Architecture Evolution through Dynamic Reconfiguration in jADL

A. Papapostolu, D. Birov

Key Words: *Software architecture; component-and-connector view; software evolution; dynamic reconfiguration; ADL; context-oriented software development.*

Abstract. *In software architecture the dynamic structure of a software system can be described in terms of components and connectors and expressed through the use of Architectural Description Languages (ADLs). We present jADL, a new ADL, designed for the creation and validation of dynamic and mobile architectures. It, also, aims to help towards the process of "unifying" the definition and implementation of an architecture in a way that the final result is consistent with the architecture in terms of both functional requirements and quality attributes. In this paper, we examine into details the definition and expression of jADL's architectural elements; components, connectors, ports, roles and behaviour describing statements – attach, detach, config, bind. The attributes of synchronicity and multiplicity concerning the connections between architectural elements are presented. A special case of connections is presented – the bind statement and the construction of composite architectural elements. Also, a case study of the client-server dynamic model in jADL is presented and the possibilities of jADL for dynamic reconfiguration are explored.*

Introduction

Software architecture [1,2,3] provides the various stakeholders with the possibility of "observing" the software system from a more abstract level, thus allowing them to reason about a system's both functional requirements and quality attributes. Due to the high complexity of software systems, software architecture considers the *static*, *dynamic* and *deployment* perspectives. The static perspective concerns the static part of a software system (classes, packages, modules, frameworks APIs, etc.). The dynamic perspective outlines the runtime behavior of the system. The deployment perspective describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment (the hardware environment that a system needs, the technical environment requirements for each element etc.). The perspectives have a number of views. The important view of the dynamic perspective is the component-and-connector (C&C) view where components and connectors are the constituent elements and their *interrelationships*, *behavior* and *constraints* are presented. *Components* are computational elements and data stores and they communicate with their environment exclusively through their declared *ports*. Components can communicate with each other only through connectors. *Connectors* represent the various forms of communication of components and their declared *roles* (respectively to a component's ports) are their exclusive points of interaction. A connection is established when a connector's role is attached to a component's port. If all components communicate with others only through connectors then the system has a property named *communication integrity* [4].

The connections between components and connectors compose the topology of the architecture of the system which is described and formally expressed through Architectural Description Languages (ADLs). ADLs are domain specific languages for description of the structure (topology) and the behavior of the software architecture. The topology can be formalized as a graph of components and connectors connected to each other by arcs. The behavior of components and connectors provides designers with information about their functionalities, the data flow, the way they communicate with each other etc.

In the last two decades, a lot of ADLs have been proposed like ArchJava [6], PADL [7,16], πADL [8], Wright [9], ACME [10] and others. Most of them are based on formal methods, which allow architects to reason about software architecture structure and behavior [7,8]. Formal methods allow to verify, validate and ensure syntactically and semantically correctness of the software architecture. In despite of the ADLs' evolution, the biggest challenge remains their practical usefulness in commercial software development companies. We are further exploring issues we discussed in [18] such as the dynamic reconfiguration [5] of a software intensive system. A significant issue is the "correspondence" of the "design" architecture with the "implementation" architecture of a software intensive system. It is common that the actual software implementation deviates from the original design and architecture – e.g. communication integrity is violated or not all of the quality attributes are taken into consideration.

We present a new ADL, called jADL [11], which has a Java-like syntax (for familiarity reasons) and is designed for the description and validation of dynamic and mobile software architectures. It further aims in providing a complete toolset to the architect for validating/verifying/etc. the architecture and, eventually, producing implementation code stubs (we chose Java [12] for our prototype) that are consistent with the defined architecture and embody its quality attributes. jADL describes the software architecture as a process; in order to actually "create" the architecture a jADL script needs to be executed.

