# ENHANCING THE ROLE OF INTERFACES IN SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES (ADLS)

Seamus Galvin, J.J. Collins, Chris Exton and Finbar McGurren
*Software Architecture Evolution (SAE) Group, Dept. of Computer Science and Information Systems, University of Limerick, Limerick, Ireland*

**Abstract:** One of the key reasons why ADLs are yet to be adopted commercially on a large scale is due to shortcomings in their ability to describe adequate interface specifications. An interface specification that is vague, lacking in detail, too style focused or too language-specific results in an ADL description with a restricted scope of use. This paper demonstrates how an XML-based ADL (xADL 2.0) can be extended to model detailed, meaningful interface specifications, and is used as part of a simple prototype to demonstrate how they form an integral part of an architectural description, paying particular attention to interface-level constraints. The approach is based on the principle that an ADL's interface modeling features should provide sufficient flexibility to allow them to reflect stakeholder's interface concerns at all stages in the lifecycle.

**Key words:** Software Architecture; ADLs; interface specification; interface constraints.

## 1. INTRODUCTION

Architecture Description Languages (ADLs) provide a structured means of representing a system's architecture that is both human and machine-readable, and have been proposed as a modeling notation to provide support for some of the problems experienced in architecture-based development [1]. However, the diverse nature of existing ADLs indicates a lack of clarity with respect to the kind of language an ADL should be and how it should be used. Some ADLs have a narrow usage scope, providing highly specific support during the early stages of architectural analysis, while others perceive varying degrees of relationship between architectural description and the

underlying implementation. Also, it is still unclear how ADLs and their associated descriptions might interrelate with other design and runtime artifacts, such as requirements and domain models, modeling tools, implementation platforms and execution engines. The lack of such relationships minimizes the ADL's potential.

While these realities present a broad range of problems, a fundamental requirement is the provision of adequate interface modeling capabilities. An accurate interface description is an essential part of an architectural specification, and a key requirement of architectural stakeholders at all stages in the project lifecycle [2]. It is pivotal to an ADL's malleability as constrained and monolithic interface support results in a limited ADL. Most importantly, it is required to establish an accurate relationship between architectural description and the underlying implementation, allowing the ADL to support maintainable and evolvable software.

The remainder of the paper is summarised as follows. Sections 2 and 3 provide a brief overview of ADLs, the latter paying attention to their diverse approaches to supporting interface description. Section 4 discusses interface modeling concerns in architectural documentation. Section 5 discusses two approaches to defining an ADL meta-language for interface description using the XML-Schema standard. Section 6 discusses the application of the second approach to a suitable ADL, section 7 demonstrates an example of its application and section 8 offers conclusions and discusses future work.

## 2. OVERVIEW OF PROMINENT ADLS

The goal of achieving architecture-driven development and architecture-centric evolution presents many diverse challenges, and this is reflected in the ADLs that have been developed throughout the research community, some of which are listed in Table 1. Despite the lack of a common, universally acceptable definition of software architecture [3], all adopt a relatively standard approach to modeling basic architectural structure. Most recognise an architecture as being a set of components (or modules) whose interactions are represented as connectors [1]. Relationships between components and connectors are represented as links (or attachments). Some ADLs allow components to be directly linked to one another; others require them to communicate via connectors, while some allow direct links between connectors. ADLs allow these basic architectural elements to be refined to various degrees, thus allowing the notion of a sub-architecture to be modelled. They also attempt to distinguish between template definitions of architectural elements and instances of those elements that are defined in actual instantiations of a particular architecture.

While there is a general conformance to this core set of architectural concepts, there is significant diversity in their detailed ADL features, e.g. naming conventions and keywords used, their scope of concern, and their associated technologies and tools. For example, Darwin, Wright and Rapide have focused on static and dynamic analysis of abstract architectural descriptions; Aesop has investigated the customisation of architectural design environments; Unicon has identified and implemented commonly occurring connector abstractions; ArchJava investigates communication integrity; SOFA highlights the relationship between architecture and component-based middleware; xADL aims to support rapid prototyping and tailoring to assist ADL research, and ACME and ADML strive for ADL standards and recognition. Some of these ADLs have a narrow scope of use, providing high-level, conceptual support during the early stages of architectural analysis. Others perceive a relationship between the architectural description and the underlying implementation. The nature of this relationship varies; some ADLs provide code generation facilities, while others aim to represent the architecture explicitly in the underlying system. Both conceptual and more concrete ADLs have yet to receive widespread acceptance. This indicates a difficulty in providing an architectural language that is sufficiently conceptual to support high-level abstractions, yet simultaneously capable of supporting and governing the code level in an acceptable manner. Important issues still remain largely unaddressed by ADL research – these include the lack of explicit support for the definition of architectural viewpoints [4] and the insufficient emphasis on the relationship between ADLs and other developmental and runtime artefacts such as requirements models, design models, domain models, modelling tools and languages, implementation platforms and execution engines.

