24

An Extensible ADL for Service-Oriented Architectures

R. Bashroush and I. Spence

Abstract

While architecture description languages (ADLs) have gained wide acceptance in the research community as a means of describing system designs, the uptake within the service-oriented architecture (SOA) domain has been slower than might have been expected. A contributory cause may be the perceived lack of flexibility and, as yet, the limited tool support. This chapter describes ALI, a new ADL that aims to address these deficiencies by providing a rich, extensible and flexible syntax for describing component and service interface types and the use of patterns and meta-information. These enhanced capabilities are intended to encourage more widespread ADL usage.

Keywords Architecture description languages · Service-oriented architectures

1. Introduction

In recent years, architecture description languages (ADL) have emerged as potential tools for formally describing system architectures at a reasonably high level which enables better intellectual control over the system [1]. ADLs model not only system structure, but also address component behavior specification as well as communication protocols. While some ADLs provide graphical notations (e.g. boxes and lines), others also provide textual notations.

Architecture descriptions can also be used as a communication vehicle among the different stakeholders. With the formality introduced by ADLs to the architecture description, more architectural analysis of qualities such as consistency, modifiability and performance can be carried out on the system at an early stage. Although it is not clear yet what aspects of the architecture should be included or excluded from the architecture description (e.g. behavior, structure, interfaces), it is widely agreed within the ADL community that software architecture is a set of components (or services) and the connections among them conforming to a set of constraints.

Although some ADLs have been put to industrial use [2], the majority of ADLs have not scaled up and remain confined to small-scale case studies. Yet, little adoption of ADLs has been witnessed within the SOA domain. A number of potential limitations demonstrated by current ADLs were identified in previous work [3]. Among these limitations are over constraining syntax, single view presentation of the architecture and lack of tool support. The ALI ADL has been designed to address these limitations. The rationale behind the ALI notation was discussed in [3]. Among the main concepts driving the ALI notation are flexible interface description, architectural pattern description, formal syntax for capturing meta-information and linking the feature and architecture spaces. ALI built on our experience with the ADLARS [4]

R. Bashroush and I. Spence • ECIT, Queen's University Belfast, Belfast, UK.

G.A. Papadopoulos et al. (eds.), *Information Systems Development*, DOI 10.1007/b137171_24,

ADL and adopted many of the solution space provided by ADLARS such as its support for software product lines.

In this chapter, we introduce the different parts of the ALI notation to show how the goals of [3] are realized in the language. ALI comprises seven parts:

1. *meta types*, which provides a notation for capturing meta-information
2. *interface types*, which provides a notation for creating types of interfaces
3. *connector types*, where architectural connectors are defined
4. *component types*, where architectural components are defined
5. *pattern templates*, where design patterns are defined
6. *features*, where the system features are catalogued
7. *system*, where the system architecture is described

In the following sections, the different parts of the ALI notation are discussed. Section 9 concludes with a discussion.

2. Meta Types

Meta types provide a formal syntax for capturing (meta-)information related to the architecture. A meta type is defined by the information it contains. The information is captured within fields, where each field has a data type (text, number, etc.) and a name (*tag*). Consider the example below for defining a meta type called *MyMetaType1*:

```
meta type MyMetaType1 {
   tag creator, description: text;
   tag cost, version: number;
   tag edited*: date; }
```

In this example, the keyword "`meta type`" is used to start a meta type definition. `MyMetaType1` is the name of the meta type being specified. Each meta type contains a number of tags which can be either textual, numeral or date (if needed, the tag types could be extended to include enumeration, character, etc.). In the example above, five tags are defined: two textual, two numeral and one date. The date tag "*edited*" is marked with an asterisk "*" to indicate an optional tag.

Once meta types are specified, *meta objects* conforming to these types can then be created throughout the architecture. These meta objects are attached to architectural elements (e.g. components, connectors) to provide a corner for appending additional information related to these elements. Below is an example meta object that conforms to the meta type given in the example above.

```
meta: MyMetaType1 {
   creator: "John Smith";
   cost: 5,000;
   version: 1;
   edited: 12-02-2006;
   description: "A GUI component ... "; }
```

A meta object could also conform to more than one meta type. It is also possible to create meta objects that do not conform to any meta type. This enhances the language flexibility. However, little automated analysis can be done over such informally provided information.

The formal specification of meta-information would considerably enhance the development of CASE tool support that could harness these meta objects and conduct automated analysis on the data (e.g., cost/

benefit analysis, project timing/scheduling, based on what meta-information is available). Other meta-information might include design decisions, component compatibility, etc., which, when extracted and formatted using proper CASE tools, allow automated architecture documentation to be achieved on the fly.