In the following section, we further examine the use of *ports* and *roles* in jADL, as well as, their various features – their *interfaces* and *kinds* and issues that are connected with their *multiplicity* and *synchronicity*. We, also, present a new definition of the previously introduced *attach*, *de-*

tach and *bind* statement (used in the construction of composite architectural elements). In the third section, we illustrate jADL's possibilities for dynamic reconfiguration through two simple, but common, cases – an implementation of a dynamic client-server model where the server is replaced dynamically (e.g. in case of a shutdown) and the reconfiguration of the behavior of a component at runtime. In the last three sections, an early version of the compiler, the related work and conclusions are discussed.

Architectural Elements in jADL

In jADL, the basic building blocks and first-class architectural entities are *components* and *connectors* and they are defined (using BNF [13]) as presented in Appendix. All the definitions for jADL declarations, statements etc. could be found in the Appendix.

Component: declares a number of ports in order to communicate through connectors with other components. The ports are of two kinds: *requires* and *provides*. A requires port waits for input from a connector, but a provides port must be configured inside the component's body. Only this configuration is visible to the other elements; other internal methods can be accessed only by the component itself.

Throughout this paper we will present an architecture in jADL for the client-server architectural family (*figure 1*). We start here by defining the two components that participate – a *Client* and a *Server*. Below follows their declaration, as well as the declaration of their interfaces (the definition and use of interfaces in jADL is presented in detail after the definition of components and connectors).

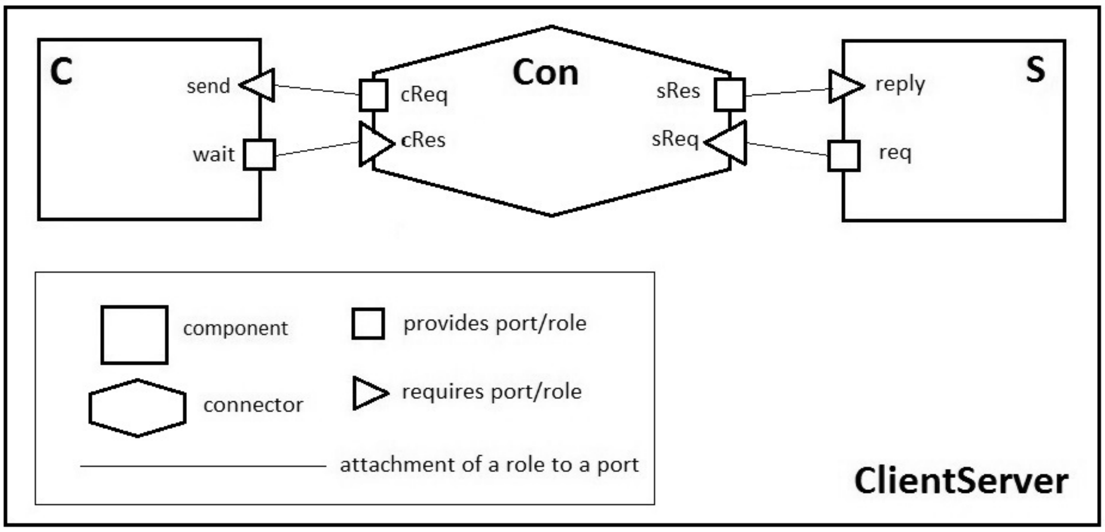

Figure 1. Client – Server Architecture

Interfaces and Client Declaration

```
//parametric type variable declaration. Architecture description
//is parameterized by type of client request data type.
type Type;

//client's interfaces
interface IRequest {
   void Request(Type data);
}
interface IReceive {
   void Received(Type data);
}

component Client {
   requires port IRequest send;
   provides port IReceive wait;

   config wait as {
      void Received(Type data){
         //process the result-response
         browser.display(data);
      }
   }
}
```

Code snippet 1. The declaration of the interfaces and the client in jADL

The *Client* component has two ports; one requires port (send) from where it sends its request to the server (through a connector) and one provides port (wait) from where the response from the server is delivered (again through the connector) and can be processed by the client. The client (the server and the connector too) is parameterized with *Type* which stands for various standard types (String, for example).

