Sofia University “St. Kliment Ohridski”

Faculty of Mathematics and Informatics

jADL, μσADL – Case Study of New Generation ADLs for Architecting Advanced Software Architectures

PhD Thesis

Conducted for the purpose of receiving the educational and scientific degree “Doctor of Philosophy” (PhD) in the field of 4.6. Informatics and Computer Science, Scientific Major "Computer Science"

Submitted by:

Anastasios Georgios Papapostolu

Advisor:

assoc. prof Dimitar Yordanov Birov, PhD

Sofia, 2019

Acknowledgments

I would like to express my biggest thanks to all those who contributed and supported me for the successful completion of this thesis.

To the late assoc. prof Dimitar Birov, who provided me huge support and offered ways to solve any problems I encountered through my PhD education, but could not see it complete.

To assoc. prof Aleksandar Dimov, who helped me finalize it and offered valuable advices for the final corrections of this dissertation.

To all the faculty and staff members of the Faculty of Mathematics and Informatics of Sofia University and their continuous support through all the years of my PhD studies.

Finally, I would, also, like to thank all the members of the committee for their help and collaboration.

Contents

List of Figures

List of Code Snippets

Chapter 1

Introduction

1.1 Software Architecture

Software architecture (Shaw and Garlan, 1996; Bass et al., 2013) over the last decades has matured and turned into a main engineering discipline. There are a lot of definitions (both formal and informal) concerning this term such as "*definition and structure of a solution that meets technical and operational requirements*" or "*a structured framework used to conceptualize software elements, relationships and properties*" or "*the decisions made about a system that are hard to change*" etc. A formal definition that best describes, in my opinion, its meaning is that software architecture is "*the set of structures needed to reason about a software system, which comprise software elements, relationships among them and properties of both*" (Clements et al., 2011). Every system encompasses its own architecture and as (Taylor et al., 2009) point out this architecture consists of *the set of principal design decisions made during its development and any subsequent evolution*. It provides the necessary means to software architects in order to "observe" a software system at an abstract level, thus allowing them to reason about a system's both functional requirements and quality attributes.

An important aspect of the discipline of software architecture is the adequate documentation of the architecture of a given system so that it can be used during design time, during the development process as well as during the evolution/maintenance of the system. A representation of a software system should be both detailed enough, so that it adequately describes the system, and useful for the communication of the architecture between the various stakeholders. So, in pursuance of the best way to describe the architecture of a software system a number of approaches have been proposed, which focus on and define different perspectives to represent different aspects of a system. Each one of these perspectives, usually, contains a number of architectural views (or viewpoints).

As software systems become more and more complex, a way to achieve their effective documentation is to "divide" it in three parts – called *perspectives*, each one accompanied by a number of *views* (Clements et al., 2011). The three perspectives defined are the *static*, the *dynamic* and the *allocation*. Each one of these perspectives is explained and presented in the following paragraphs of this section. There are other approaches proposed concerning the successful documentation of software architectures like the Rational Unified Process (RUP), a five-view approach based on the classification proposed by Kruchten (Kruchten, 1995). It is comprised of

four main views; i) logical (containing design classes), ii) implementation (architectural decisions concerning the implementation), iii) process (consisting of tasks and threads) and iv) deployment (concerning the physical nodes for the platform configurations). The fifth (plus-one) view consist of various use cases and scenarios concerning the behavior of the software system. Other approaches are the Rozanski and Woods Viewpoint Set (Rozanski and Woods, 2005), where they suggest a set of six views (or viewpoints) for the documentation of software architectures. These six viewpoints are based on the extension of the Kruchten 4+1 set. Additionally, there are proposals which specialize in concrete software development approaches, like Agile for example. In (Schwaber and Beedle, 2001; Beck and Andres, 2004) can be found effective ways of documentation concerning user stories, short iterations, etc., which are some common practices in instances of the Agile software development approach like Scrum, Extreme Programming, etc.

In this thesis, we will be dealing with the first approach mentioned, the one proposed in (Clements et al., 2011). The three perspectives defined in this approach present a similarity with the 4+1 approach, since, in a way, the dynamic perspective can "contain" the implementation and process ones from the 4+1. We believe that the segregation of the perspectives in (Clements et al., 2011) is a more abstract way of representing the software architecture of a system and it, also, provides suitable views for describing dynamic reconfigurations and expressing behaviors, which constitute an important factor in the scope of this thesis. Such a view, is the *component and connector view* presented below.