In general, it is expected that the meta types will be created once and used repeatedly within different systems developed by the same enterprise. A standard set of information required (tags) may be first identified by the project management team (or any other stakeholder), and then provided to architects to conform to. This insures that critical information is always provided within an architecture description. The flexible syntax also allows the architects to augment this information with fields (tags) that they may need temporarily or internally within the architecture team.

3. Interface Types

Interface types have been introduced to ALI to allow for the usage of multiple interfaces within a system description. The practice would be to create a set of common interface types needed within an application domain once (e.g., WSDL, IDL, Invocation), and then use these interfaces in the design of components and systems.

The interface type definition is divided into two sections:

- *Syntax definition*, where the syntax of the interface description is specified using a subset of the JavaCC [5] notation.
- *Constraints*, where the interface binding (connectivity) constraints are specified. These include
  - *Should match*: Here the terms (identified in the syntax definition section using the JavaCC notation) that should match between two interfaces to be considered compatible (allowed to bind) are identified. For example, in a functional interface, for two interfaces to be compatible, the function names and argument types should match.
  - *Protocols supported*: A list of the protocols that this interface type can support for communication is provided, e.g., IIOP, HTTP, method invocation.
  - *Allow multiple bindings*: This is a Boolean value that states whether multiple binding is allowed on this interface. For example, this property is set to true on a server socket interface to allow for binding multiple client socket interfaces. On the other hand, it is set to false on the client socket interface.
  - *Factory*: This is a Boolean value that states whether the interface is a factory. A factory interface means that when a connection request is received on this interface, a new connection dedicated interface is created to handle that particular request while the main interface continues to listen to new incoming requests. For example, server socket interfaces in java are factories. On the other hand, C++ sockets are not. In C++, the factory functionality is to be implemented by the programmer if needed.
  - *Persistent*: This is a Boolean value which when set to true indicates a persistent interface (the internal data of the interface component are kept unchanged after the current connection has ended) and when set to false indicates a transient interface (internal data are reset to initial values when the current connection is terminated).

Below is an example for defining an interface type *functional*:

```
interface type functional {
  syntax definition:    {
   "Provided" ":"  "{"
    [ "function"   <PROV_FUNCTION_NAME>
        "{"
           "impLanguage"  ":"
```

```
                                    <PROV_LANGUAGE_NAME> ";"

                          "innvocation"  ":"
                                    <PROV_INVOCATION> ";"
                          "paramterlist" ":"
                             "("[  <PROV_PARAMETER_TYPE> [","
                              <PROV_PARAMETER_TYPE:]* ]? ")" ";"
                       "return type"  ":"
                              <PROV_RETURN_TYPE> ";"
                     "}"]*    "}"
                   // Required:  etc.
                }
                constraints:  {
                   should match: {
                  PROV_INVOCATION_NAME,
                   PROV_PARAMETER_TYPE
                          }
                   protocols supported: {  RMI-IIOP, JRMP }
                   allow multiple bindings: false;
                   factory: false;
                   persistent: false;
                }
             }
```

For further details about the notation used for specifying the interface syntax, please refer to JavaCC [5].

It is important to emphasize here that the interface type definition is not meant to be read by humans, but rather created once and then read by CASE tools that would verify the interface descriptions and bindings made throughout the architecture definition.

4. Connector Types

As in Acme [6] and other ADLs, connectors are considered first-class citizens in ALI. Below is a simple example of a connector type definition:

```
connector type SOAP/HTTP {
   interfaces {
      a, b of type WSDL;
   }
   layout {
      if (supported(FULL_DUPLEX_FEATURE))
            connect a and b;
      else
            connect a to b;
   }
}
```

The connector type definition consists of two parts:

- *interfaces*: Where the connector interfaces are defined. These resemble the input/output terminals of the connector. A connector must have at least two interfaces (for input/output) while theoretically there is no restriction on the maximum number of interfaces allowed. For example, a bus connector would need to have a number of bi-directional interfaces to serve all components connected to the bus. On the other

hand, a simple connector like the one in the example above has only two interfaces (of type WSDL, where WSDL is an interface type that should be defined in the interface type section).