Ports are the only point of interaction for components (roles for connectors respectively). Both ports and roles in jADL are treated as first-class architectural citizens. They are used to ensure the control and data flow, which is established with the attachment of a role to a port. They are characterized by their *interfaces*, their *kind* (provides – output, requires – input) and the *multiplicity* and *synchronicity* of their connection. These are, also, the factors that define whether an attachment will be successful or not; the two interfaces must be compatible, their kinds must be opposing (a provided role with a required port or vice versa) and of the same synchronicity (both synchronous or both asynchronous).

Interfaces. They are used to define the shape of communication and behaviour of a port or role – the way this architectural element can be used by the rest of the elements. They constitute descriptions of protocols that define the communication between the architectural elements. A major advantage of interfaces is the ability to group different connection channels expressed by a *signature* grouped together. The port or role interface defines the communication shape and it should not be misunderstood as functional or method call because of their syntax similarity. For example ***void f(Integer** a, **Double** b, **String** c)* as a part of an interface represents a channel according to polyadic high order typed applied pi-calculus [14] where ***f*** is the name of the channel. If the interface is used for the declaration of a *requires* port then values tuple (*a, b, c*) is expected through channel ***f*** to be received. The type of tuple is ***(Integer*, *Double*, *String*)**. jADL uses two different type systems – one classical type system which consists of primitive types of values like Integer, Double, etc. plus type constructors like Array[T], Vector[T], Queue[T], etc. Second type system defines port and component types.

jADL views ports and roles as channel of communications. These channels have a data type and interfaces are flexible common way to describe complexity of communication. But channels can also be typed with primitive types as well. In this case the port and or role represents a channel of some kind – provides or requires – of sending or receiving the values. When a port and role are connected their "interfaces" need to be compatible. This is achieved by unification between the port's interface shape and role's interface shape. During the unification interfaces and types of both component and connectors are unified and their corresponding types inferred. The set of connections shape constitutes an interface and is used to express the behaviour of an architectural element in jADL. The standard control flow statements – if-then-else, switch-case, while and for loops – are allowed. The *config* statement is used for assigning a behaviour (interface) to a port/role, as seen on the implementation below.

Interfaces and Server Declaration

```
//server's interfaces
interface IResponse {
   void Response(Type data);
}
interface IProcess {
   void procRequest(Type data);
}

component Server {
   provides port IProcess req;
   requires port IResponse reply;

   attribute boolean down = false;

   config req as {
      void procRequest(Type data){
         //create response and reply
         Type resp = processReq(data);
         reply.Response(resp);
      }
   }
   Type processReq(Type data){
      //do sth - return result
      //...
      return res;
   }

   while(!down){
      select {
         process; }
      or {
         delay 300;
         down = true; }
      end;
   }

}
```

Code Snippet 2. The declaration of the interfaces and the server in jADL

The server has also two ports; one provides port to accept the request to be processed (req) and one requires to send its response (reply). It also has an attribute – boolean down – which indicates the state of the server. All these four ports implement different interfaces; the successful communication is achieved through the connector.

Connector [15]. declares a number of roles which are attached to ports so that the component communication is ensured. The same rules defined for ports, apply to a connector's roles. We continue with our client-server example by defining the connector that connects the two components defined. The interfaces used by the connector are the same that were presented above.

The connector has four roles (correspondingly to the four ports presented). The unification of client.send and conn.cReq (*figure 1*) ensures that the request will be accepted from the connector and it will be pushed forward to the server through the attachment of con.sReq and s.req. After the response is calculated it will reach the client in a similar way (using the attachments between s.reply - con.sRes and con.cRes - c.wait).

Connector Declaration

```
connector Conn {
   provides role IRequest cReq;
   requires role IReceive cRes;
   provides role IResponse sRes;
   requires role IProcess sReq;

   config cReq as {
      void aRequest(Type data){
         sReq.pRequest(data);
       }
   }

   config sRes as {
      void aResponse(Type data){
         cRes.Received(data);
      }
   }

}
```

Code Snippet 3. The declaration of the connector in jADL

Components and connectors participating in a communication can be part of the same process (or thread) as well as parts of different processes and threads and they can, also, be grouped together to produce a composite component or connector (an architectural element consisting of other elements appropriately connected together). In order for a communication to occur between a component and a connector a connection must be established between them. In jADL this is achieved by attaching (connecting) a role to a port using a simple statement. In the code below we can see the use of attach and how the architecture of *figure 1* is expressed in jADL.