## 3. ADL SUPPORT FOR MODELING INTERFACES

An overview of the interface modeling characteristics of a broad range of existing ADLs (including UML) is given in Table 1. Different keywords are used, e.g. ports, roles, players, interfaces. They are difficult to compare and categorise, as ADL interface modeling characteristics are often influenced by their primary intended use. For example, if the ADL is geared for formal analysis, then the interface modeling support is influenced by the formal methods used (e.g. Wright, Darwin, Rapide), or if an ADL is closely aligned to a particular architectural style or underlying platform, style or platform specific interface features are provided. This is sufficient to fulfill the ADL's primary intent, at the expense of constraining its capability in other respects. ADLs and modeling notations aiming for a broader usage scope allow

detailed interface features to be represented as user-defined properties (e.g. ACME and UML). This allows any interface feature to be modeled, but the language's native tools cannot interpret them, leaving this task to the ADL user, and hindering the possibility of a more standardized representation. Also, as this approach does not provide specific syntax for the features within the core ADL, it is more difficult to clarify feature semantics.

*Table 1.* ADL support for modeling interfaces

| ADL | Type | Keyword | Key Features | Semantic Modeling |
|---|---|---|---|---|
| **Aesop [5]** | Implementation independent | Ports and Roles | Allows ports/roles to be associated with a style | Semantics associated with certain architectural styles |
| **ACME [6]** | Implementation independent | Ports and Roles | Allows ports/roles to be associated with a style | User-defined properties |
| **ADML [7]** | Implementation independent | Ports and Roles | XML version of ACME – same as above | Same as ACME |
| **Wright [8]** | Implementation independent | Ports and Roles | Uses CSP notation to capture interaction semantics | Port/Role behaviour can be modeled in CSP |
| **Darwin [9]** | Implementation independent | Interface | Interfaces can contain provides and requires services, focuses on bindings between interfaces specified in pi-calculus | Supports parameterization and subtyping, portal semantics added using tags |
| **Rapide [10]** | Implementation independent | Interface | Can model synchronous and asynchronous features, advanced parameterization and subtyping possible | Poset (event) patterns characterised using behaviour and constraint declarations |
| **xADL 2.0 [11]** | Implementation constraining | Interface | C2-specific features | C2-specific semantics |
| **Unicon [12]** | Implementation constraining | Players and Roles | Players/roles associated with specific component and connector types | Specific properties associated with port and role types |
| **ArchJava [13]** | Implementation constraining | Roles | Java-like syntax, ports can have provides, requires, broadcast features | Provides features can include a Java implementation |
| **C2SADL [14]** | Implementation constraining | Interface | Can model C2-specific provides/requires features | C2-specific semantics |
| **UML** | Implementation independent | Interface | "Lollipop" or rectangular notation can be used | Semantics modeled using properties and tagged values |

## 4. INTERFACE CONCERNS IN ARCHITECTURAL DOCUMENTATION

In order to develop adequate support for the high-level, platform independent modeling of interfaces during the early stages of the project lifecycle, one should take cognizance of the features found in the interface

documentation of architectural specifications. Bachmann et al. suggest a standard organization for architecture-level interface documentation [2]:

1. Identity - This identification may include versioning information.
2. Resources provided - This includes syntactic (e.g. name, arguments etc.) and semantic information (i.e. the implications of using the resource).
3. Data types - user-defined data types declared on the interface.
4. Errors raised by interface resources.
5. Configuration information - This might involve the passing of parameters to the interface.
6. Quality attribute characteristics.
7. What the associated element requires from its environment.
8. Rationale and design issues - This may be a narrative description of the motivation/considerations behind the interface's design.
9. Usage guide - This allows stakeholders to gain a better view of the interface's overall role. This can be achieved by identifying and depicting resource usage scenarios that the architect expects to repeatedly occur.
10. Exceptions - These could be errors on the part of the actor invoking the resource, or errors that occur due to software or hardware events that result in a violation in the element's assumptions about its environment.

Like all aspects of architectural documentation, some of the discussed features are structured and should be carried through and directly reflected in design and implementation. Other information is prose based and is intended to enhance the understanding of the structured information. The structured architectural information would ultimately be much more useful and accurate if it was a part of the system, rather than being part of the associated documentation. Therefore, the ADL's language features should be sufficient to allow the modelling of the structured information by providing explicit syntax for relevant features, and should also support the formal or semi-formal specification of features where required (in order to support the latter, it should facilitate more than one formal notation if required). Also, ADL modelling tools should allow this structured information to be annotated with relevant prose-based information.

