МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2018
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2018
Proceedings of the Forty-seventh Spring Conference
of the Union of Bulgarian Mathematicians
Borovets, April 2–6, 2018

ARCHITECTURAL SELF-ADAPTATION AND DYNAMIC RECONFIGURATION IN jADL*

Tasos Papapostolu, Dimitar Birov

During the last decade there is a rapid growth of new internet technologies like Internet of Things (IoT), mobile services and cloud and service computing. Software architecture is the modelling tool for describing such widely used internet applications focused primarily on their quality attributes like security, self-adaptability, self-∗, etc. In this paper, we outline a new Architecture Description Language (ADL) jADL, through a case study of a self-adapting load balancer architecture. jADL is a modern ADL which, up to now, supports the creation and validation of dynamic and mobile architectures. Numerous studies have highlighted the absence of industrial usage of ADLs because of two reasons: language constructs represent too formal theoretical concepts which are incomprehensive to stakeholders and the lack of tools transforming high level architectural models to software artifacts. The jADL architecture description language is based on a Milner's version of an asynchronous process pi-calculus, called *applied pi-calculus*, for studying concurrency and process interaction. It provides intuitive semantics behind of the architectural elements and communication, which makes architectural description self-explanatory, supports dynamical architectural reconfiguration and allows for self-adaptation which is discussed in this paper.

1. Introduction. Software architecture [6] is "*the set of structures needed to reason about a software system, which comprise software elements, relationships among them and properties of both*". It is used to define the behaviour of a software system during runtime using abstractions and notations in order to facilitate the communication between the stakeholders. In pursuance of the best and most detailed definition of a software system and due to their high complexity, there are three perspectives defined in software architecture: the *static*, the *dynamic* and the *allocation* perspective, each one with a number of *views*. In this paper we are concerned with the dynamic perspective, where the important views are the *Component-and-Connector* views (*C&C*). The C&C views are comprised of two sets of architectural elements: *components* and *connectors*. The components are the computational and data store elements and the connectors represent the communication between them. The components have declared *ports* whose interfaces are used when communicating with other elements. The jADL architecture description language defines shaped by interfaces ports and roles. Port interfaces, differ from the traditional interfaces

*2010 Mathematics Subject Classification: 68N01, 68M14, 68Q85.
Key words: software architecture, ADL, dynamic reconfiguration, self-∗, load balancer.

in programming, since here they define the shape of the communication. The connectors define *roles* with which they participate in the communications. Same as ports, they have interfaces which shape the communication. The communication between the elements occurs when a role is *attached* to a port.

The architectural elements, their interconnections and the constraints concerning them compose the *topology* of the software architecture. The communication, the data flow and the component-connector interactions describe the *behaviour* of the software architecture according to the topology. If the topology or the behaviour changes during run-time, the architecture is referred to as *dynamic* or *mobile*. When these changes are performed without a human assistance the architecture is called *autonomic*, or *self-adaptable*, or *self-*$*$. Self-$*$ systems can be defined as systems capable of managing themselves, where the $*$ character stands for a number of properties: self-configuration, self-adaptation, self-diagnose, self-repair, etc.

The architectural dynamism is a natural feature of current social media applications, IoT, mobile computing, cloud and other advanced distributed applications. The formal expression of all this variety of computing structures in software architecture is through the use of ADLs. The ADLs can express the architecture from a higher level focusing both on functional requirements and quality attributes. They can provide the formal basis for the tools generating software artifacts such as implementation code stubs, lower level models, etc.

This paper illustrates the capabilities of jADL, a new ADL, which describes quality attributes of a software system and software architecture properties. The lack of support and tools from the majority of the available ADLs is the main reason for introducing jADL, which aims to provide these capabilities integrated in the language and therefore promoting the use of ADLs in industrial software implementations processes. It is designed to describe static, dynamic and mobile architectures, focusing on providing the flexibility and expressiveness required in order to express the dynamic reconfigurations (foreseen and unforeseen) of software intensive systems. Its architectural elements, syntax and various statements are presented and explained in more details in [14]. The syntax of jADL is influenced by good practices such as ACME [9], π-ADL [5] and PADL [1].

