

Extension of the Fractal ADL for the Specification of Behaviours of Distributed Components

T. Barros¹, L. Henrio², A. Cansado², E. Madelaine², M. Morel², V. Mencl^{3,4}, and F. Plasil⁴

1) Un. Diego Portales, Santiago, Chile, tomas.barros@udp.cl

2) INRIA Sophia-Antipolis, BP 93, 06902 Sophia-Antipolis Cedex, France
{antonio.cansado, eric.madelaine, matthieu.morel}@sophia.inria.fr

3) United Nations University IIST, Macao, mencl@iist.unu.edu

4) Charles University, Praha, Czech Republic, {mencl, plasil}@nenya.ms.mff.cuni.cz

1 Introduction

Inheriting from a long experience about modules, objects and interfaces, component programming has emerged as a programming methodology ensuring both re-usability and composability. Among the components models, Fractal [6] provides hierarchical composition for a better structure, and specification of control interfaces for dynamic management. Fractal defines a highly extensible component model which enforces separation of concerns, and separation between interfaces and implementation.

The Fractal Architecture Definition Language (ADL) is an extensible language allowing one to define a component architecture to be defined by statically specifying the structure and content of a component system. A component can be instantiated by passing its ADL definition to a factory.

Synchronous and Asynchronous implementations: the Fractal model does not specify how communications are done. There are implementations with synchronous (e.g. Julia [3]) and with asynchronous method invocation (e.g., ProActive [10]), so behaviour specification and verification must respect this heterogeneity. The Dream [2] framework mixes synchronous and asynchronous communications through composite bindings, and in the context of potentially distributed components some implementations directly mixing synchronous and asynchronous communications (depending on the interfaces involved) should come out.

The Grid Component Model is a component model focusing on Grid applications, that is currently being designed inside the CoreGrid Network of Excellence. It is proposed as an extension of the Fractal component model, integrating the specificities of Grid computing that need to be taken into account in the definition of the GCM. Fractal was chosen as the reference model for designing the GCM because it is a simple though extensible model with clear specifications. At some point, the GCM standard will probably lead to a proposal for an extension of the Fractal ADL.

Multicast and Gathercast Interfaces: to meet the specific requirements and conditions of multiway communications (1-to-n, n-to-1 or n-to-n), for distributed components systems in general, and for Grid computing in particular, *multicast* and *gathercast* interfaces give the possibility to manage a group of interfaces as a single entity, and expose the collective nature of a given interface.

Our proposal is articulated in 2 parts. The first part defines the elements required to attach behavioural specifications to Fractal ADL documents. The second part addresses the question of multicast communications within distributed component systems, and of policies for scattering arguments and gathering results. Then in a perspective section we discuss parameterized topologies of components expressing indexed patterns in component composition (sets, arrays, etc), and also representing dynamically created assemblies.

2 Behaviour Protocols and Labelled Transition Systems

The authors of this proposal have a significant experience in the domains of the specification of the behaviour¹ of components, and of verification of their behavioural properties. They use different semantic formalisms

¹ in the ADL keywords we use the American English spelling “*behavior*”

and different underlying models to reach goals that are similar in nature: provide some guarantees that components in a composite system behave smoothly together, and/or respect some user requirements.

Researchers from the SOFA team use “Behavior Protocols” [9], a notation specifying component behaviour in terms of ordering of method invocation events. A behaviour protocol is an expression composed from basic and advanced operators; its syntax stems from regular expressions. The *behaviour compliance* and *consent* relations are defined on behaviour protocols based on their trace semantics, allowing to reason on substitutability and compositional compatibility. The behaviour of a component is specified by its *frame protocol*. In a composite component, the *architecture protocol*, constructed from frame protocols of its subcomponents, is checked for compliance with the frame protocol. For a primitive component, its Java implementation may be checked for compliance with a model checking tool [8].

Researchers from the OASIS team are using parameterized, hierarchical, networks of labelled transition systems [4], and a bisimulation-based semantics, to specify and verify the behavioural properties of component systems. They build finite abstractions of the component system behaviour, taking advantage of the congruence properties of bisimulation theories, and using standard model-checking tools from the CADP toolset [7]. The models are generated from an ADL description and from the associated Java interfaces, and include specific controllers expressing the non-functional aspects of either synchronous (Julia) or asynchronous (ProActive) Fractal implementations [5].