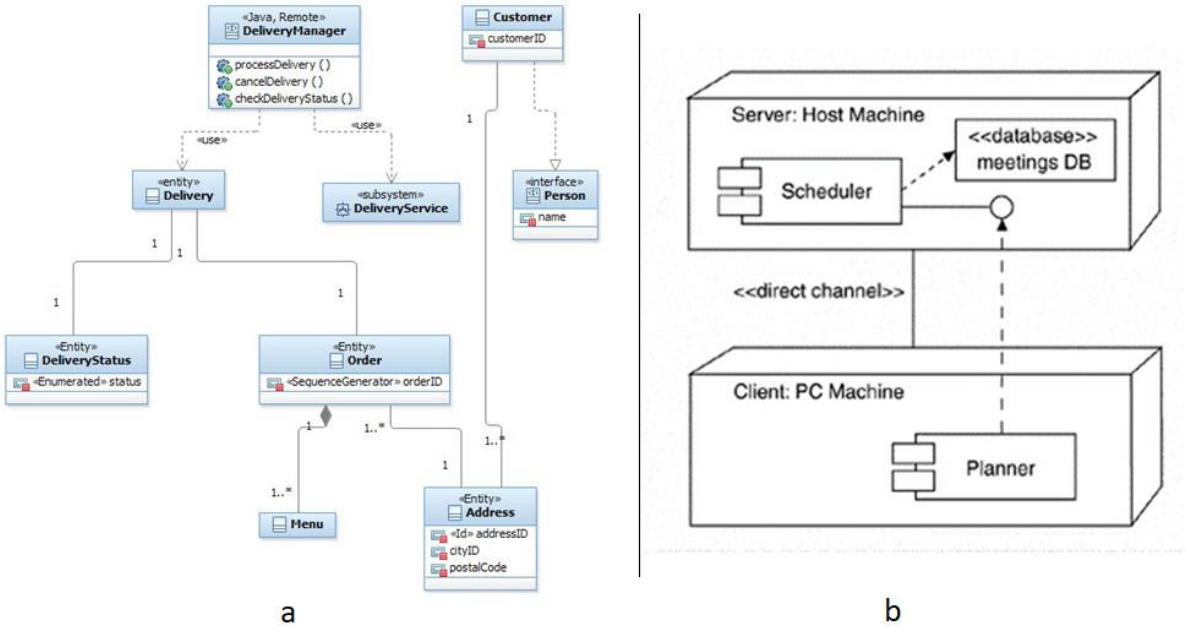


Fig. 1. *Example views from the: (a) static and (b) allocation perspectives*.

The first perspective proposed is the *static* perspective of a system. It concerns with the static parts of a system and it helps architects to reason about how the implementations units of a system are structured. Therefore, an example view of this perspective could be a UML diagram that consists

of the classes that need to be implemented for the realization of a given functionality of the system. The second perspective proposed is the *allocation* (or *deployment*) perspective. Here, it is described the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment (the hardware that a system needs, the technical environment requirements for each element etc.), showing how the software structures correspond to the environment's structures. An example of architectural views of these two perspectives can be seen in figure 1.

The third perspective proposed is the *dynamic perspective* of a system. It outlines the runtime behavior of the system, how this structured set of elements interact dynamically with one another during the execution of a system. One of the most important views in this 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. In figure 2 below, we can see such an architectural view of a system. I will be dealing extensively with this kind of view and more concretely with its expression using an Architectural Description Language.

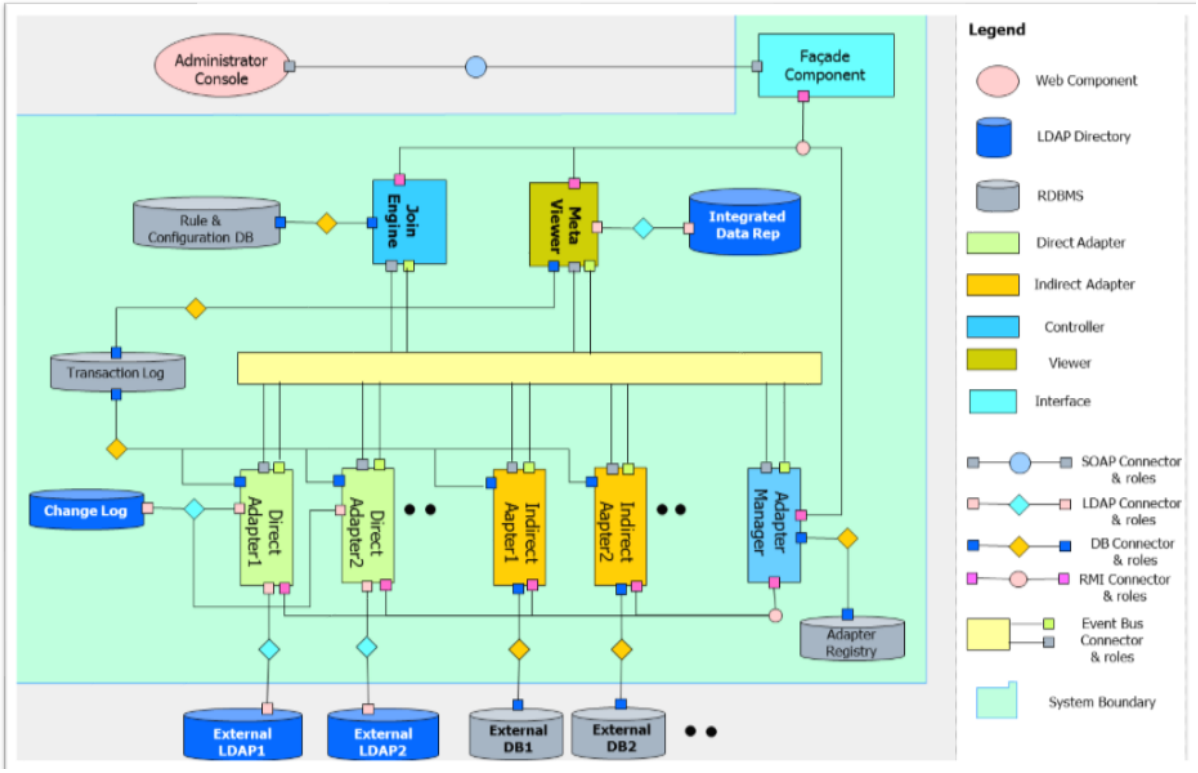


Fig. 2. *A Component-and-Connector View of a system.*

Components are the computational and data store elements (locus of computation) and they communicate with their environment only through their declared ports. Components can communicate with each other and their environment exclusively through connectors. *Connectors* represent the various forms of communication between the various components or the components

with their environment (locus of communication) 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. The architectural elements, their interconnections and the constraints concerning them compose the *topology* 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.

The communication, data flow and component-connector interactions describe the behavior of the software architecture according to the topology. If the topology or the behavior 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-** (Kephart and Chess, 2003). These systems can be defined as systems capable of managing themselves, where * stands for a number of properties: self-configuration, self-adaptation, self-diagnose, self-repair, etc. Architectural dynamism is a natural feature of current social media applications, IoT, mobile computing, cloud and other advanced distributed applications.

