elements used.

### 7. CONCLUSIONS AND FURTHER WORK

We have shown the feasibility of expressing architectural information expressed in a mainstream ADL (Acme) using the UML 2.0 design language. It is possible to obtain a mapping from a given ADL to UML, through a two-step approach. We could first map the architecture from the original ADL to Acme and then use the mapping proposed in this paper to obtain the corresponding specification in UML. Details lost in the ADL to Acme conversion can always be added later to the resulting UML specification. The proposed mapping builds upon the added expressiveness of UML 2.0 for architectural concepts, when compared to UML’s previous versions. In particular, the availability of components with ports typed by provided and required interfaces has proved to be a step forward in the exer-
The proposed mapping focuses mainly on structural aspects and design constraints. Although it also points out to ways of dealing with the definition of system properties, including semantics and behavioral specification, further research is required to provide more specific guidance on these aspects.

8. REFERENCES