A new instance for the connector and each compo-

Topology Declaration

```
//architecture definition
architecture ClientServer {

   instance client = new Client();
   instance server = new Server();
   instance conn   = new Conn();

   attach(conn.cRes, client.wait);
   attach(conn.cReq, client.send);

   attach(conn.sRes, server.reply);
   attach(conn.sReq, server.req);

   while(true)
      client.send.aRequest(myRequest);

}
```

Code Snippet 4. Element instantiation and attachment definition in jADL nent is created. All the ports are attached to the appropriate roles and the communication is ensured so that the client can make a request with a call from his port (send).

Kind. From the declaration of ports and roles, the keywords *provides* and *requires* are used to declare their kind. Every port or role must have a kind. Provides is used for the declaration of a port or role which submits data through a connection. The information processed in the implemented methods of a component, for example, is available to its port and will be provided to any successfully attached role to it. Requires is used for a port or role which expects data through connections. Upon the creation of an attachment, the kinds of the participants are compared and if they are not opposed the attachment is unsuccessful and the compiler reports an error.

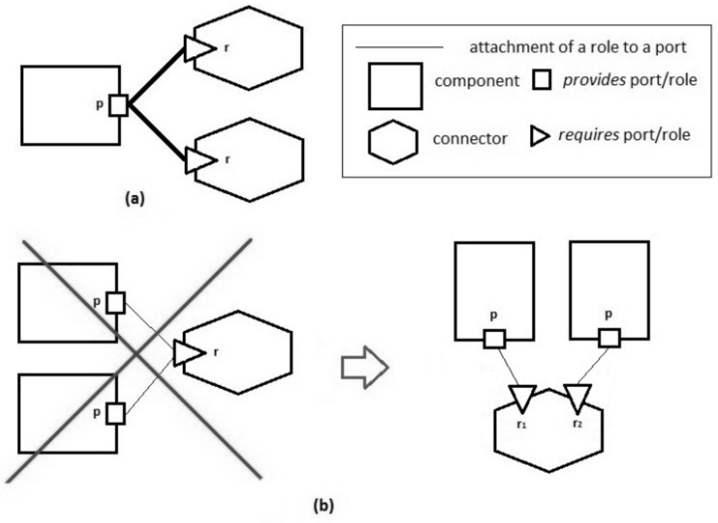

Figure 2. Variations of 1-N communications in jADL[1]

Multiplicity. The simplest type of a connection is when one role is attached to one port (1-1 communication). In addition to that, jADL supports and more complicated connections of the type of 1-N communication. *Figure 2* provides cases where attachments have more than two architectural elements involved. While the attachment in *2a* is successful, the one in *2b* is not and had to be transformed as shown in the figure. This is due to the fact that in jADL there is a constraint concerning the ports and roles of the requires kind. Only a declared as a provides port (or role) can be attached to multiple requires roles (or ports). When more than one provides ports or roles are attached to one requires role or port, then issues of non-determinism appear. jADL in order to avoid that prohibits these connections and each requires role or port should be attached to exactly one provides port or role.

Synchronicity. In a 1-N communication, additional problems than those mentioned above will appear when each of the architectural elements participating is part of a different thread. In *figure 3* is illustrated this case; the two connectors, each executed in a different thread, might attempt to gain access to the same resource of the component (a third thread), so concurrency issues will arise.