1.2 Domain Specific Languages

Domain Specific Languages (Fowler, 2010) (DSLs) are computer languages of, usually, limited expressiveness specifically designed to address a concrete set of problems of a certain domain, in contrast to General Purpose Languages (GPLs) which can be used across multiple domains. Examples of DSLs are the Structured Query Language (SQL – managing data in a relational database), or the Hyper-Text Markup Language (HTML – creating web pages). These languages are applicable only to their concrete domains, unlike C or Java which as GPLs can be used for solving a variety of problems across various domains. But, despite the diversity provided, GPLs can be very verbose and usually harder to learn and use than DSLs, especially for domain experts that do not deal with programming extensively as part of their everyday job. Also, the level of abstraction provided most of the times is smaller than the one provided in DSLs. On the other hand, DSLs are tailored for a specific application domain, hence they can be more expressive and easier to use. Another important aspect of DSLs, is that their integrated domain experts' knowledge can be leveraged to an inexperienced user. They focus on communicating solutions and they provide a lot of flexibility, which results in less and easier to read and write code. Additionally, since they are targeting specific problems, they are relatively small languages, which means less maintenance and increased productivity.

In the figure below, a simple example is presented of how, with the use of a DSL, a description can be transformed to either an interchangeable XML specification or a more visually human-friendly version.

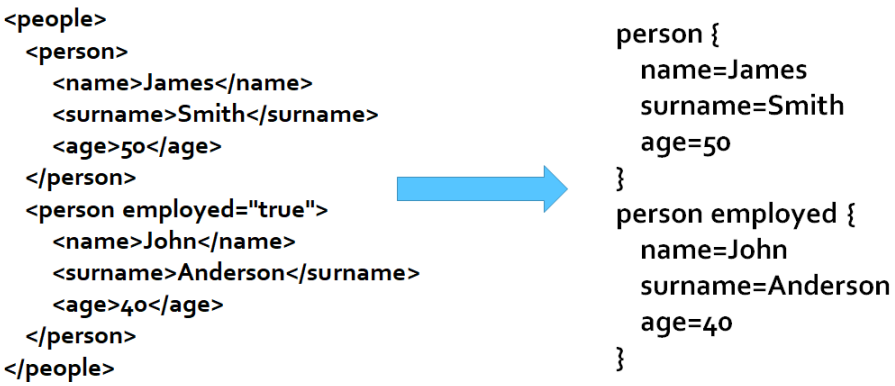


Fig. 3. *A simple transformation that can be easily achieved through a DSL*.

DSLs are divided into two major categories: *internal* (or *embedded*) and *external* DSLs.

- Internal DSLs are defined using a host language to give a different "feel" on the language and to use it in a more standardized and easier way from the people on a certain domain. Since they are "riding" on a host language they are both influenced and restricted by this language. Their major advantage is that the host language covers the needs regarding the grammar and the parser and they can benefit from existing tools developed for the particular language.

- External DSLs are built from the ground up and they require a custom defined parser for translating the syntax into something a computer understands and can be used. Since they are independent from other existing languages, they provide great flexibility for defining the grammar, regarding the syntax, operators, structure, etc. On the other hand, they require greater effort because the compiler that will parse and process the syntax and map it to the appropriate semantics takes more time and hard work to compile it.

Another distinction that can be made is between textual and graphical DSLs.

1.3 Architecture Description Languages

Architecture Description Languages (Medvidovic and Taylor, 2000) (ADLs) are DSLs used in the domain of software architecture and software engineering in order to formally describe system architectures. They have a high level of abstraction and they, usually, ignore lower level implementation details. By using formal methods, they manage to verify, validate and ensure syntactical and semantical correctness of the software architecture. Tools are, usually, provided by an ADL to perform various actions to an architectural description, like simulation, generation of software artefacts (e.g. implementation code stubs), etc.

There is a large number (over 120) of developed ADLs through the years, focusing on different domains addressing different concerns. Since there is great variability in the concerns of

stakeholders across various domains, it would be quite unlikely for a single ADL to address all of them. Therefore, ADLs tend to focus to the issues of their particular domain, providing varying options for the description/validation/analysis/etc. of the architecture of a software system. For example, Wright (Allen, 1997) provides the means for specifying complex interaction mechanisms, while Rapide (Luckham, 1996) offers possibilities for the simulation of the architectures.

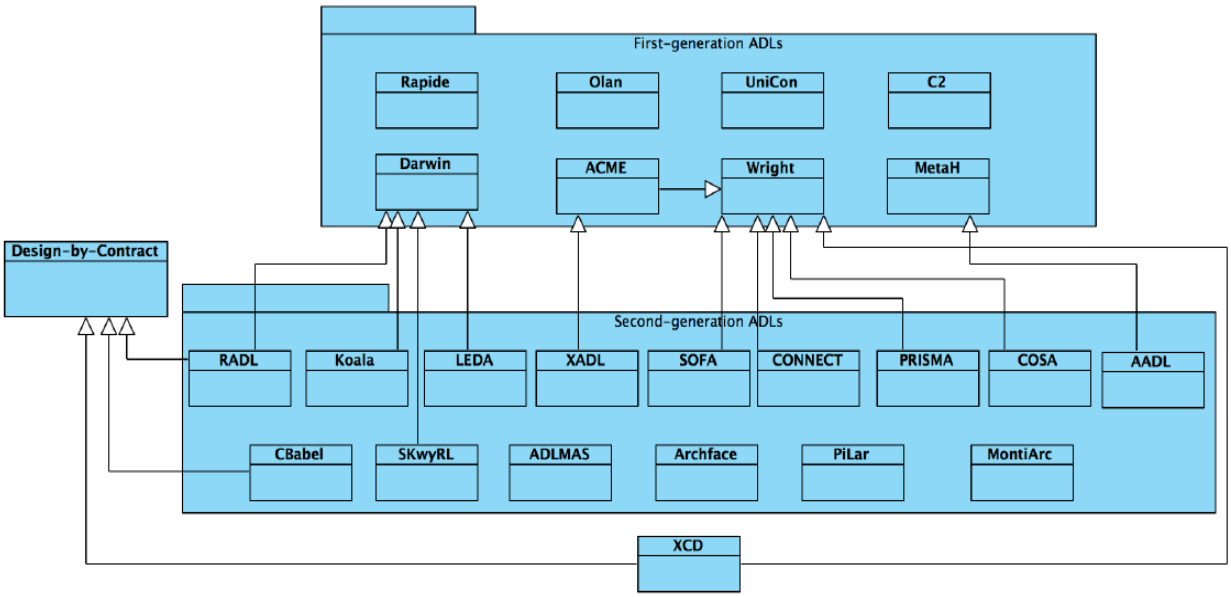


Fig. 4. *Relationships between some of the early (before 1999) and some of the recent ADLs (starting from 1999)* – reprinted from (Ozkaya, 2014).