- *layout*: The layout section describes the internal configuration of the connector. It shows how the connector interfaces are connected internally, that is, how the traffic travels internally from one interface to another. There are two types of configurations allowed between connector interfaces:
  - *unidirectional connections (to)*: Which specify that the data/requests received on one interface to be output on another interface. This is done using the keywords: "`connect`" and "`to`". For example, `connect a to b`; outputs the data/requests received on the *a* interface to the *b* interface.
  - *bi-directional connection (and)*: Which specify that the data/requests received on one interface be output on another interface and vice versa. This is done using the keywords: "`connect`" and "`and`". For example, `connect a and b`; outputs the data/requests received on the *a* interface to the *b* interface and vice versa. The keyword "`all`" can be used to connect a connector interface to all other interfaces of the connector using a bi-directional or unidirectional communication as described above. For example, `connect a to all` makes the input on interface *a* available as output on all other interfaces of the connector. In contrast, `connect a and all` makes the input on *a* available on all other interfaces and the input on all other interfaces available on *a*. The statement: `connect all to all` can be used to create bi-directional connections among all ports (`connect all and all` is not defined).

As with interface types and meta types, a set of connector types can be defined per domain which can then be reused across multiple projects within that domain.

In the example given above, the connector definition is linked to the system feature model to allow for connector customization based on features selected. This is done using the `if /else` structure and the keywords "`supported /unsupported`." So, in the example above, if the system supports the FULL_-DUPLEX_FEATURE, interfaces *a* and *b* are connected as bi-directional (using "`and`"); otherwise, they are connected as unidirectional (using "`to`"). This syntax introduces a high level of configurability to the connector definition which provides better support for defining configurable and product line architectures.

Meta objects can be attached to connector types by simply defining the meta object (as explained in Section 2) inside the connector type definition (anywhere between the start and end brackets).

5. Component Types

Component type definition forms a crucial part of the ALI notation. In this section, a very brief description is given due to space limitation.

The component type definition consists of two sections:

- *interfaces*: which specifies the different component interfaces. These interfaces are described conforming to defined interface types (included in the interface type section). A component can have one or more interfaces of different types.
- *subsystem*: Where the internal structure (subsystem) of the component is described. The subsystem section is divided into three sections:
  - *Components*: where the different subcomponents included within the component are defined.
  - *Connectors*: where the different connectors to be used in connecting subcomponents are defined.
  - *Configuration*: where the way in which subcomponents are connected is described. Three methods can be used to connect components:
    a. *Using connectors*: where a connector mediates the connection between two or more components.
    b. *Direct connection*: where component interfaces are bound directly without the use of a connector.

c. *Using patterns*: where predefined connection patterns can be used to connect a set of components according to a selected architectural pattern. More details on architectural patterns are given in the next section.

Below is an example of a component type definition:

```
component type MyComponentType1
{
//a meta object attached to the component type
 meta: MyMetaType1 {
  description: "this is an example component";
  cost: 20,000;
}
 interfaces:   {
  // specifying a functional interface
  myInterface1 of type functional {
       Provided: {
           function myAddFunction
    {
      impLanguage: "Java";
      invocation: "add";
      parameterlist: ( "int" );
      return: "void";
      } // etc.
        }
        Required: { }
    //no required functions specified
   }
 if(supported(Provide_WSDL_Interface_Feature))
 {
    myInterface2 of type WSDL {
      // WSDL interface description
    }
 }
}
sub-system: {
  components {
   comp1  <custom_feature_set1>: ComponentType1;
    if( supported(Some_Feature_A))
     comp4  <custom_feature_set4>:
                          ComponentType3;
    else
     comp4  <custom_feature_set5>:
                          ComponentType3;
     //etc.
  }
  connectors {
   conn1  <custom_feature_set1>: ConnectorType1;
   // etc.
   }
  configuration {
```

```
//1 - connecting components using connectors
 connect comp1.interface1 with conn1.a;
 connect comp2.interface1 with conn1.b;
//2 - connecting components without connectors
 bind comp3.interface1 with comp1.interface2;
//3 - connecting components using patterns
 if( supported(Some_Feature_B) ){

   Client_Server(ServerComponent1.interface1,
     [ ClientComponent1.interface1,
         ClientComponent2.interface1,
         ClientComponent3.interface2]
       );
 } } }
```

In the example above, we begin the component description using the keyword '`component type`' followed by the component type name, `MyComponent1` in this example.

The first section of the component definition contains a meta object which conforms to meta type `MyMetaType1`.

The second section is the component interfaces section where two interfaces are defined: `myInterface1`

of type *functional* (an interface type that was defined as an example in Section 3) and `myInterface2` of type WSDL that only exists if the feature `Provide_WSDL_Interface_Feature` is supported by the system.

We could define as many interfaces as we wish, where we could link the existence of interfaces to the `support`/`unsupport` of system features. We could also attach meta objects to interfaces simply by defining them within the scope of the interface definition (somewhere between the two curly brackets of the interface definition).