## 5. TOWARDS AN ADL META-LANGUAGE FOR INTERFACE DESCRIPTION

A suitable ADL interface meta-language should facilitate the definition of abstract, language-independent interfaces and should also support their transition into concrete, language-specific ones. Also, it should be extensible – this is required to support the diversity in existing ADLs, architectural

documentation, development platforms, and the many different, and possibly unanticipated ADL usage contexts. Also, the approach should reflect the commonality that exists between interface modeling features of the different platforms. This will allow each platform's interface modeling requirements to be defined in terms of a common format and will ease the future addition of features related to other, possibly newer platforms. It will also facilitate the definition of structured mappings between language-independent and language-specific interfaces, a trait which is advocated by the Model Driven Architecture (MDA)[15]. This section discusses the potential of two possible approaches to providing a suitable interface meta-language for ADLs. The experimentation discussed in this section is applied to the interface features of Java, C# and OMG's IDL3, the latter a part of the CORBA Component Model (CCM). To ease the application and extension of the approaches, both have been defined using the XML-Schema standard.

## 5.1 First approach – generic language-independent schema and language-specific extensions

The first candidate approach is based on the premise that the common, or overlapping features of language-specific interface specifications provide a basis from which an ADL's language-independent interface features can be built. Their specific, differing features act as a basis for refinement in detailed design. For example, all of them exhibit notions of identity, resources provided and input/return parameters. Most of them allow properties or constants to be defined – these can be primitive or user-defined. However, while they allow syntax to be specified unambiguously, they generally do not support the specification of resource semantics [16], a void which could be addressed by an ADL.

To investigate the potential of this approach, the overlapping interface modeling features of Java, C# and IDL3 were identified and represented in an XML-Schema as complex types. This provided the basis for the language-independent interface description. Based on the language-independent schema in the study, their differing features were then modeled as separate schema complex types. Each of these types used the XML-Schema extension capabilities to extend the language-independent complex types. To demonstrate the potential for modeling semantics, the generic schema also contains features for specifying design-by-contract [17] constraints. The relationship between the language-independent and language-specific schemata for this experiment is shown in Figure 1. The language-independent schema contains three nested complex types that support the modeling of optional and mandatory features, including interface identity, references to extended interfaces, preconditions, postconditions,

invariants, operation names, and input/return parameters. The three language-specific schemata contain specialist features outside this generic set. For example, the Java specific complex types (JavaInterfaceDefinition JavaInterfaceBody and JavaInputParam) contain the AccessType feature for optionally specifying the interface's accessibility (i.e. public, private, protected etc.) and features for constant declarations, references to exceptions, inner interface declarations and inner class declarations. In a similar fashion, C# interface specifics are represented in C_SharpInterfaceDefinition, C_SharpInterfaceBody and C_SharpInputParam, and IDL3 specifics in CCMInterfaceBody, and CCMInputParam. C# specific features include attributes, access types, properties, indexers, and events and support for specifying input parameters as being of type value, output, reference or array, while IDL3 specific features include events, constants, types, attributes and exceptions, and also support for specifying input parameters as 'in', 'out' or 'inout'.

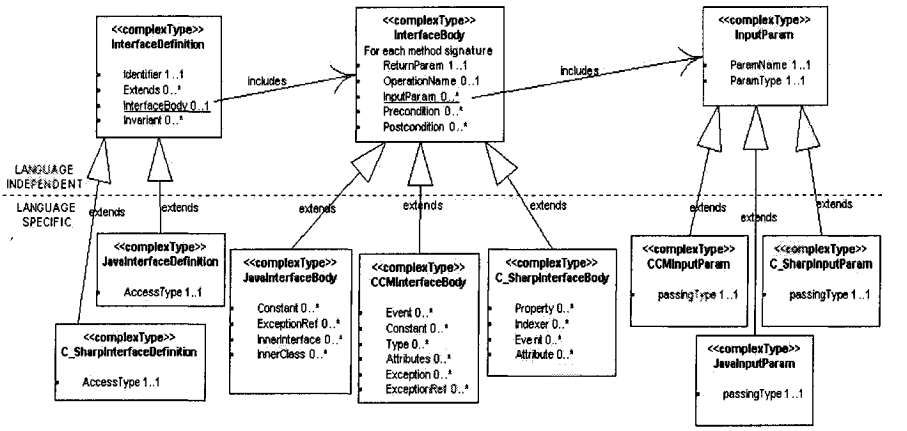


*Figure 1.* Conceptual overview of first approach - interface features of Java, C# and IDL3. Common features are represented in the language-independent schema, differing features are modeled in extension schemata.