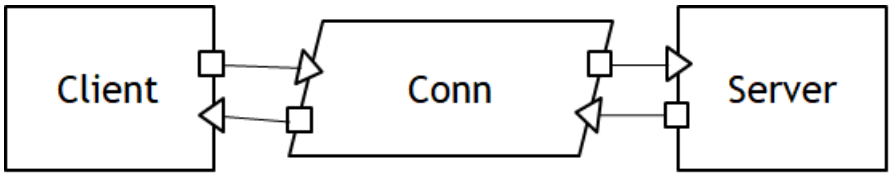
The main building blocks in most ADLs are *components*, *connectors* and *configurations*. ADLs provide the possibility for both describing the structural specification and expressing the behavioral aspects (i.e. how these elements in the structure interact) of the given architecture. ADLs usually present the architecture of a software system in a visual way, as a graph of interconnected components and connectors, which constitutes the topology of the system.

1.3.1 Categories of Architecture Description Languages

The existing architecture description languages can be categorized in various ways and research has been conducted regarding this issue, with one of the most characteristic being the framework for classification and comparison from (Medvidovic and Taylor, 2000).

A common way that ADLs are divided is between first and second generation of ADLs. As seen in figure 4, ADLs before 1999 belong to the first category, and after 1999 to the second. One important issue addressed from second generation ADLs, is the expression of the behavior of the architectural description during run-time and the need for dynamic reconfiguration of an architecture.

Also, they can be divided based on their (or the lack of) capabilities of providing a graphical way to represent the described architecture.



a

```
1. interface IReceive {
2.     service void Received (Type data);
3. }
4. //...
5.
6. component Client {
7.     provides port IReceive wait;
8.     //...
9.
10.    config wait as {
11.    service void Received (Type data) {
12.      //...
13.    }  }
14. }
15. component Server {
16.    //server declarations
17.    //...
18. }
```

```
19. connector Conn {
20.   provides role IRequest cReq;
21.   //...
22.   config cReq as {
23.   service void aRequest (Type data) {
24.     //...
25.   }      }
26. }
27.
28. architecture ClientServer {
29.   instance client = new Client();
30.   //...
31.
32.   attach(conn.cReq, client.send);
33.   //...
34. }
```

b

Fig. 5. *A (a) visual representation and (b) textual (partial) description of a Client-Server architecture using the ADL jADL* - an ADL with explicit support for connectors.

We view connectors as important architectural elements that ADLs should provide as first-class entities for the definition of architectures. Therefore, an important taxonomy regarding ADLs, is the one that classifies them according to their support for defining connectors, proposed by (Amirat and Oussalah, 2009):

- *ADLs with implicit connectors*. They do not support connectors because they distort the compositional nature of software architecture. ADLs like Darwin (Magee et al., 1995) and Rapide (Luckham 1996) do not consider connectors as first-class entities. Inside components, apart from the computations, the coordination is entangled too, leading to more complex and harder to reuse components.

- *ADLs with predefined set of connectors*. UniCon (Shaw et al., 1996) is an example of such languages. The connectors are predefined and built-in in the language. Though reusability

is improved compared to the previous category, the language still poses limitations since connectors cannot be evolved.

- *ADLs with explicit connector types*. Most ADLs fall into this category by considering connectors as first-class entities of the language. The computations are described inside components and connectors describe the interaction mechanisms between them, thus separating computation from coordination and promoting their reusability. Examples of such languages can be found both in early ADLs, such as Wright (Allen, 1997), and newer ones, like π-ADL (Oquendo, 2004).

1.3.2 Use of Architecture Description Languages

Despite their large number and the benefits offered by ADLs, especially in design time, their use outside of academia is still limited. As far as practitioners are concerned, the high degree of formality in these languages, makes them hard to learn and to integrate them in industrial processes. They tend to use informal ways to describe software architectures, such as UML or box-line diagram drawing tools, as indicated by a number of surveys like (Ozkaya, 2016; Malavolta et al., 2012). It is worth noticing that in (Ozkaya, 2016) for example, the majority of the participants (68%) were not aware of ADLs and even a third of them who were, ignored the capabilities of ADLs when it comes to analysis.