Based on the case study of defining a Load Balancer Architectural Pattern (LBAP) [7] we exhibit the main features of jADL and aim to demonstrate its expressiveness and formal support of software architectural properties and its capabilities to define dynamic and self-adaptable architectures. The rapid expansion of distributed applications (mainly micro-service architectures impose handling of heavy loads of component communication), which now provide any type of service (purchases, payments etc.) led to techniques for handling the traffic. An example is load balancing – a number of low-cost servers and/or micro-service containers is used to distribute the incoming requests amongst them.

2. Load balancer architectural pattern. In this section we present how a Load Balancer Architectural Pattern (LBAP) [7] can be defined using jADL. It is comprised of a client, a load balancer server and a couple of dedicated servers. This pattern is applicable for server-less technologies like micro-services after simple modifications. In the example below we use for simplicity 3 servers, but in section 4 we extend it to N. We outline the main software architect activity for modifying and extending the LBAP, in order to trade off the stakeholders' requirements for performance and self-adaptability, through scalability of servers.

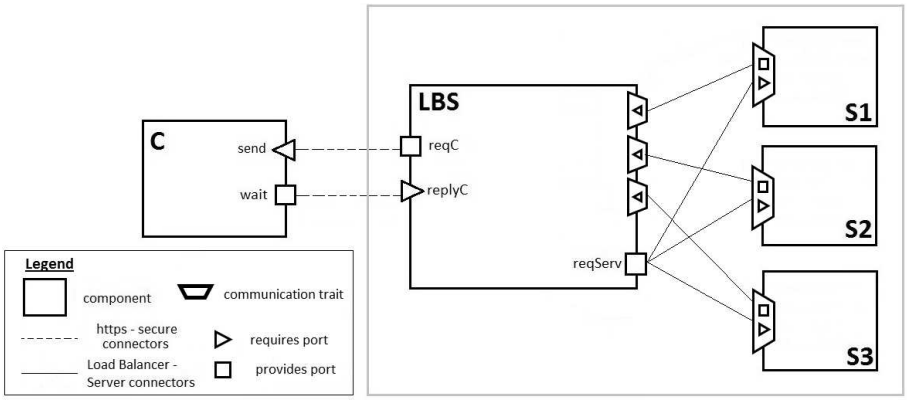

Fig. 1. Architecture of a simple load balancing system

The jADL is a scripting language, so in order for the architecture to be created an initialization script needs to be executed after the definition of the elements (code snippet 5). jADL assumes a usual situation where the architectural elements are executed in different processes and multiple threads of execution.

The client (C) shown in Fig. 1 represents one component, abstracting the client's interface and behavior (clients could be any mobile devices, traditional desktop browsers, smart phones, etc.). In jADL all components are connected to each other through connectors. The architectural schema (Fig. 1) defines two kinds of connectors; a secure connector for the client – load balancer communication and a simple one for the load balancer – server communication.

```
1.   type Type;

2.   interface IRequest {
3.     service void aRequest (Type data);
4.   }
5.   interface IReceive {
6.     service void Received (Type data);
7.   }

8.   component Client {
9.     requires port IRequest send;
10.    provides port IReceive wait;
11.    config wait as {
12.      service void Received (Type data) {
13.        //process the response
14.        display.media(data);
15.      } } }
```

Code snippet 1. Client description in jADL

The client component communicates with the Load Balancing System as if it "sees" a single server. It has 2 interfaces of communication which are represented with two ports. Through its *requires* port *send*, it sends a request and awaits for the response at its *provides* port *wait*. When a *provides* port (or role) is defined, jADL specifies configurations that describe its behaviour. During the compilation of the script, this requirement is checked and if it is not fulfilled the compiler produces an error. This static inspection of the script code prevents modeling errors of an application which could occur during run-time.