The language-independent schema is a good representation of the main interface features in commonly used programming languages and middleware technologies, but it has shortcomings that hinder its potential to support all of the previously discussed guidelines. First, the approach does not support the establishment of a stable boundary between the sets of features contained in the language-independent and language-specific levels. Most notably, the set of language-independent features depicted in Figure 1 would become smaller if more interface examples from other programming

languages and middleware technologies were factored into the approach. Also, as the features contained in the language-independent interface are directly based on those in language-specific interfaces, changing it in response to syntactic changes in platform-specific interface modeling features, or to accommodate new language-specific platforms would be cumbersome. The approach supports language-specific concerns well, but its ability to support language-independent concerns is restricted by its narrow feature set, which restricts its ability to extensively support the criteria for architectural documentation in Section 4. The narrow language-independent feature set would also restrict its ability to support the simulation and analysis of language-independent architectural models at an early stage in the lifecycle, like other ADLs such as Rapide. Also, the schema layout in the approach is rather unintuitive, making associated tool support more difficult to construct.

## 5.2 Second approach - core set of feature declarations

The second approach involves the definition of a broad set of individual feature declarations and their use as the basis for constructing the language-independent and language-specific schemata (Figure 2). It consists of three parts - the first being the core set of feature declarations, the second part being a language-independent schema that is defined using the core feature declarations, and the third part which facilitates the definition of language-specific interface schemata. A language-specific schema may use features from the core set as a basis for definition, and may additionally define exclusive features to represent interface features that are not supported by the core set.

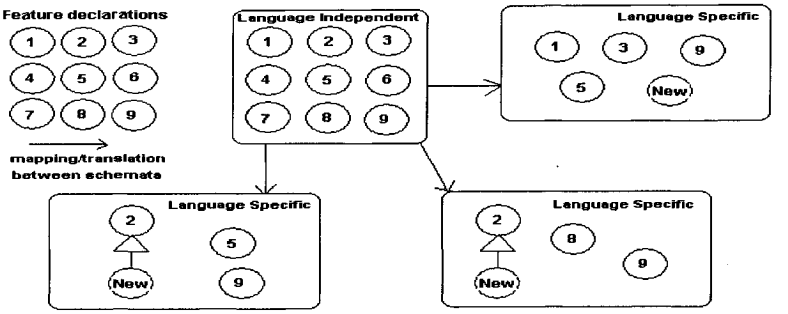


*Figure 2.* Conceptual view of interface model

As interface descriptions based on the schemata are defined in XML, XSL stylesheets could be used to transform language-independent interface descriptions into language-specific ones, and/or to generate code if required. Some of the defined language-independent features are also explicitly

represented in corresponding language-specific schemata (e.g. InterfaceName, ExtendedInterface, ResourceDeclaration and ResourceName). Other language-independent features may be realised in the implementation of an element that implements the interface at a language-specific level, rather than being explicitly represented in the language-specific interface. For example, the language-independent schema in Figure 3 contains features for modeling the configuration parameters of an interface, but the Java specific interface schema does not contain this feature. Instead, a Java implementation might represent the configuration parameters in the constructor of a class that implements a corresponding Java interface. In this case, a translation could generate corresponding code outside the ADL's scope of description if a particular usage context required it.

*Table 2.* Core types in interface model

| **Feature** | **Overview of Semantics** |
|---|---|
| AccessType | Represents the accessibility scope of the interface. |
| Array | Represents a CORBA-like array declaration on an interface. |
| AttributeDeclaration | Represents a CORBA-like attribute declaration – attributes indicate the variables in an element that are accessible to clients. |
| ConfigParam | Facilitates the configuration of elements through configuration parameters (e.g. specification of size of data structure element that implements interface). |
| ConstantDeclaration | Represents the declaration of a CORBA-like constant declaration on an interface. |
| Enum | Represents a CORBA-like enumerated type declaration . |
| Event | Represents an event declaration. Elements that implement the interface will either publish the event or subscribe to it. |
| Exception | Represents the specification of any exceptions that can be raised by resources declared in the interface. |
| ExceptionRef | Represents any references to exceptions that may be made in an interface-level resource declaration. |
| ExtendedInterface | Represents a reference to an interface that is extended by this interface. ADLs and platforms generally allow an interface to extend multiple interfaces. |
| InOutParam | Represents an 'inout' parameter which combines value and return parameters, allowing a calling method to pass and receive a value. |
| InParam | Represents a value (or input) parameter. Parameter passed by method caller (actual parameter) is copied into the parameter used by the called method (formal parameter) when the method is invoked. |
| InterfaceName | Represents the identity of the interface – may also include versioning information. |
| Invariant | Represents specification of interface invariants, i.e. constraints enforced for all elements that implement the interface. |
| I_OptParam | Represents the optional specification of a formal parameter. Implicitly supported by some programming languages in the form of variable length arrays. However, in the context of documenting an architecture's interfaces, one may prefer to represent optional parameters explicitly. |
| I_RefParam | Represents a reference parameter – address of formal parameter is the same as actual parameter. Subtle difference between reference and result parameters is that any changes to the formal parameter immediately affect the actual parameter. |
| OutParam | Represents a result (or output) parameter. Value of formal parameter is |