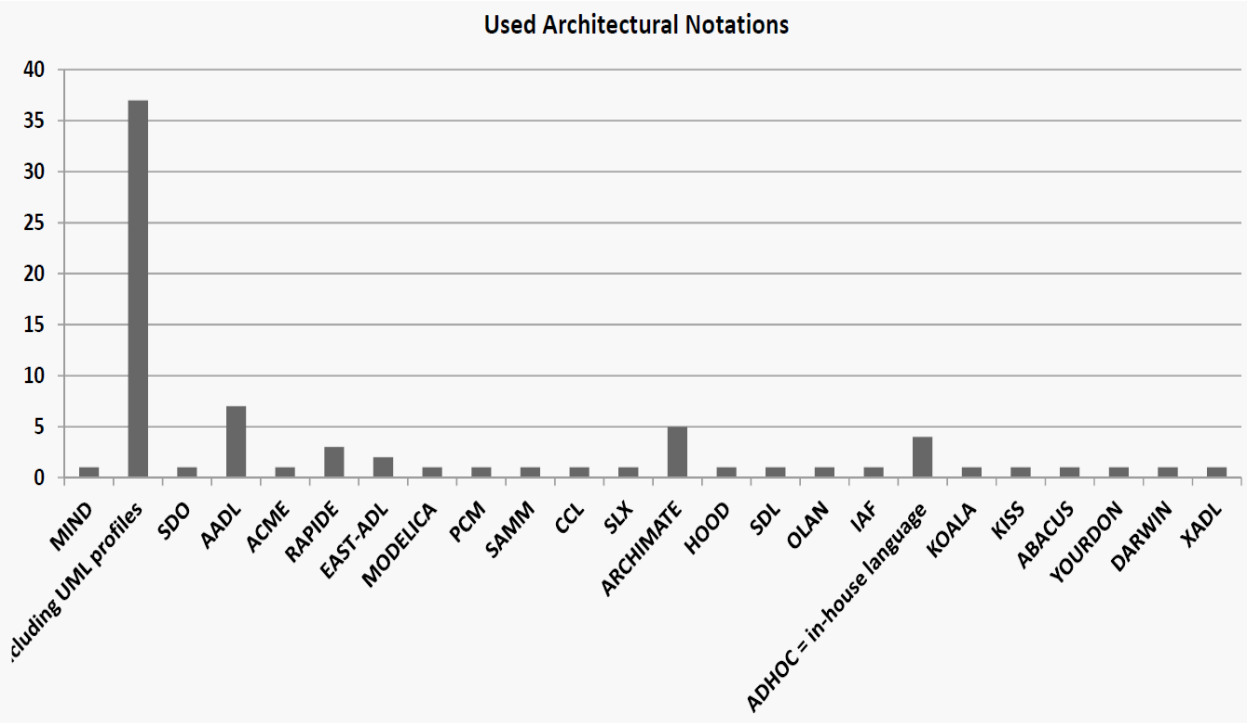


Fig. 6. *Usage of ADLs in industry* – reprinted from a research conducted by (Muccini, 2013).

Another problem indicated regarding ADLs is their support in the handling of dynamic reconfigurations as indicated by (Minora et al., 2012). Also, the tool support for these languages is relatively poor, especially compared to programming languages or informal ADLs like UML. Finally, when it comes to modern architectural styles, such as MicroServices, as (Francesco, 2017) point out there is a lack of an ADL to formally describe microservice architectures and architects tend to use informal modelling languages that describe service-based architectures like SoaML (SoaML 2019).

1.4 Thesis Goals

In the previous section, problems surrounding the ADLs have been outlined. In this thesis, I am trying to address mainly two of them: their, usually, problematic (for developers) high degree of formality in their syntax and their issues when it comes to expressing dynamic architectures, in order to contribute to their continuing evolution.

The goal of this thesis is to create a new Architecture Description Language, named jADL, which should provide means to formally describe dynamic and mobile software architectures with a relatively simple syntax. It should also offer architects and stakeholders the necessary means and constructs so that the dynamic reconfigurations met in todays' systems can be adequately expressed. Additionally, it should support modern architectural styles (e.g. microservices) and be accompanied by a tool that would ease and support its use.

The objectives of this research are:

- creation of a new generation architecture description language for the expression of modern and dynamic architectures, named jADL, which should:
    - define of such syntax and structure for the language, that will help towards the promotion of ADLs in further use outside out of academia, by being simple and familiar to practitioners.
    - provide the means and language constructs for expressing dynamic reconfigurations of a given architecture.
    - support the description of modern architectural styles, such as MicroServices.
- development of a tool for the support of the language.
- validation of the language by describing well-known and much used architectural patterns (e.g. Enterprise Service Bus) and more complex modern architectures.

1.5 Publications Related to Thesis

Most of the presented work has been published to various conference proceedings. Below follows a list with the references divided in journal and proceedings publications.

Publication in journal:

- A. Papapostolu, D. Birov, *Architecture Evolution Through Dynamic Reconfiguration in jADL*, Information Technologies and Control, 2017 (1), pp. 23-32. Available at: http://www.aksyst.com:8081/Sai/Journal/Docum/4-papapostoulu_engl_1_17-color.pdf

Publications in conference proceedings:

- T. Papapostolu, D. Birov, *Architectural Self-Adaptation and Dynamic Reconfiguration in jADL*, in proceedings of the 47th conference of the SMB, Borovets, Bulgaria, pp. 168-177, 2018. Available at: http://www.math.bas.bg/smb/2018_PK/tom_2018/pdf/168-177.pdf

- T. Papapostolu, D. Birov, *Towards a Methodology for Designing Micro-service Architectures Using μσADL*, In Lecture Notes in Business Information Processing book series (LNBIP, vol. 319), Springer-Verlag, 2018, pp. 421-431, 2018. Available at: https://link.springer.com/chapter/10.1007/978-3-319-94214-8_33

- T. Papapostolu, *Utilizing Frameworks for Developing DSLs for Automated Transformation of ADLs*, In proceedings of the Doctoral Conference "Young Scientists", Sofia, Bulgaria, pp. 542-551, 2018.

- A. Papapostolu, D. Birov, *Structured Component and Connector Communication*, Proceedings of International Conference "Balkan Conference in Informatics '17", ACM Digital Library, 2017. Available at: https://dl.acm.org/citation.cfm?id=3136291

- A. Papapostolu, D. Birov, *Dynamic Reconfiguration Statements and Architectural Elements in jADL*, In proceedings of the International Conference "Automatics and Informatics '16", Sofia, Bulgaria, pp. 153-157, 2016.

- A. Papapostolu, D. Birov, *jADL: Another ADL for Automated Code Generation*, In proceedings of International Conference "Science and Business for Smart Future", Varna, Bulgaria, pp. 10-18, 2016.

1.6 Thesis Structure

The structure of the rest of this thesis is:

- In the next chapter previous related work regarding architecture description languages is presented. A number of languages is examined in terms of how they describe architectures.