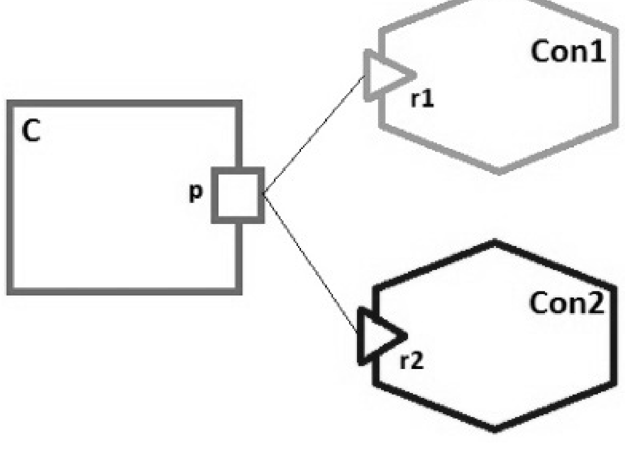

Figure 3. 1-N communication of three different threads

[1] *The legend of this figure applies to all figures presented.*

In jADL, when declaring a port or a role, as shown above, there is an optional keyword (*synchronized*) which defines the synchronicity of the communication; when used the communication is synchronous and when omitted the communication is asynchronous. For example, in *figure 3*, let's assume that in component C there is a queue (q1) defined in which other elements push their events and the two connectors Con1 and Con2 need the size of this queue in order to process their calculations. Then in component's C definition there will be:

```
provides port IQueue p;
…
config p as {
    int getSize() {
        return q1.size();
    }
}
```

and in connector's Con1 definition (and respectively to Con2):

```
requires role IQueue r1;
…
r1.getSize();
```

This code would be correct if we had only one thread of execution. But, since we have three different threads our shared resource – the queue – must be protected. So the code should be modified; the keyword *synchronized* must be added to both the port p and the roles r1, r2 declarations. The result will be that during the compilation of the script a synchronized block of code will be generated, concerning the shared queue, and the Java Virtual Machine, during runtime, will handle the concurrency making sure that the connectors will always receive the true size of the queue.

Bind Statement. In [11] we have presented a special case concerning the attachments in jADL; the *bind* statement. A statement describing a connection between an external port or role of a composite component or connector with an internal port or role of one of its internal architectural elements that constitute it. A simple example was presented which falls under the type shown in *figure 4a*. The definition of bind as presented there – "the arguments must both be of the same type (port, role) and kind (provides, requires)" – does not cover the cases shown in *figure 4b* and *4c* where a role must be binded to a port and neither does attach which requires opposing kinds.

In order to allow more flexibility we extend the definition to: *the arguments in the bind statement must be of the same kind*. In this way it is now possible to describe any variation of the types presented in *figure 4*.