| Feature | Overview of Semantics |
|---|---|
| | copied into the actual parameter when the procedure returns. |
| O_RetParam | Represents a return parameter returned as a function result. |
| PostCondition | Represents a postcondition of a resource, i.e. a description of the effects of that operation on its parameters and element state [18]. |
| PreCondition | Represents a precondition of a resource, i.e. a definition of the situations under which a postcondition will apply [18]. |
| ResourceDeclaration | Represents the amalgamation of information for an interface resource – ResourceName, various types of resource parameter, preconditions, postconditions, references to exceptions etc. |
| ResourceName | Identify of an interface resource. |
| String | Represents a CORBA-like string declaration. |
| Struct | Represents a CORBA-like struct declaration. |

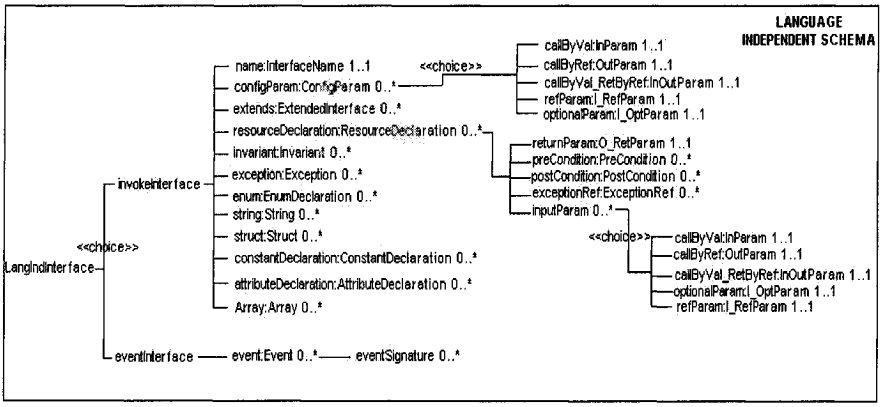


*Figure 3.* Language-independent interface schema

The selection of core interface feature types in Table 2 are influenced by a range of various sources, including the taxonomy in Section 4, Corba IDL, Java, C# and Visual Basic. Some of the features are a fundamental part of most linguistic approaches to modeling interfaces, for example, InterfaceName, ExtendedInterface, ResourceDeclaration, ResourceName and O_RetParam. CORBA IDL also has additional interface features that are intended to epitomize a broad set of language-specific features, and some of them are also suited for inclusion in the set. These are Event, Exception, ExceptionRef, Struct, AttributeDeclaration, ConstantDeclaration, Enum, String and Array. The classification in Section 4 identifies other important features that are not directly supported by Corba IDL, for example the explicit declaration of configuration parameters (ConfigParam) and accessibility (AccessType). The authors also mention resource semantics – while it may be more practical to document some of these concerns as prose, important resource semantics can be characterized as design-by-contract constraints using the PreCondition, PostCondition and Invariant types in the core set. The core set also includes six different types of resource parameters

- these are return (O_RetParam), input (InParam), output (OutParam), input-output (InOutParam), reference (I_RefParam) and optional parameters (I_OptParam). This broad set of core types is shown in Table 2, and is used as the basis for defining the language-independent interface schema in Figure 3.