The behavior of ports and roles is defined through the use of the *config* statement, inside the brackets ***{ }***. This definition consists of *services*. They are the same like the ones defined in the interfaces of the ports/roles, but they "contain" the behavior, which

```
1.   interface IResponse {
2.     service void aResponse (Type data);
3.   }
4.   interface IProcess {
5.     service void procRequest (Type data);
6.     service int loading ();
7.   }
8.   trait ServCommTrait {
9.     provides port IResponse req;
10.    requires port IProcess reply;
11.  }
12.  component Server {
13.    attribute int curLoad = 0;
14.    attribute int maxNum = 1000;
15.    trait CTraitS aggregate ServCommTrait {
16.      config req as {
17.        service void procRequest (Type data) {
18.        curLoad=curLoad+1;
19.        Type resp;
20.        reply.aResponse(resp);
21.        curLoad=curLoad-1;
22.      }
23.    CTraitS comServ = new CTraitS();
24.    while (true) {
25.     select
26.       when (curLoad < maxNum) => {
27.         process; }
28.      or
29.       when (curLoad == maxNum) => {
30.       delay_until curLoad == (maxNum–100); }
31.      end; } }
```

Code snippet 2. Server description in jADL

is defined under the form of statements. The config statement can be used at runtime as well, for dynamically assigning a behavior. This means that we can reconfigure a port behavior and this is one of the mechanisms of jADL to support reconfigurability of the architectural elements during runtime.

Furthermore, the first line (code snippet 1) defines that our architecture is parameterized by the data type of the data exchanged during the communication of the participating elements. *Type* in this example could be instantiated with various standard data types (string etc.).

The architecture presented here is a distributed software architectural pattern. Components represent variations of server(s) abstracting their interface and behavior. The server component has two ports. The first port (*req*) accepts the clients' request and the other sends the response. These two ports are created at runtime with the instantiation of a new *CTraitS* (code snippet 2) *communication trait*. Each server has two attributes; *curLoad* – the current server's load, and *maxNum* – the maximum number of requests allowed. These two attributes are used to define the server's behavior under the control of the *select* statement. This statement is used to manipulate the behavior of an architectural element. In the server component, we use its complex form (the *when condition* is optional as shown in code snippet 4). If *curLoad* reaches the limit (*maxNum*), the *delay* statement is executed and the server stops accepting new requests. While the first condition is true, the *process* keyword is used to define that the server is working in a normal state. The *delay* statement is used to block further execution of the server (using a condition – code snippet 2, or by explicit declaration – code snippet 4).

The two connectors are used for data passing between the components. Ports and roles are unified in order to ensure the correct data flow between the client and the load balancing system creating a communication channel. The significant difference between them is the attributes in *Conn1*; they are defined because the public communication between a client and a server should be encrypted and secure. The other connector (*Conn2*) used for the communication between the servers is only for internal communication.

Communication trait is a new structured architectural element in jADL [14] that can group together ports and roles. Each trait can hold only ports or only roles of

```
1.  connector Conn1 {
2.    provides role IRequest cReq;
3.    requires role IReceive cRes;
4.    provides role IResponse sRes;
5.    requires role IProcess sReq;

6.    attribute string conntype = "secure";
7.    attribute string prot = "SSL";

8.    config cReq as {
9.      service void aRequest (Type data) {
10.       sReq.procRequest(data);
11.     }
12.   }
13.   config sRes as {
14.     service void aResponse (Type data){
15.       cRes.Received(data);
16.     } } }
```

```
17. connector Conn2 {
18.   provides role IRequest cReq;
19.   requires role IReceive cRes;
20.   provides role IResponse sRes;
21.   requires role IProcess sReq;

22.   config cReq as {
23.     service void aRequest (Type data) {
24.       sReq.procRequest(data);
25.     }
26.   }

27.   config sRes as {
28.     service void aResponse (Type data) {
29.       cRes.Received(data);
30.     }
31.   } }
```

Code snippet 3. Connectors description in jADL

both kinds. Traits are included in a component or a connector using the keywords *trait* and *aggregate*. After that they can be instantiated and used by the element. The compiler will check if there is a behavior assigned for each of the declared *provides* ports (if any). By being encapsulated in a separate structure the ports/roles can be dynamically instantiated/configured/etc. at runtime, an innovation proposed by jADL. This complex structure enhances significantly the flexibility and expressiveness of jADL, especially when it comes to describing dynamic and self-adapting architectures like the one presented in section 3.