The extension to the Fractal ADL proposed in this article can be used together with any of these models, because the choice of the communication model (and the choice of the semantical formalism) is encapsulated within the behaviour specification given in a separate file. In both cases, behavioural specifications are expressed in expressive and well-established languages, that have their own parsers. Both for separation of concern, and for practical reasons, it would not have been reasonable to incorporate these descriptions within the ADL syntax. Other models in the litterature are specifying behaviours in different ways, attaching them e.g. to interfaces (as previous versions of Behavior Protocols) or to bindings (as in Wright). While it is unlikely that any formalism use all those possibilities together, we provide them in our proposal.

2.1 Proposed Addition to the ADL

Our common proposal is to add the following elements to the existing ADL formalism. These elements are sufficient to manage both SOFA and FIACRE formalisms, and seem to be open enough for other teams to integrate easily if needed.

Only minor changes to the ADL are needed, as we propose that every implementation-specific details be treated by a specialised parser as they are not related to the architecture. The elements and attributes needed to comply with the above requirements are summarised as:

- A **behavior** element, that contains behavioural specifications covering both the Sofa and the Oasis approaches, and is open to others in the future. The element can be nested inside the **component**, **definition**, **interface** or **binding** element. The **behavior** element will have the following attributes:
- A **language** attribute, specifying the behaviour specification language. Currently it can have values **fc2**, **lotos**, or **behavior-protocol**.
- A **file** attribute of type String, used to designate a file containing the behaviour specification. The **language** attribute will be used to select the adequate parser for this file.
- A **value** attribute of type String, when the behaviour specification is provided inline in the ADL file. Again the **language** attribute will be used to select the adequate parser for this string. Only one of the attributes **file** or **value** should be specified.

Examples:

```

<component name="Phone">
  <interface ...>
    <content class="Phone"/>
    <controller desc="primitive"/>
    <behavior language="lotos"
      file="Phone.lotos">
</component>

```

```

<component name="IFirewall">
  <interface name="IFirewall" role="server"/>
  <content class="FirewallImpl"/>
  <controller desc="primitive"/>
  <behavior language="behavior-protocol"
    value="?IFirewall.Enable* | ?IFirewall.Disable*">
</component>

```

Extensibility: Adding a new possible behaviour formalism would not require any modification of the ADL DTD: it consists in adding another keyword as a valid value for the `language` attribute, and providing the corresponding parser.

Additional information: In a previous implementation, some ADL extensions had been defined besides the plain behaviour specification. Work at Charles University [1] has shown that while for static and runtime checking, associating a component with a protocol was sufficient, the task of employing a model checker to check the compliance of the implementation of a primitive component with its specification (frame protocol) required additional information, which has been embedded in an additional element, `environment`, introduced into a locally developed extension of Fractal ADL. Similarly, the tools developed by the Oasis team use information coming from Java interfaces, or from annotations in the Java code.

As we want a generic extension of Fractal, we have decided to avoid extensions of the ADL dtd that would carry information specific to one approach or another. Instead, we group additional information together with the behaviour specification in a separate document, which can be, e.g., again an XML document, with syntax (DTD) determined by the specific Fractal extension — selected by the `language` attribute.

3 Collective communications for distributed components

In large distributed component systems, and in particular in Grid applications, the physical location of process units is decorelated from the functional structure of the system; a component can be formed from smaller pieces allocated on different sites, and the allocation of primitive components (basic computation units) on physical resources is often dynamic. The multiplicity of distributed components implies multiway communications, and notably parallel communications.

Multiway communications between components may be expressed thanks to collective interfaces; collective interfaces (multicast and gathercast) for Fractal are the subject of ongoing work in the OASIS team, and results are expected soon, as a formal proposal and a proof-of-concept implementation.

Multicast interfaces have been proposed to specify one-to-n behaviours on Fractal interfaces: an invocation reaching a multicast interface triggers parallel invocations on the components bound to this interface. These invocations between distributed components need specific policies to duplicate or scatter the computation parameters, and to gather the results.