It is recommended that interface definitions conform to defined interface types as per the example above (*functional* and *WSDL* types). However, to allow for maximum flexibility, it is possible to define interfaces that do not conform to any predefined interface type, in which case, no analysis or automated tool support can be enabled over that interface definition or any connection made over it (similar to the concept of creating arbitrary meta objects that do not adhere to any meta type definition). This is done by dropping the interface type name that follows the interface name in the interface definition. For example, one could define a port-like interface without having an interface type readily available:

```
myPortInterface3 :
{
input in1, in2, in3;
output out1, out2, out3;
}
```

However, it will not be possible to verify whether the connection between this interface and any other interface within the system is valid or not (as the interface syntax and constraints are not formally defined). This could be practical at early design stages when the exact interface type specification is not clear. When the interface type matures enough throughout the design process, an interface type is defined for this type of interface, and then the interface type name is appended to the interface definition above to allow for verification, and perhaps automated analysis with the aid of appropriate CASE tool support.

The third section in the component definition is the description of the sub-system. In the example above, three components are defined in the components section, each customized with a different feature set. Also, a component of type `ComponentType3` is defined; however, its customization is dependent on the existence of the feature `Some_Feature_A`.

Similarly, a number of connectors are defined in the connectors section within the subsystem description.

The configuration section shows how the components and connectors defined in the subsystem section are configured (connected). As explained earlier, there are three ways in which components can be connected and these are demonstrated in this example.

6. Pattern Templates

The ALI notation allows for the definition and usage of architectural patterns. This is done using *pattern templates*. Pattern templates are first defined and then used throughout the architecture with a simple call to the pattern template needed. Pattern templates take as an argument the interfaces to be connected according to the pattern template definition.

Pattern templates are defined in similar way to the definition of functions (methods) in programming languages. A pattern template definition contains

- *Pattern name*: a unique pattern name.
- *Arguments*: the set of interfaces to be connected. Single interface and/or arrays of interfaces can be passed as arguments. In the case of arrays of interfaces as arguments, the minimum and maximum number of interfaces passed can be specified.
- *Definition*: the specification of how the interfaces are to be connected (the pattern). The syntax used for defining patterns is very simple and provides support for
  - *connecting interfaces*: using the same syntax used in the connections section of the connector type definition (discussed in Section 4).
  - *defining loops*: to allow for connecting arrays of interfaces. The syntax used here is the same syntax used in C for creating *for* loops. Note here that the arrays of interfaces start at index 1 and not at 0 (like in C).

Below is an example that defines a `Client /Server` pattern:

```
pattern templates:
{
  Client_Server( server : InterfaceType1,
                 clients [1..N] : IntefaceType1 ) {
          for( i = 1 ; i   <= N ; i++)
            connect clients[i] and server;
      } }
```

In this example, the `Client _Server` pattern takes as an argument one interface called server of type `InterfaceType1`, and an array of interfaces called `clients` (with `[1..N]` meaning a minimum of one client interface) of type `InterfaceType1`. The pattern is defined as for all N clients, create a bi-directional connection with the server interface (refer to Section 4 for more details on the use of the keywords: "`connect` ", "`and` ", and "`to` " for connecting interfaces).

An example of how to invoke the `Client /Server` pattern template to connect a number of component interfaces was given within the example in Section 5.

7. Features

The feature description section provides a catalogue of the features used within the system. The feature definition consists of

- *Alternative names*: In many cases, different groups within the development process refer to the same feature using different names. This part of the feature definition keeps track of the different names (if any) that are used to reference the same feature (within the different design and development groups involved in the project).
- *Feature parameters*: A feature can carry a number of parameters (textual, numerical, etc.). For example, if the feature is "Manual Gearbox", the parameter would be the "number of gears" available (a numerical value).

Below is an example of how features are defined in ALI:

```
features {
   featureA {
      alternative names: {
          Developer.X, Evaluator.F112
      }
      parameters: {
           (windowTitle: text),
           (windowWidth,windowHeight: number)
      }
   }
   // etc.
}
```

In the example above, `featureA` was defined showing that it is referred to as "`X`" by the development team and as "`F112`" by the evaluation team. The feature encompasses three parameters: one textual and two numerical.

The features defined in this section are usually extracted from the feature model of the system. This is carried out at a prior stage of embarking on the architecture design. CASE tools could be used to read feature models and populate this section (work on this aspect is ongoing in our group). This is an important part of the notation as it makes ALI independent of any particular feature modeling technique.

8. System

Finally, the system section is where the overall product (or product line) architecture is specified. The syntax used in this section is the same as the syntax used in the subsystem section (described in component types, Section 5) with the major difference that the system section is not contained within any component definition but rather provides the description of the overall system architecture (rather than a subsystem of a component). As a result, the keyword "`external`" can be used in the `system` description section to reference interfaces of external systems (when needed) providing a means of capturing the system interaction with its environment (operating system, other systems, etc.).