The component lbServer is responsible for choosing the appropriate server to forward the current request. Once it is received, the *lbServer* has the following options: to forward it to one of the servers or to wait (if none of the servers is responding). In order to select the least loaded server, the load of each one of them is requested and they are compared.

```
1.  trait LBCommTrait {
2.    requires port IResponse rServ;
3.  }
4.  component lbServer {
5.    requires port IResponse replyC;
6.    provides port IProcess reqC;
7.    provides port IProcess reqServ;
8.    trait CTraitLB aggregate LBCommTrait { }
9.    config reqServ as {
10.     service void procRequest (Type data) {
11.       replyC.aResponse(data);
12.     }
13.   }
14.   CTraitLB comLB1 = new CTraitLB();
15.   CTraitLB comLB2 = new CTraitLB();
16.   CTraitLB comLB3 = new CTraitLB();
17.
18.   config reqC as {
19.     service void procRequest (Type data) {
```

```
20.       attributes {
21.         int m1 = comLB1.rServ.loading();
22.         int m2 = comLB2.rServ.loading();
23.         int m3 = comLB3.rServ.loading();
24.       }
25.       select
26.         when (m1≤m2 && m1≤m3) => {
27.           comLB1.rServ.procRequest(data); }
28.       or
29.         when (m2≤m1 && m2≤m3) => {
30.           comLB2.rServ.procRequest(data); }
31.       or
32.         when (m3≤m2 && m3≤m1) => {
33.           comLB3.rServ.procRequest(data); }
34.       or {
35.         delay 10; }
36.       end;
37.     }
38.   } }
```

Code snippet 4. Load Balancer Server description in jADL

```
1.  architecture LB {
2.    //elements declarations
3.    instance client = new Client();
4.    instance lbserv = new lbServer();

5.    instance s1 = new Server();
6.    instance s2 = new Server();
7.    instance s3 = new Server();

8.    instance secureC = new Conn1();
9.    instance simpleC1 = new Conn2();
10.   instance simpleC2 = new Conn2();
11.   instance simpleC3 = new Conn2();

12.   //client and load balancer attachments
13.   attach(secureC.cReq, client.send);
14.   attach(secureC.cRes, client.wait);
15.   attach(secureC.sReq, lbserv.reqC);
16.   attach(secureC.sRes, lbserv.replyC);
17.   //load balancer and servers attachments
18.   attach(simpleC1.cRes, lbserv.reqServ);
19.   attach(simpleC1.cReq, lbserv.comLB1.rServ);
20.   attach(simpleC1.sRes, s1.comServ.reply);
21.   attach(simpleC1.sReq, s1.comServ.req);
22.   attach(simpleC2.cRes, lbserv.reqServ);
23.   attach(simpleC2.cReq, lbserv.comLB2.rServ);
24.   attach(simpleC2.sRes, s2.comServ.reply);
25.   attach(simpleC2.sReq, s2.comServ.req);
26.   attach(simpleC3.cRes, lbserv.reqServ);
27.   attach(simpleC3.cReq, lbserv.comLB3.rServ);
28.   attach(simpleC3.sRes, s3.comServ.reply);
29.   attach(simpleC3.sReq, s3.comServ.req);

30.   //client that makes periodic requests
31.   while(true) {
32.     client.send.aRequest(myRequest);
33.     delay 5;
34.   } }
```

Code snippet 5. Instantiation and creation of the architecture in jADL

A new port is created, using the communication trait *CTratitLB*, for each of the three servers. The requests are forwarded through these ports to the appropriate server. If all of the servers are down and no response is received the load balancer waits before attempting to forward the request again. Eventually, when a server sends its response, it is forwarded back to the client through its *replyC* port.

In order for the architecture to be created code snippet 5 needs to be executed. First we instantiate our components and connectors using the *new* keyword and next we define the attachments between the various elements using their ports and roles. A communication channel is established between a component and a connector when a role is attached to a port. The *attach* statement in jADL expresses this communication and represents the unification of a role with a port.

3. Self-adaptive architecture of a load balancing system. We extend the architecture previously described by allowing for dynamism and self-adaptability when it comes to the number of servers available depending on the current load from the dynamically changing number of clients. The Load Balancer server can add or remove servers based on the total load during runtime, thus allowing the dynamic reconfiguration of the system and making it self-adaptable. The new architecture can be seen in Fig. 2 below.

The first difference here is that we define an additional trait for the load balancer which holds the port for sending the response to a client. Additionally, three data structures (*hashmap*) are defined for managing the servers. The first one (*requests*) holds the requests from the clients and their assigned trait. The hashmap *servers* holds the traits created for each server and its status (*active* – processing and accepting requests, *inactive* – does not accept requests). The last one (*conns*) holds the traits of each server and the connector used for their communication.