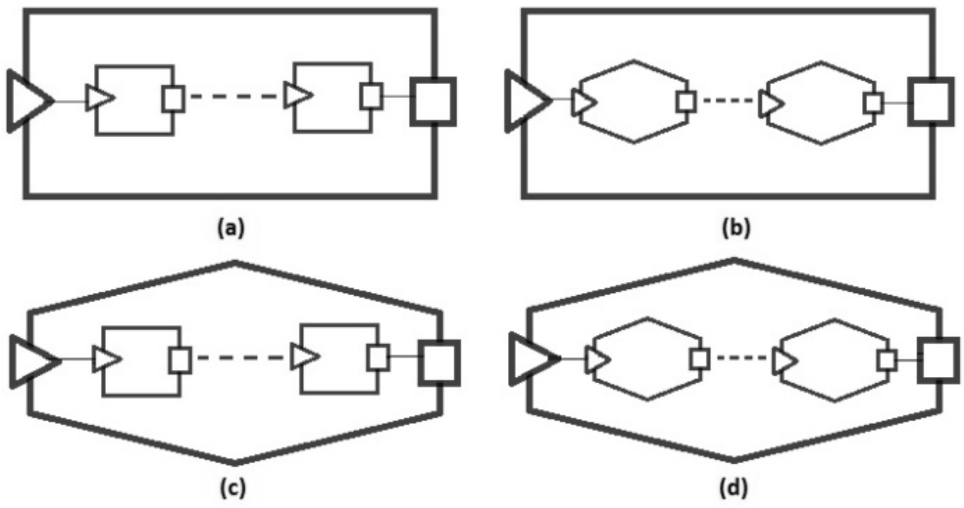

Figure 4. Combinations of a composite architectural element

Dynamic Reconfiguration

Dynamic reconfiguration is the evolution of a software system performed at runtime without disrupting the execution of the system. It is one of the most important aspects that an ADL needs to cover in order to be integrated in software development processes and develop practical use. jADL provides statements for dynamic reconfiguration. We present two simple cases of dynamic reconfiguration and how they are implemented in jADL. The use of the *detach statement* is demonstrated. Detach can be used both when defining an architecture and at runtime. It is used in order to destroy an attachment (connection) between a given role and port. Furthermore, after explaining the use of the config statement for assigning a behavior to a port (or role) when defining an architectural element, we present its use at runtime when it is used for dynamically assigning a behavior.

Case 1 – switching between two components. Here we will present a simple implementation of a dynamic client-server architecture in jADL. The architecture consists of three elements; a client, a connector and a server we presented before.

The architecture is dynamic because (as we can see in the implementation below in the component ClientServer) the infinite loop checks constantly the attribute down. This attribute informs us about the state of the server (true- not working, false – working). If the value is false the server keeps processing the requests, but if the server does not respond and remains inactive for 300ms the attribute is set to true and the server is stopped. When the server stops working, it is detached from the connector, a new instance is created and then attached to the connector, hence making this architecture dynamic.

This is the implementation of the client server architecture in jADL.

Dynamic Architecture Declaration

```
//architecture definition
architecture ClientServer {

   instance client = new Client();
   instance server = new Server();
   instance conn   = new Conn();

   attach(con.cRes, c.wait);
   attach(con.cReq, c.send);
   attach(con.sRes, s.reply);
   attach(con.sReq, s.req);

   while(true){
     c.send.aRequest(myRequest);
     select {
       process;
     }
     or {
       delay 300;

       dettach(con.sRes, s.reply);
       dettach(con.sReq, s.req);

       instance s = new Server();

       attach(con.sRes, s.reply);
       attach(con.sReq, s.req);
     }
     end;
   }
}
```

Code Snippet 5. Definition of a dynamically reconfigurable architecture in jADL

Case 2 – creation of a new component and architecture reconfiguration. In this case, let's consider the following scenario: a component or a connector needs to be replaced with a new one with enhanced functionality. We assume that the changes forcing us to perform this replacement require backward compatibility of the architectural element replaced, so what we need is a new element, similar to the old one, with a "reconfigured" port or role. Such changes might be, for example, the replacement of internet communication protocol IPv4 with IPv6 or a change in the format of the output data provided by the element.

As it is shown above, *config* is used in jADL when statically defining components and connectors, and more specifically when describing their behaviour. But, config can also be used at runtime to assign new behaviour to concrete port(s) or role(s) of an element, which will result in the creation of a new architectural element. This applies only to ports and roles of the *provides* kind.

First, the reference of the architectural element (<*portRoleRef*>) that needs reconfiguration must be obtained. After the implementation of the new behaviour inside the block surrounded by *{ }*, a new architectural element is created and its reference is obtained and can be used from that point on. For example, assuming that we have a component C with a provides port p that must be replaced, connected to a connector Conn with a requires role r, then that is how the script performing the replacement could be:

Sample of a Dynamic Architecture

```
architecture SampleDynArch {

   instance comp1 = new C();
   instance conn  = new Con();

   instance comp2 = comp1 with {
                        config comp1.p as {
                          //new behaviour
                        }
                   }

   attach(con.r, compA.p);
   //…
   detach(con.r, comp1.p);
   attach(con.r, comp2.p);

}
```

Code Snippet 6. Example of dynamic reconfiguration in jADL

From this simple script above, a new component will be created (*comp2*) and at this stage we'll have a reference to it (*comp2.p*). Now, the replacement is easy and it is achieved with the use of detach as presented in case 1.