- In chapter 3, the architecture description language jADL is presented. Its syntax and semantics are explained. Additionally, a couple of example architectural descriptions are presented with emphasis on the constructs that the language offers for dynamic reconfiguration.

- In chapter 4, the extension of jADL is presented, named μσADL (which focuses on the description of microservices). Through illustrative examples, the applications of the language are presented.

- In chapter 5, the tool designed to support the language is presented. It is built using the Xtext framework for development of domain-specific languages.

- In chapter 6, conclusions and future work is discussed.

- In the Appendix section, the syntax of jADL is presented in detail using the extended Backus-Naur form. Also, the code for the realization of the tools is presented.

- In the final chapter, Bibliography, the complete list of the references used for this thesis can be found.

Chapter 2

Related Work

2.1 Introduction

In this chapter a number of both formal architecture description languages and informal languages is examined and presented.

There is a big number of architecture description languages, more than 120 such languages are defined through the years. A lot of scientific research has been conducted regarding them, as can be seen for example in (Clements, 1996; Medvidovic and Taylor, 2000; Malavolta et al., 2012). For this thesis, I performed an analysis so that I can obtain a representative subset of languages to compare, which covers their evolution. This was achieved by the classification and then the extraction of a sample of languages based on the important (in the context of this thesis) criteria of their support for dynamic reconfiguration and the definition of complex and user-defined connectors.

First, as mentioned in the previous chapter, there are various ways regarding the classification of architecture description languages. I focus on two of them; the division in generations (first and second) and their support for connectors. The first one was chosen because one of the main distinctions between the two generations is that the second one tried to address the problem of the expression of run-time behavior and dynamic reconfigurations. The second way was chosen because I believe the separation of computation and communication eases the reuse of components and/or complex communication protocols between them. Therefore, the languages were, also, divided, as proposed by (Amirat and Oussalah, 2009), in three categories based on their support for complex user-defined connectors. The three categories are architecture description languages with: i) implicit connectors (no connectors defined and communication between components is modeled through simple attachments), ii) predefined set of connectors (connectors are recognized as important elements but, still, the support is limited to only the ones integrated in the language) and iii) explicit connectors (connectors are treated as first-class entities, thus separating computation from coordination and communication).

Based on the above mentioned criteria, the languages that constitute the subset of the architecture description languages presented in this chapter, were chosen. From the first generation for example, Darwin as one of the first architecture languages was chosen, alongside Wright and Rapide which provide different support for connectors. From the second generation, languages

from all three different categories for connector support were chosen. Also, emphasis was given on their focus on behavior specification, like π-ADL for example.

Other important factors I took into account were their industrial usage and their focus regarding the descriptions (e.g. structural or runtime, analysis, simulation, etc.). Their industrial usage still remains a problem, as shown in the previous chapter, and is one of the issues I am trying to address in this thesis. Therefore, languages with known industrial use were chosen like Koala. Additionally, languages that focus on different capabilities were chosen, like for example AADL (analysis) and Xadl (flexibility/extensibility).

Finally, languages that influenced the creation and syntax of jADL were chosen. jADL is an architectural description language, created in this thesis, with a focus on the description (both structural and behavioral) of dynamic software systems. It is presented through the next chapters, starting from the following one. Such languages are for example π-ADL and ACME.

In the rest of this chapter, the obtained subset of architecture description languages is presented. The examination and analysis of each of these languages consists, mainly, of three points:

- the way that they describe components.
- their support for connectors.
- an example of a simple Client-Server architectural description.

Finally, in the conclusion of this chapter, aggregated results regarding the languages examined are shown. Also, the main reasons for the creation of an additional language are presented based on the findings of this analysis.

2.2 Darwin

Darwin (Magee et al., 1995; Imperial College of Science, Technology and Medicine, 1997), one of the first Architecture Description Languages, encompasses a component-based approach to describe architectures, with a focus on distributed applications. Components are defined with the use of interfaces, which represent services that the component either provides to or requires from its environment. It, also, supports the definition of composite components, with the creation of instances of primitive components and the attachments of their interfaces (such as the component System from the code below), thus allowing for the specification of more complex behavior. Connectors are not considered to be first-class entities in Darwin, so there is no such specification when describing an architecture. The communication can be specified in the bind section of the definition of composite components. In this way, the interaction mechanisms are encapsulated inside the components, thus making them more complex and harder to reuse. A number of tools can be used in order to verify and analyze a Darwin architectural description, but first it is required a formal behavior specification with the use of the Finite State Process (FSP) language (Magee et al., 1999).

Darwin Architectural Description

```
1. component Server {
2.     provide p;
3. }

4. component Client {
5.     require r;
6. }

7. component System {

8. inst
9.     A:Client;
10.    B:Server;

11. bind
12.    A.r -- B.p   }
```

Code Snippet 1. *Client-Server description in Darwin* – adapted from (Magee et al., 1995).

2.3 Wright

Wright (Allen, 1997) is an Architecture Description Language that, also, follows the component/connector/configuration style for describing architectures. Components in Wright express independent computations and are defined in two main sections, as can be seen on the example code below. First, the interface part, which consists of ports, defines the interaction points of the specified component. Second, the computation part defines the behavior of the element when it interacts with its environment. Connectors express the communication between components and, similarly to components, are defined in two parts. First, the interface part, consisting of roles, defines its point of interaction with its environment. Second, the Glue specification of a connector (respectively to the computation in components) defines the behavior of the connector. By treating both components and connectors as first-class elements, Wright increases the independence, reusability and eases the analysis of the architectural elements and the whole architectural description. Also, when it comes to patterns, this increases the support for modeling pattern-specific variability (e.g. each pipe in a pipe-filter pattern can express its own glue specification) (Kamal and Avgeriou, 2007). Finally, the last thing that needs to be defined is the configuration section (lines 12-19 below). It is comprised of a set of uniquely named component and connector instances. Once they are declared, the attachments of the components' ports to the connectors' roles define the topology of the described architecture. A number of tools can be used (e.g. consistency and completeness checks using FDR), but first a Wright description has to be translated to CSP (a tool provides the possibility to be done automatically) (Wright tools).