When a new request arrives at the *reqC* port a new trait is created and stored in the *requests* hashmap. Then, the average load of the servers is calculated (*findAvgLoad* – code snippet 6). The keyword *abstract* defines that an algorithm should be implemented there by developers, splitting this way the architecting activities from the implementation/de-

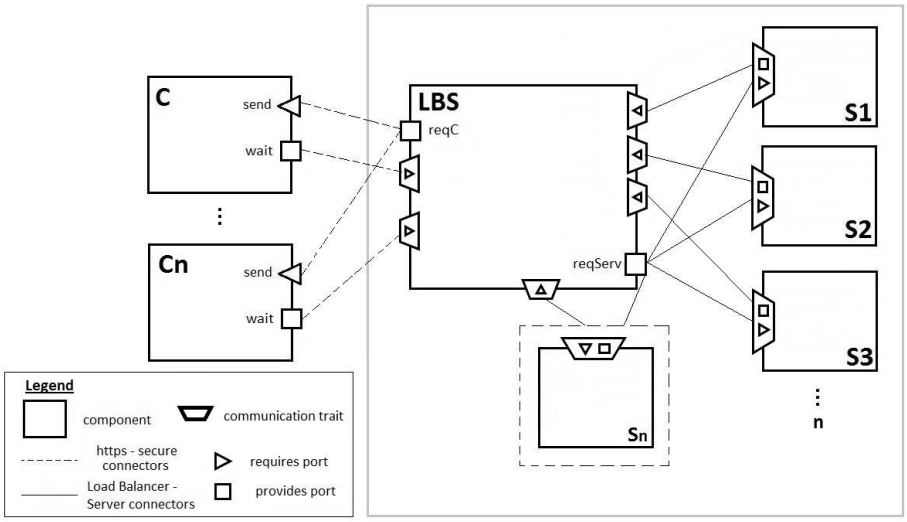

Fig. 2. Architecture of a self-adapting load balancing system

velopment part. If the average load is below 20% of the allowed one it sends the request to the least loaded server and changes its status to *inactive* so that no new requests will be sent to this server. If the average load is above 80% a new server is instantiated and attached to the load balancer. Its status is changed to *active* and the request is forwarded to it. Finally, if the average load is between 20% and 80% the request is just forwarded to the least loaded server.

Inside the infinite loop (code snippet 6, lines 55-62) the load balancer continuously checks the load of the servers with status *inactive*. When any of these servers finishes processing its last request and there are more than 1 available servers, the server is detached from the load balancer. This is defined in jADL with the use of the *detach* statement, which is used for destroying the communication channel (attachment) between a role and a port. It is the "opposite" of the attach statement and it accepts two arguments – the port and the role that will be detached (or two traits). These statements, *attach* and *detach*, can be used both when defining the architecture and at runtime (dynamic reconfiguration of the architecture).