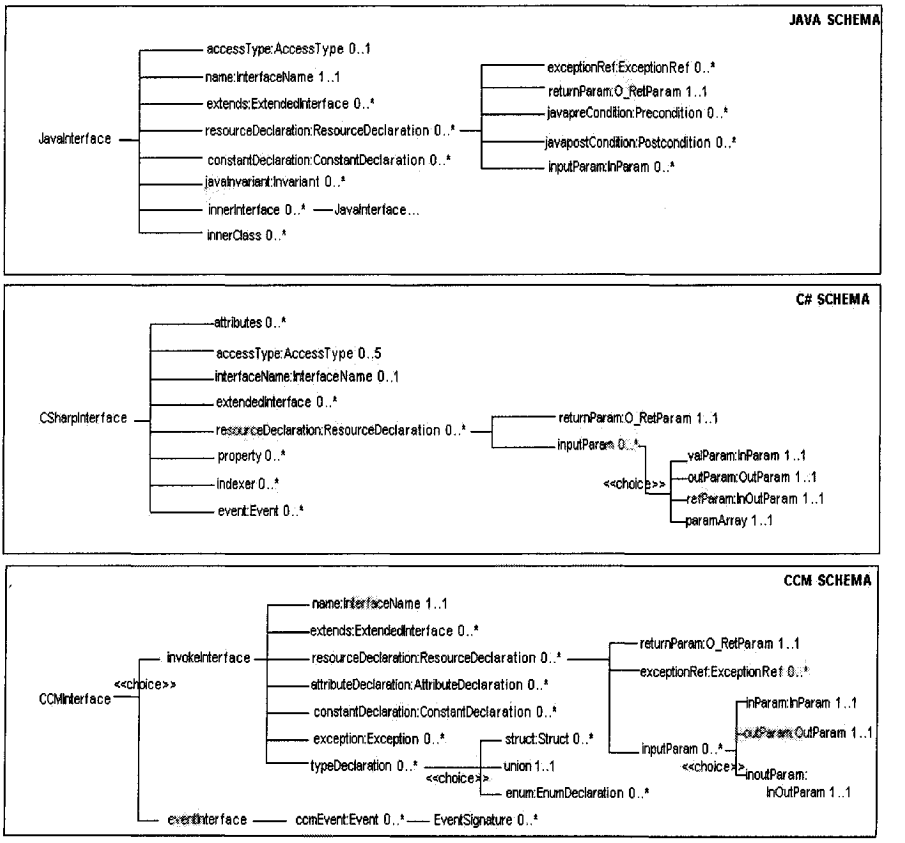


*Figure 4.* Language-specific schemata for Java, C# and IDL3 interfaces

The interface features of Java, C#, and CCM are depicted in Figure 4. While the official language-specifications of these platforms provide a detailed informal description of language semantics, rigourous formal descriptions are not provided. To show how the semantics of language-independent and language-specific interface features interrelate, input parameter features are used as an example. In the core types, five input parameter feature types are declared, each representing different parameter passing semantics – value or input parameters (InParam), output parameters (OutParam), value-result parameters (InOutParam), reference parameters

(I_RefParam), and optional parameters (I_OptParam). The language-independent schema allows a resource declaration to contain any of these features, whereas the three language-specific schemata have different sets of parameter passing semantics. Java only supports input parameters so it only uses InParam. IDL3 supports input, output and value-result parameters, so it uses InParam, OutParam and InOutParam. C# supports four types of input parameter, value, out, ref and param. Value and out use the InParam and OutParam core types respectively. However, despite its name, ref is semantically equivalent to InOutParam rather than I_RefParam. As param is not semantically equivalent to any of the core types, it is defined as a feature unique to the C# schema.

The second approach is influenced by some of the problems identified with the first approach and attempts to address them. In contrast to the first approach, overlapping features in the language-independent and language-specific schemata are defined in terms of the same set of core types, but the actual language-independent and language-specific schemata are independently defined. This means that the addition of new language-specific features do not force change upon the language-independent schema, and the number of features that can be included in the language-independent schema is no longer restricted, giving it the potential to provide broader support for language-independent interface modeling at an early stage in the lifecycle, while still providing adequate support for language-specific refinement at a later stage. Also, the schema layout is more intuitive and easier to apply in comparison to the first approach. Therefore, as the second approach is a more comprehensive solution, it is currently the basis of our future work, and is applied in the remaining sections of this paper.

## 6. APPLYING THE MODEL TO AN ADL

The most recent C2-based ADL (xADL 2.0) provides features for specifying architectural structure and supporting basic reconfiguration [11]. As it is based on XML-Schemas it is compatible with many existing tools. Also, xADL is designed to be extensible, allowing modifications to be made to it more easily than any of the other existing ADLs, and it is therefore used as the basis for experimentation. xADL 2.0's language structure is split across a number of interrelated schemata. The most important are the *Instance* and *Structure&Types* schemata. The *Instance* schema is designed to represent a completed, running architecture that is instantiated from a design-time *Structure&Types* architectural representation. The *Structure&Types* representation allows component, connector or interface elements to be represented as types. Design-time or run-time architectural

topologies can be created using the *Structure&Types* or *Instance* schemata respectively by declaring one or more instances of a type and creating links between them. The interface meta-language is added by replacing xADL's style-specific interface modeling features with a new interface schema containing the core, language-independent and language-specific parts. This modification is made in the *Structure&Types* schema, where the existing construct used to model interface types is extended.

Alternatively, other ADLs such as ACME and Darwin provide support for extensibility in the form of properties or tagged values. Such ADLs can apply the presented interface meta-language by referencing XML interface descriptions that conform to the presented XML-Schemata.