Below is an example of the overall structure of the `system` section showing how the `external` keyword could be used to reference external interfaces (parts similar to the example given in Section 5 are replaced with "`...`" due to space limitation):

```
system {
    components { ... }
    connectors   { ...}
    configuration  {  ...
       bind comp1.interface with external.windowHandleAPI;
    }
}
```

9. Discussion

Potential limitations within existing ADLs which could be discouraging their use within the SOA domain and restricting their application to small-scale case studies were discussed in [3]. Restrictive syntax/structure, lack of tool support and single view presentation are among the limitations identified. In this chapter we have presented the different parts of the ALI notation which were designed to address the identified limitations. ALI built on our experience with ADLARS [4] and introduced a blend between flexibility and formalism. While flexibility gives freedom for the architect during the design process, formalism allows for architecture analysis and potential automation using proper CASE tool support (e.g. on-the-fly architecture documentation, code generation).

Among the new concepts in ALI, the notation provides no predefined interface types. Instead, ALI introduces a sublanguage that gives users the flexibility to define their own interface types. Also, the notation focuses on capturing architectural meta-information and introduces formal syntax (*meta types* and *meta objects*) for this purpose.

Continuing the theme of flexibility, ALI permits the user significant scope for defining architectural patterns. In essence, patterns may be defined and instantiated in similar fashion to function calls in programming languages.

Among the successful concepts adopted from ADLARS, ALI supports the relationship between components, connectors, patterns etc. in an architecture description and features in the feature model using first-order logic. This direct link between the architectural structure and the feature model [7] allows the capture of complex relationships that might arise between the two spaces in real-life systems.

The textual notation described in this chapter serves as a central knowledgebase for the architecture description. CASE tools may then be used to extract the necessary information from this knowledgebase to be presented as different views of the architecture. The centralized approach would help alleviate multiple architectural views mismatch when the different views are maintained separately [8].

As for future work, two items top the list for the work on the ALI project. The first is to develop a CASE toolset for the notation. The toolset will benefit from the experience gained with designing the *ADLARS Development Studio* [9, 10]. And the second is to explore the potential for providing *round-trip* to code. The ability to go from architecture to code and back seems to be attracting more interest and momentum in industry (e.g. the work on model-driven architecture, MDA[11]).

References

1. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architecture: Methods and Case Studies*: SEI series in software engineering. Addison-Wesley, 2002.
2. R. v. Ommering, F. v. d. Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." *IEEE Computer*, pp. 78–85, March 2000.
3. R. Bashroush, I. Spence, P. Kilpatrick, and T. Brown, "Towards More Flexible Architecture Description Languages for Industrial Applications." V. Gruhn and F. Oquendo (Eds.). *EWSA 2006, Lecture Notes in Computer Science*. Vol. (4344), pp. 212–219, September 2006.
4. R. Bashroush, T. J. Brown, I. Spence, and P. Kilpatrick, "ADLARS: An Architecture Description Language for Software Product Lines." *In proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, Greenbelt, Maryland, USA, April 2005. pp. 163–173.
5. "The Java Compiler Compiler [tm] (JavaCC [tm]) – The Java Parser Generator.," https://javacc.dev.java.net/.
6. D. Garlan, R. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems." In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman (Eds.) Cambridge University Press, 2000, pp. 47–68.
7. T. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan, "Weaving Behavior into Feature Models for Embedded System Families." *In proceedings of the 10th International Software Product Line Conference SPLC 2006*, Baltimore, Maryland, USA, August 2006. pp. 52–64.
8. J. Muskens, R. Bril, and M. Chaudron, "Generalizing Consistency Checking Between Software Views." *In Proceedings of the 5th International Working Conference on Software Architecture, WICSA-05*, Pittsburgh, PA, November 2005. pp. 169–180.

9. R. Bashroush, I. Spence, P. Kilpatrick, and T. J. Brown, "Deriving Product Architectures from an ADLARS Described Reference Architecture using Leopard." *ACM SIGSOFT Foundations of Software Engineering FSE-12*, October 2004.
10. R. Bashroush, I. Spence, P. Kilpatrick, and T. J. Brown, "Towards an Automated Evaluation Process for Software Architectures." *In Proceedings of the IASTED International Conference on Software Engineering SE 2004*, Innsbruck, Austria, February 2004. pp. 54–58.
11. OMG, "Model Driven Architecture," http://www.omg.org/mda/