4. Related Work. The ACME language [8, 9] started as a multi-style ADL framework providing the opportunity of using it as a common interchange platform for multiple ADLs. It does not provide a specific model to describe system behavior, functional properties or any other aspect of the system, so it allows the annotation with properties that represent this descriptive information. It provides explicit configurations and hence facilitates understandability and readability. It supports architectural styles, and provides a template mechanism for their implementation [11]. The need for dynamic reconfiguration grew over the years and since it was not "integrated" in ACME the help of additional tools is needed (e.g. Plastik [2]). Despite the various extensions ACME's support is still limited when it comes to dynamic reconfiguration. Also, for dynamically reconfiguring the behavior general formal languages must be used, while jADL offers these opportunities integrated. The ACME language supports user defined constraints on its elements (Armani

```
1.  trait LBCommTraitC {
2.    requires port IResponse rClient;
3.  }
4.  trait LBCommTraitS {
5.    requires port IResponse rServ;
6.  }

7.  component DynamicLB {
8.    provides port IProcess reqC;
9.    provides port IProcess reqServ;

10.   hashmap<Type, CTraitLBC> requests =
      new hashmap<Type, CTraitLBC>();
11.   hashmap<CTraitLBS, string> servers =
      new hashmap<CTraitLBS, string>();
12.   hashmap<CTraitLBS, Conn2> conns =
      new hashmap<CTraitLBS, string>();

13.   config reqServ as {
14.     service void procRequest (Type data) {
15.       requests.get(data).
      rClient.aResponse(data);
16.     }
17.   }
18. trait CTraitLBS aggregate LBCommTraitS
      { }

19.   CTraitLBS comLBS1 = new CTraitLBS();
20.   CTraitLBS comLBS2 = new CTraitLBS();
21.   CTraitLBS comLBS3 = new CTraitLBS();
22.   servers.put(comLBS1, "active");
23.   servers.put(comLBS1, "active");
24.   servers.put(comLBS1, "active");
25.   config reqC as {
26.   service void procRequest (Type data) {
27.    float avg = abstract finAvgLoad(servers);
28.   trait CTraitLBC aggregate LBCommTraitC { }
29.   CTraitLBC comLBC = new CTraitLBC();
30.   requests.put(data, comLBC);
31.   select
32.    when (avg<20) => {
33.      x = abstract firstActiveServer(servers);
34.      x.rServ.procRequest(data);
35.      if (count(servers.getKey("active"))>1)
36.        servers.put(x, "inactive");  }
37.   or
38.    when (avg>80) => {
39.      trait CTraitLBS aggregate LBCommTraitS { }
40.      CTraitLBS comLBS = new CTraitLBS();
41.      instance sN = new Server();
42.      instance conn = new Conn2();
43.      servers.put(comLBS, "active");
44.      conns.put(comLBS, conn);
45.      attach(conn.cRes, reqServ);
46.      attach(conn.cReq, comLBS.rServ);
47.      attach(conn.sRes, sN.comServ.reply);
48.      attach(conn.sReq, sN.comServ.req);
49.      comLBS.rServ.procRequest(data);  }
50.     or {
51.      proccess; }
52.     end;
53.   }
54.  }
55.  while (true) {
56.   for (servers srvK : srvV) {
57.     if (srvV == "inactive" && srvK.loading() == 0
      && count(servers.getKey("active"))>1) {
58.     detach(conns.get(srvK).cRes, reqServ);
59.     detach(conns.get(srvK).cReq, srvKey.rServ);
60.     servers.remove(srvK);
61.     conns.remove(srvK);
62.   }  }   delay 25;  } }
```

Code snippet 6. Dynamic Load Balancer Server Description in jADL

[13]), which are automatically validated by the ACME Studio.The latter is an extension to Eclipse and provides explicit visualization support.

The $\pi-$ADL language [5] is a formal ADL designed to address the description of dynamic and mobile architectures. It provides the constructs to express both structural and behavioral viewpoints and has a textual and a graphical representation as well. It comes with a complete set of tools [5] under the ArchWare European Project. It can adequately express dynamic architectures and, as evaluated in [4], can support (using tools or other languages like π-AAL [12]) both foreseen and unforeseen dynamic reconfigurations. jADL, also provides these opportunities and extends them with the addition of the communication traits, which allows for bigger flexibility and expressiveness, especially when it comes to self-adaptable systems.

The PADL [1] is a process algebraic ADL with high expressiveness and analyzability. This language is equipped with TwoTowers, an open-source software tool for the functional verification, security analysis, and performance evaluation of software systems modelled in the ADL Æmilia [10]. But, the options provided for dynamic reconfiguration are limited compared to jADL. The PADL offers a class of predefined connectors for the architect to

choose from, whilst jADL considers them as first class entities. The PADL2Java software tool [3] is built to translate PADL models into Java implementation code stubs. It provides a library of software components for adding architectural capabilities to the targeted programming language.

5. Conclusion. In this paper, we have presented the implementation of a Load Balancer Architectural Pattern in jADL, as a case study, in order to exhibit jADL's architectural constructs for support of dynamism and self-adaptability architectural quality attributes. The jADL provides the necessary constructs for adequately expressing the behavior of this system – both its functional and non-functional requirements. With this case study it is shown that the software architecture, through ADLs, can be a useful tool for architects and various stakeholders. The next step for the evolution of jADL is to create a model for the assessment of the system concerning the embodiment of the quality attributes and constraints into the script as well as a model for translating it into compilable implementation code stubs.