In an effort to satisfy the need for dynamic reconfiguration of the architecture, Dynamic Wright (Allen et al., 1998) was created as an extension and more control events and statements were added

to the language (such as remove, attach, etc.) (Minora et al., 2012). The language provides very good support to determine the moment to perform a dynamic reconfiguration, but it is limited to support of only foreseen dynamic reconfigurations (Minora et al., 2012).

Wright Architectural Description

```
1. System ClientServer

2.      component Server =
3.            port provide [provide protocol]
4.            spec [Server specification]
5.      component Client =
6.            port request [request protocol]
7.            spec [Client specification]
8.      connector C-S-connector =
9.            role client [client protocol]
10.           role server [server protocol]
11.           glue [glue protocol]

12. Instances
13.     s: Server
14.     c: Client
15.     cs: C-S-connector

16. Attachments
17.     s.provide as cs.server
18.     c.request as cs.client

19. end ClientServer
```

Code Snippet 2. *Client-Server description in Wright* – adapted from (Barros, 2005).

2.4 Rapide

Rapide (Luckham, 1996) is an architecture description language with a focus on dynamic architectures and the simulation of architectures as suggested, also, in the research by Ozkaya (2014). Rapide specifies components through interfaces. They can be used in order to model both synchronous and asynchronous types of communications and can, also, include behavioral specifications. Composite component types are supported by Rapide, where a configuration of components can be included as a computation (in a similar way that the element architecture is defined in the example code below). Since Rapide adopts a component-based approach, it does not consider connectors as first-class entities in the language and the communication mechanisms are integrated inside the component specifications. This leads to harder to reuse and more complex components.

An important aspect of this language, as indicated by Ozkaya (2014), is the introduction of architectural constraints. They serve as global coordinators ensuring the compliance of the

components participating in the architectural specifications to a certain set of rules. But, again due to the lack of first-class connectors, their reuse is difficult in other specifications, since they are integrated inside a given architectural specification.

Rapide Architectural Description

```
1. type ClientInterface() is interface
2.     action out get();
3.     action out set();
4.     behavior
5.         ...
6. end Client;

7. type ServerInterface() is interface
8.     action in get();
9.     action in set();
10.    behavior
11.        ...
12. end Server;

13. module Client () return ClientInterface is
14. --internal actions
15.    ...
16. --internal behaviour
17.    ...

18. module Server () return ServerInterface is
19. --internal actions
20.    ...
21. --internal behaviour
22.    ...

23. architecture Client_Server is
24.    ClientIns : Client();
25.    ServerIns : Server();
26. connect
27.    ClientIns.get() ⇒ ServerIns.get();
28.    ClientIns.set() ⇒ ServerIns.set();
29. end architecture Client_Server;
```

Code Snippet 3. *Client-Server description in Rapide* – adapted from (Ozkaya, 2014).

2.5 ACME

ACME (Garlan et al., 1997) started as a multi-style ADL framework providing the possibility of using it as a common interchange platform for multiple ADLs. The main elements used in ACME architectural descriptions are: components, connectors, systems (which represent graphs of interconnected components and connectors) and properties (which provide semantic information regarding the architectural elements). In the example code below, it can be seen how the graph (system) is defined through the connections between the architectural elements, in the attachments

section of the description. ACME supports user defined architectural constraints on its elements, in terms of so-called invariants, with the use of one of its extensions named Armani (Monroe, 1998). It provides explicit configurations and hence facilitate understandability and readability. It supports architectural styles, and provides a template mechanism for their implementation (Kamal and Avgeriou, 2007).

While the need for dynamic reconfiguration grew over the years and since it was not "integrated" in ACME the help of additional tools/extensions is needed (e.g. ACME/Plastik (Batista et al. 2005)). Despite the various extensions created, there are still issues when it comes to dynamic reconfiguration. For example, when it comes to unforeseen dynamic reconfigurations or reconfigurations that require behavior reconfiguration ACME/Plastik relies on external languages (Minora et al., 2012).

ACME Studio (The Acme Studio Homepage 2009) is a software tool built as an extension for the Eclipse environment, integrated in it as a plugin. It provides a user-friendly interface for the editing of architectural descriptions based on the Acme Architectural Description Language. The tool provides support both for editing the architecture textually and visually.

ACME Architectural Description

```
System simple_cs = {
    Component client = {
        Port sendRequest;
        Properties { requestRate : float = 17.0;
                     sourceCode : externalFile = "CODE-LIB/client.c" } }

    Component server = {
        Port receiveRequest;
        Properties {
                     idempotent : boolean = true;
                     maxConcurrentClients : integer = 1;
                     multithreaded : boolean = false;
                     sourceCode : externalFile = "CODE-LIB/server.c" }
    }

    Connector rpc = {
        Role caller;
        Role callee;
        Properties {
                     synchronous : boolean = true;
                     maxRoles : integer = 2;
                     protocol : WrightSpec = "..." }
    }

    Attachments {
        client.sendRequest to rpc.caller ;
        server.receiveRequest to rpc.callee }
}
```

Code Snippet 4. *Client-Server description in ACME* – adapted from (Garlan et al., 2000).

2.6 Koala

Koala (van Ommering et al., 2000) is another component-based oriented architectural description language that focuses on the description of software architectures of the more and more complex software in consumer electronics products. Components are the computational units and they communicate through their interfaces. In Koala, interfaces are considered as first-class entities and they are used to model the connections between components at a higher level. The configuration of the architectural topology is described in two parts; first declaring the instances and then interconnecting their interfaces. Despite the existence of interfaces, the lack of connectors as first-class entities doesn't allow to describe complex communication mechanisms. As seen in the code below, interactions are simply described in the connects section of the declaration of a composite component. Finally, Koala does not allow for specification of architectural behavior of the components.