Compiler

A compiler is created for the definition/validation/etc. of jADL scripts. The parser used for the lexical, syntax and semantic analysis of the scripts was built at the beginning using the ANTLR (ANother Tool for Language Recognition) [17] tool. During the time of the review process, we changed the tool used for the creation of the compiler to Xtext [19] (which, also, uses ANTLR internally for the generation of the parser). Xtext is an Eclipse extension, a framework used for defining Domain Specific and General Purpose Languages. The decision for changing the tool was based on the enhanced options for configurability provided by the Xtext framework and the toolset available (parser, compiler, editor, etc.). The compiler is currently under construction.

Related Works

π-ADL[8] provides a way for the description of dynamic and mobile software architectures. It is designed as a domain-specific extension of the higher-order typed ð-calculus [14] and it aims in providing both structural and behavioural architecture-centric constructs for high architecture expressiveness. Connectors in π-ADL are not defined separately as distinct architectural elements, but instead of that they are simulated through the use of special-purposed components. PADL [7,16] is a process algebraic ADL with high expressiveness. In PADL primarily are described architectural types – an intermediate abstraction between a single system and an architectural style [1]. An architectural type is defined by its behaviour (the architectural elements and their topology) and its interactions (their synchronicity and their multiplicity).

The main difference with those languages is that in jADL we treat as first-class elements both components and connectors, and their ports and roles as well, unlike the predefined connectors of PADL or the use of special-purposed components of π-ADL. Treating them as first-class entities enables the support of a number of very important operations, that otherwise would be unavailable, such as being passed as an argument, returned from a function or assigned to a variable. jADL is created with the goal to provide to architects/designers/developers the means to easily being able to perform dynamic reconfigurations of the system and evolving it.

Conclusion

In this paper, we extensively described the definition of jADL's architectural elements – components, connectors, ports and roles and, also, various statements and approaches concerning the dynamic reconfiguration support in jADL. Additionally, we presented the various factors characterizing their attachments and the attributes of the connections created in terms of multiplicity and synchronicity and proved that jADL is capable of expressing a wide range of communication types (1-N, 1-1, synchronous etc.). The bind statement was, also, presented which is used for the creation of composite architectural elements and we extended its definition to ease architects in the creation of composite components or connectors. Finally, two simple cases of dynamic reconfiguration were presented and how they are implemented using jADL. Both cases concerned reconfiguration on the instance level; one unforeseen and one foreseen. jADL provides the language constructs for these reconfigurations to occur easily and safely, but there is still a lot to be done when it comes to dynamic reconfiguration. Our next goal is to extend the support of jADL to reconfigurations on the type level and the reassessment of the system after these changes are applied. We, also, aim at creating a model for the integration of the better embodiment of the quality attributes (non-functional requirements) and constraints into the generated implementation code stubs.

Acknowledgement

This work was supported by NSF DO 02-102/2009.

Appendix

Here we present the syntax for the statements and declarations in jADL using BNF.

```
<component_declaration> ::= "component" <component_ID> "{" {
                                    [<port_declaration>]*
                                    [<config_statement>]*
                                    [<internal_method>]*
                               } "}"

<connector_declaration> ::= "connector" <connector_ID> "{" {
                                    [<role_declaration>]*
                                    [<config_statement>]*
                                    [<internal_method>]*
                               } "}"

<port_declaration> ::= ("provides" | "requires") ["synchronized"]
                               "port" <interface> <id> ";"

<role_declaration> ::= ("provides" | "requires") ["synchronized"]
                               "role" <interface> <id> ";"

<bind_statement> ::= "bind" "("<roleOrPortId>,< roleOrPortId>")" ";"

<attach_statement> ::= "attach" "(" <role_id>, <port_id> ")" ";"

<detach_statement> ::= "detach" "(" <roleID>, <portID> ")" ";"

<newElementReconfiguration> ::=
           <elementType> <ID> = <elementRef> "with" "{"
                "config" <portRoleRef> ["include" <interface1>, ...] "{"
                             //new behaviour
                 "}"    "}"

<select_statement> ::= "select"
                         ["when" <booleanGuard> "=>" ] "{"
                              statement(s)        "}"
                       [ "or"
                          ["when" <booleanGuard> "=>" ] "{"
                              statement(s)        "}" ]*
                       "end;"
```

References

1. Shaw, M. and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

2. Taylor, R. N. N., N. Medvidovic E. and Dashofy. Software Architecture: Foundations, Theory, and Practice. United Kingdom, Wiley, John & Sons, 2009.

3. Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford. Documenting Software Architecture Views and Beyond. 2nd Edition, Addison-Wesley, 2011.

4. Moriconi, M., X. Qian and R. Riemenschneider. Correct Architecture Refinement. – *IEEE Trans, Software Engineering,* 21, 1995, 4.

5. Buisson, J., T. V. Batista, L. Minora & F. Oquendo. Issues of Architectural Description Languages for Handling Dynamic Reconfiguration. 2012.

6. Aldrich, J., C. Chambers and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. ICSE 2003, 187-197, 2003.

7. Aldini, A., M. Bernardo and F. Corradini. A Process Algebraic Approach to Software Architecture Design. Springer, 2009.

8. Oquendo, F. Dynamic Software Architectures: Formally Modeling Structure and Behaviour with π-ADL. Proceedings of the Third International Conference on Software Engineering Advances, IEEE Computer Society, Malta, 2008.

9. Allen, R., R. Douence and D. Garlan. Specifying Dynamism in Software Architectures. Proceedings of the Workshop on Foundations of Component-Based Systems, 1997, 11–22.

10. Batista, T. V., A. T. A. Gomes, G. Coulson, C. Chavez and A. F. Garcia. On the Interplay of Aspects and Dynamic Reconfiguration in a Specification-to-Deployment Environment. Proceedings of the 2nd European Conference on Software Architecture, Berlin, Heidelberg, 2008, 314–317.

11. Papapostolu, A. and D. Birov. jADL: Another ADL for Automated Code Generation. Science and Business for a Smart Future, Varna, Bulgaria, 2016.

12. https://www.oracle.com/java/index.html.

13. http://matt.might.net/articles/grammars-bnf-ebnf/.

14. Milner, R. Communicating and Mobile Systems: The Pi Calculus. 1st Edition. Cambridge University Press, 1999.

15. Amirat, A. and M. Oussalah. First-Class Connectors to Support Systematic Construction of Hierarchical Software Architecture. – *JOT*, 8, 2009, 7.

16. Bonta, E. Automatic Code Generation: From Process Algebraic Architectural Descriptions to Multithreaded Java Programs. Universita di Bologna, Padova, 2008.

17. http://www.antlr.org/.

18. Papapostolu, A. and D. Birov. Dynamic Reconfiguration Statements and Architectural Elements in jADL. Proceedings of the International Conference Automatics and Informatics'16, Sofia, Bulgaria, 4-5 October 2016, 153-157.

19. https://eclipse.org/Xtext/.

Manuscript received on 02.11.2016

Anastasios Papapostolu *is a PhD candidate in Sofia University "St. Kliment Ohridski" in Computer Science – Software Architectures at the Faculty of Mathematics and Informatics, Department of Computing Systems. He received his M. Eng. Diploma in Electronic and Computer Engineering from the Technical University of Crete, Chania, Greece in 2014. His professional and scientific research interests include self-adaptable and dynamic architectures, architecture description languages, compiler construction and language engineering.*

Contacts:
Sofia University "St. Kliment Ohridski"
Sofia, Bulgaria
e-mail: papapostol@fmi.uni-sofia.bg

Dimitar Birov, *PhD is an associate professor at the Faculty of Mathematics and Informatics of Sofia University "St. Kliment Ohridsky". He has professional experience as research fellow, lecturer, and project manager at Sofia University, University College of Dublin, Ireland, University of Orleans, France, Microsoft Corporation, Redmond, USA, Carnegie Mellon University, Pittsburgh, USA. He has industrial experience like software developer, software architect, consultant and CEO. He is patent inventor. His primary research interests are in software architectures, and software language engineering – formal and practical architecture description and analysis languages, software engineering and design, programming languages, and type systems.*

Contacts:
Sofia University "St. Kliment Ohridski"
Sofia, Bulgaria
e-mail: birov@fmi.uni-sofia.bg