## 7. APPLYING THE INTERFACE EXTENSIONS – STACK EXAMPLE

To demonstrate the approach in practice, the modified ADL is used to represent a stack component (StackImpl) and two interfaces that it implements (Stack and RemStack), using a Java-based design-by-contract framework. As StackImpl is a binary Java class, the Java-specific interface schema is used to provide a concrete definition of Stack and RemStack. These interface definitions also contain design-by-contract constraints (Figure 5). To demonstrate how the interface model can be used to broaden an ADL's set of support, a generator was written to process the ADL description, producing a series of files for each defined interface. Figure 5 depicts StackArch.xml and the files generated from it. First, a language-specific interface file is generated for each interface – in this case, Stack.java and RemStack.java. It also generates a proxy that intercepts all interface invocations and checks the relevant constraints specified in the ADL description. In the example, three files are generated - Stack_StackImpl.java and RemStack_StackImpl.java contain the constraint-checking code, while StackImplProxy.java carries out the runtime checks. The constraint checking mechanism is greatly influenced by DBCProxy, a Java-oriented design-by-contract framework [19]. DBCProxy uses Java's dynamic proxy mechanism and reflection to enforce constraints. However, as this approach is highly Java-specific, it has been used to develop a simpler, static solution that is clearer, more efficient and more applicable to other programming languages.

Instantiation and invocation of the proxy is shown in Figure 6. The proxy is used by calling the static getInstance() method. A variable of interface type Stack is declared and assigned to getInstance(). The proxy contains

boolean switches that enable/disable the assertions for each interface that the component implements. If an interface's constraints are switched on, getInstance() returns a proxy instance, otherwise it returns an instance of the component. Once the client attempts to place an item on the stack, the call is rerouted through the proxy. In this example, StackImplProxy uses Stack_StackImpl to check the relevant assertions. In this case, it firstly checks any preconditions on put(), and then it invokes the actual operation on the component. It then checks any postconditions and finally any interface invariants to ensure that the component is in the correct state.

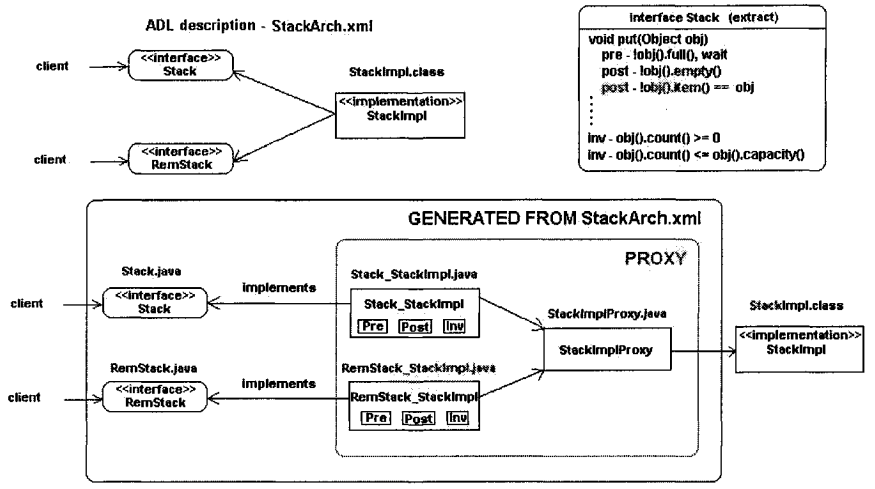


*Figure 5.* Stack Example - files generated from ADL description

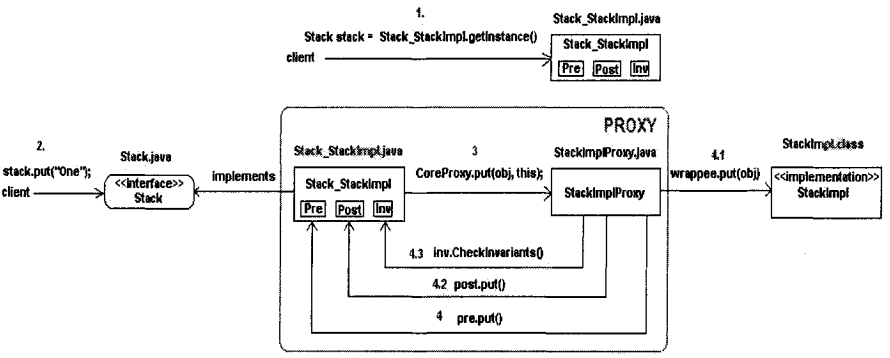


*Figure 6.* Example proxy instantiation and invocation

An ADL-based environment should be able to provide concrete support for exception handling. Beugnard et al. identify four approaches to dealing

with constraint violations [20]. The generated proxy addresses some of these concerns for interface-level constraints:

1. Reject - Raise an exception and propagate it to the client.
2. Ignore - Proceed with the operation, ignoring any adverse effect.
3. Wait - If a precondition fails, this mode defaults to waiting until the precondition becomes true. This synchronization protocol is based on separate objects [21], first used in Eiffel. Obviously, this only applies in concurrent contexts.
4. Negotiate - This involves the renegotiation of the client-server contract, allowing the client to retry the operation, possibly with new values.