REFERENCES

[1] A. Aldini, M. Bernardo, F. Corradini. A Process Algebraic Approach to Software Architecture Design, Springer, 2009.

[2] T. Batista, A. Joolia, G. Coulson. Managing dynamic reconfiguration in component-based systems. In: Software Architecture, vol. **3527** of LNCS, 439–480, Italy, 2005.

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

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

[5] E. Cavalcante, F. Oquendo, T. Batista. π-ADL: A Formal Description Language for Software Architectures. Technical Report, UFRN-DIMAp-2014-102-RT, 2014.

[6] P. Clements, F. Bachmann, L. Bass et al. Documenting Software Architecture Views and Beyond, 2nd Edition, Addison-Wesley, 2011.

[7] T. Erl, R. Puttini, Z. Mahmood. Cloud Computing: Concepts, Technology & Architecture. Published by Prentice Hall, 1st Edition, ISBN-13: 978-0-13-338752-0, 2013.

[8] D. Garlan, R. Monroe, D. Wile. Acme – Architectural Description of Component-Based Systems, In Foundations of Component-Based Systems, Cambridge University Press, 47–68, 2000.

[9] D. Garlan, R. Monroe, D. Wile. ACME: An Architecture Description Interchange Language, Proceedings of CASCON 97, Toronto, 169–183, 1997.

[10] `http://www.sti.uniurb.it/bernardo/twotowers/`, 2006.

[11] A. W. Kamal, P. Avgeriou. An Evaluation of ADLs on Modelling Patterns for Software Architecture. In Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007). Springer, Heidelberg, 2007.

[12] R. Mateescu, F. Oquendo. π-AAL: An Architecture Analysis Language for Formally Specifying and Verifying Structural and Behavioural Properties of Software Architectures. *ACM Software Engineering Notes*, **31**, No 2 (2006), 1–19.

[13] R. Monroe. Capturing software architecture design expertise with ARMANI. Technical Report CMU-CS-163, Carnegie Mellon University, 1998.

[14] A. Papapostolu, D. Birov. Structured Component and Connector Communication, In Proceedings of BCI'17, Skopje, Macedonia, doi: 10.1145/3136273.3136291, 2017.

Tasos Papapostolu
e-mail: papapostol@fmi.uni-sofia.bg
Dimitar Birov
e-mail: birov@fmi.uni-sofia.bg
Faculty of Mathematics and Informatics
Sofia University "St. Kliment Ohridski"
5, James Bourchier Blvd
1164 Sofia, Bulgaria

АРХИТЕКТУРНА САМОАДАПТИВНОСТ И ДИНАМИЧНА РЕКОНФИГУРАЦИЯ В jADL

Тасос Папапостолу, Димитър Биров

В последните години се наблюдава ускорен процес на развитието на нови технологии, като Internet of Things (IoT), мобилни и компютърни услуги и др. Софтуерните архитектури предоставят модели от високо ниво на абстракция, с чиято помощ се описват интернет-базирани приложения, като се вземат предвид техните качествени атрибути – сигурност, самоадаптивност, надеждност, бързодействие и други. В статията представяме базовите архитектурни конструкции на нов език за описание на архитектури (ADL), jADL, като използваме експериментален пример на самоадаптируем load balancer архитектурен стил. Езикът jADL е модерен език за описание на архитектури, който до момента описва и валидира динамични и мобилни архитектури. Многобройни статии показват отсъствието на промишлена употреба на ADLs поради две причини: (i) архитектурните езици предоставят теоретични формализми, които са удобни за извличане на определени свойства на архитектурите, но не са лесно разбираеми от стейкхолдери и (ii) липсата на инструменти, които да трансформират архитектурните модели от високо ниво към софтуерни артефакти, свързани с реализацията на софтуера. Езикът jADL е базиран на приложен pi-calculus, версия на Milner за асинхронно процесно смятане за изучаване на конкурентостта и взаимодействията между процесите. Този език предоставя интуитивна семантика на архитектурните елементи и комуникациите, което прави архитектурното описание разбираемо, дава възможност за моделиране на динамична архитектурна реконфигурация и позволява самоадаптивност на софтуерната архитектура, представена в статията.