The distribution of parameters from a multicast interface to connected interfaces may be specified in various ways; in the case of Java implementations of Fractal, one way is to annotate the Java interface specified as the signature of a multicast interface. Annotations, which are meta-informations associated to interfaces, methods or parameters can indicate precisely the distribution strategies for the invocation parameters, notably the possible duplication or scattering of these parameters. For example, the annotation on the following method `foo` indicates that the same parameters `listOfA` and `b` are sent to each connected interface (usually through duplication).

```
@MethodDispatch(mode = @ParamDispatch(mode =ParamDispatchMode.BROADCAST))
public void foo(List<A> listOfA, B b);
```

3.1 Proposed Addition to the ADL and Annotations

The implication on the ADL only concerns the cardinality of the interfaces: a new cardinality `multicast` may be specified to indicate a multicast behaviour. The details of the multicast behaviour are then inferred from the annotations in the signature of the interface.

Similarly to multicast interfaces for one-to-n interactions, n-to-one interactions may be expressed using gathercast interfaces. Let us just mention that the cardinality of interfaces should also be extended with the *gathercast* cardinality, which in the ADL will just appear as `cardinality='gathercast'` in the type of gathercast interfaces.

Finally, by allowing the specification of collective behaviours in ADLs, (with inferable behavioural details, for example from annotations), this proposal enables static behavioural analysis and verification of components with collective interactions.

4 Conclusion and Perspectives

We have presented a proposal for an extension to the Fractal Architecture Description Language (ADL), allowing the specification of component behaviour. This extension can be used for usual synchronous implementations of Fractal, but includes also specific constructs for handling asynchronous implementations (like ProActive). Our proposal unifies the views of the SOFA team, that develops methods and tools for the specification and verification of component systems based on regular trace languages, and of the OASIS team, that works on the specification of distributed components based on parameterized transition systems, and on the verification of their behaviour using bisimulation-based model-checkers. Moreover, we proposed another extension of the ADL in order to support multicast communications.

More generally, as we have carefully separated the specification languages themselves from the ADL definition, our approach can be very easily extended to other specification languages.

A natural extension of this work lies in the concept of *Parameterized Components*, that appears naturally in many applications, especially in Grid Computing where the actual grid resources may dynamically change between two executions or during execution. Typical examples are computations involving finite elements structures, implemented as distributed components running on a cluster, or “worker” processes in a peer architecture. Each element in an array of components would have the same behaviour apart from some knowledge of its position within the system.

The Fractal ADL has no provision for describing parameterized topologies of components. Every component must be individually declared, thus the main goal of parameterization is lost: does the designer mean these components are similar or is it just a coincidence? There is a need to properly define a higher abstraction that nicely handles multiplicity. When seen from the implementation point of view, the richer specification primitives can easily lead to communication optimisations. From the analysis and verification point of view, parameterized models generated from such high level constructs would give us more compact models, that can easily be instantiated to small finite configurations tailored for model-checking tools, or be used directly using more elaborated proof techniques able to handle parameters.

How to add in the Fractal ADL enough information to deal with parameterized components is still an open question. One possibility would be to have a simple *cardinality = multiple* attribute in the Component element, and to keep the parameter domain definitions inside annotations in the Java interfaces, as proposed for multicast communication. Another possibility would be to add some data definition at the ADL level, defining both the data domains and the dataflow between components.

References

1. Component reliability extensions for Fractal component model. http://kraken.cs.cas.cz/ft/public/public_index.phtml.
2. Dream communication framework. <http://dream.objectweb.org>.
3. JULIA framework (fractal implementation). <http://fractal.objectweb.org>.
4. T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Forte'04 conference*, Madrid, 2004. LNCS 3235, Springer Verlag.
5. T. Barros, L. Henrio, and E. Madelaine. Behavioural models for distributed components. In *FACS'05 Workshop*, Macao, 2005. LNCS , Springer Verlag.
6. E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh Int. Workshop on Component-Oriented Programming (WCOP02)*, at ECOOP 2002, Malaga, Spain, 2002.
7. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, aug 2002.
8. P. Parizek and F. Plasil. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30)*. IEEE Computer Press, april 2006. to appear.
9. F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), nov 2002.
10. ProActive. INRIA, 1999-2006. <http://www-sop.inria.fr/oasis/ProActive>.