The proxy in this example defaults to the Reject mode. The Ignore mode can be applied to any constraint by appending an "ignore" parameter to it. Also, the proxy allows the Wait mode to be applied to a precondition by appending a "wait" parameter, inheriting this feature from the DBCProxy framework. At present, the proxy does not support the Negotiate mode, but future work will investigate the implications of doing so.

## 8. CONCLUSIONS AND FUTURE WORK

The approach discussed in this paper provides a foundation that allows ADLs to support concrete, practical interface descriptions, thereby broadening their scope of use. Its treatment of platform independent and platform specific concerns is pivotal to allowing the ADL description to be a permanent, meaningful artifact from high-level architectural analysis through to maintenance and evolution. It is compliant with the Model Driven Architecture (MDA) philosophy, providing a basis for the transformation of platform independent ADL interface descriptions into platform specific ones. This is a step towards allowing platform independent ADL descriptions to evolve into platform specific ones.

In order to provide precise mappings between language-independent and language-specific interfaces, future work will aim to add further clarification of informal feature semantics, and also to demonstrate how the approach can be used to support a typical component-based development process.

## ACKNOWLEDGEMENTS

This research has been funded by QAD Ireland Ltd and Enterprise Ireland Innovative Partnership grant IP-2003-154.

# REFERENCES


1. Medvidovic, N. and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, 2000. **26**(1).
2. Felix Bachmann, L.B., Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford, *Documenting Software Architecture: Documenting Interfaces.* 2002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.
3. SEI, *How Do You Define Software Architecture?* 2003, Software Engineering Institute (SEI), Carnegie Mellon University.
4. IEEE, *IEEE P1471/D 5.0 Information Technology - Draft Recommended Practice for Architectural Description.* 1999, IEEE Architecture Working Group.
5. Garlan, D., *An Introduction to the Aesop System.* 1995, Carnegie Mellon University, Pittsburgh.
6. David Garlan, R.T.M., David Wile, *Acme: Architectural Description of Component-Based Systems.* Foundations of Component-Based Systems, ed. G.T.L.a.M. Sitaraman. 2000: Cambridge University Press.
7. Unknown, *Architecture Description Markup Language (ADML) - The XML-based standard for IT architecture interoperability and re-use.* 2002.
8. Allen, R., R. Douence, and D. Garlan, *Specifying and Analyzing Dynamic Software Architectures.* Lecture Notes in Computer Science, 1998.
9. Jeff Magee, N.D., Susan Eisenbach, Jeff Kramer. *Specifying Distributed Software Architectures.* in *Proceedings 5th European Software Engineering Conference (ESEC 95).* 1995. Barcelona, Spain.
10. Luckham, D.C. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events.* in *Proceedings of the DIMACS Partial Order Methods Workshop IV.* 1996. Princeton University.
11. Eric M. Dashofy, A.v.d.H., Richard N. Taylor. *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages.* in *Proceedings of the 24th International Conference on Software Engineering.* 2002. Orlando, Florida.
12. Mary Shaw, R.D., Gregory Zelesnik. *Abstractions and Implementations for Architectural Connections.* in *Third International Conference on Configurable Distributed Systems.* 1995. Annapolis, Maryland.
13. Aldrich, J., C. Chambers, and D. Notkin. *ArchJava: Connecting Software Architecture to Implementation.* in *ICSE 2002.* 2002. Orlando, USA.
14. Richard N Taylor, N.M., Kenneth M Anderson, James E Whitehead Jr, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, Deborah L. Dubrow, *A Component- and Message-Based Architectural Style for {GUI} Software.* Software Engineering, 1996. **22**(6): p. 390-406.
15. OMG, *Model Driven Architecture Guide Version 1.0.* 2003, Object Management Group (OMG).
16. Clements, P., et al., *Documenting Software Architectures - Views and Beyond.* 2003: Addison Wesley.
17. Meyer, B., *Applying Design by Contract.* IEEE Software, 1992. **25**(10): p. 40-51.
18. Cheesman, J. and J. Daniels, *UML Components: A Simple Process for Specifying Component-based Software*, ed. A. Wesley. 2000.
19. Eliasson, A., *Implement Design by Contract for Java using Dynamic Proxies.* JavaWorld, 2002.
20. Beugnard, A., et al., *Making Components Contract Aware.* IEEE Computer, 1999: p. 38-45.
21. Katrib, M., et al., *Java Distributed Seperate Objects.* Journal of Object Technology, 2002. 1(2).