Koala Architectural Description

```
1.  interface data_interface {
2.      void get();
3.      void set();
4.  }

5.  component client {
6.      requires data_interface client_in;
7.      provides data_interface client_out;
8.  }

9.  component server {
10.     provides data_interface server_in;
11.     requires data_interface server_out;
12. }

13. component client_server_ex {
14.     contains      client clientIns;
15.                   server serverIns;

16.     connects      serverIns.server_in = clientIns.client_out;
17.                   clientIns.client_in = serverIns.server_out;
18. }
```

Code Snippet 5. *Client-Server description in Koala* – adapted from (Ozkaya, 2014).

2.7 XADL

xADL (Dashofy et al., 2001) is highly extensible and flexible xml-based architecture description language. It describes commonly used, in such languages, elements like components and connectors, using XML schemas (Kotha, 2004). This way it takes advantage and integrates in the language their high interchangeability and modularity. So, it provides the possibility for easy reuse

of common features and easy creation of new features which can extend the language as new first-class entities.

It has both a textual and a graphical representation and provides two separate schemas for the definitions of run-time specifications and design-time aspects of a system (Dashofy et al., 2002). The Instances schema consists of instances of common architectural constructs like components/interfaces/connectors/etc. and the Structure & Types schema, which consists of types for these elements plus a generic type system. The two schemas can be separately extended.

Another important advantage that comes with the use of XML standards is the fact that there is a great number of available tools that can be used in order to hide unnecessary XML-related details, such as XML authority (XML Authority 2017), XML Spy (XMLSpy 2019), etc. Additionally, a number of tools has, also, been developed to support the language, like ArchEdit (Kotha, 2004), a graphical tree-based editor.

XADL Architectural Description

```
archInstance{
  componentInstance{
    (attr) id   = "clientComp"
    description = "Client"

    interfaceInstance{
      (attr) id = "clientComp.IFACE_TOP"
      description = "Client Top Interface"
      direction = "inout"
    }
    interfaceInstance{
      (attr) id = "clientComp.IFACE_BOTTOM"
      description = "Client Bottom Interface"
      direction = "inout"
    }
  }

  connectorInstance{
    (attr) id   = "conn1"
    description = "Connector 1"

    interfaceInstance{
      (attr) id = "conn1.IFACE_TOP"
      description = "Connector 1 Top Interface"
      direction = "inout"
    }
    interfaceInstance{
      (attr) id = "conn1.IFACE_BOTTOM"
      description = "Connector 1 Bottom Interface"
      direction = "inout"
    }
  }
```

```
componentInstance{
  (attr) id   = "serverComp"
  description = "Server"

  interfaceInstance{
    (attr) id = "serverComp.IFACE_TOP"
    description = "Server Top Interface"
    direction = "inout"
  }

  interfaceInstance{
    (attr) id = "serverComp.IFACE_BOTTOM"
    description = "Server Bottom Interface"
    direction = "inout"
  }  }

linkInstance{
  (attr) id = "link1"
  description = "clientComp to Conn1 Link"
  point{
    (link) anchorOnInterface = "#clientComp.IFACE_BOTTOM"
  }
  point{
    (link) anchorOnInterface = "#conn1.IFACE_TOP"
  }  }

linkInstance{
  (attr) id = "link2"
  description = "Conn1 to serverComp Link"
  point{
    (link) anchorOnInterface = "#conn1.IFACE_BOTTOM"
  }
  point{
    (link) anchorOnInterface = "#serverComp.IFACE_TOP"
  }  }   }
```

Code Snippet 6. *Client-Server description in XADL* – adapted from (xADL Concepts and Info 2003).

The example code above is expressed in more human-readable way without using XML notation. A description of the Client component using XML notations can be seen in the example below. Due to the multiple schemas specified the actual architectural description can get quite complicated, so non-XML notation can be used too.

XADL Architectural Description

```
<component type="Component" id="clientComp">
  <description type="Description">Client</description>
  <interface type="Interface" id="clientComp.IFACE_TOP">
    <description type="Description">Client Top Interface</description>
    <direction type="Direction">inout</direction>
  </interface>
```

```
7.   <interface type="Interface" id="clientComp.IFACE_BOTTOM">
8.     <description type="Description">Client Bottom Interface</description>
9.     <direction type="Direction">inout</direction>
10.   </interface>
11. </component>
```

Code Snippet 7. *Component using xml notation* – adapted from (xADL Concepts and Info 2003).

2.8 AADL

Architecture Analysis & Design Language (Feiler et al., 2006) (AADL) is designed with a focus on the specification and analysis of real-time performance-critical distributed computer systems and supports a model-driven development approach (Architecture Analysis and Design Language 2015). It has a textual and a graphical representation. A significant difference with the languages discussed so far, is that there is a fixed set of component categories for the architect to choose from, when defining the architecture. There are three categories (Feiler et al., 2006): i) application software, which consists of thread, thread group, process, data and subprogram types. ii) execution platform, which consists of processor, memory, device and bus types, and iii) composite, which consists of system types for the specification of composite types. Such declarations can be seen in the subcomponents section in the code below.

AADL Architectural Description

```
1. system implementation server_s.impl

2.     subcomponents
3.           server_proc: process server_p.impl;
4.           ConfidentialData: data database;
5.           otherComp: system otherComponents {cs_properties::vulnerability => true;};
6.     connections
7.           getData: data access ConfidentialData -> server_proc.dataAcc;
8.           inReq: event data port p_serv -> server_proc.serv_p;
9.           outResp: event data port server_proc.serv_p -> p_serv;
10.          accessDataOther: data access ConfidentialData -> otherComp.dataAcc
{cs_properties::vulnerability => true;};
11.          comOut: event data port otherComp.inout -> otherCom;
12.          comIn: event data port otherCom -> otherComp.inout;
13.    flows
14.          process_req: flow path p_serv -> inReq -> server_proc.processReq -> outResp
-> p_serv;
15.    annex Behavior_specification { ... }
16. end server_s.impl;

17. system implementation client_server_arch
18.
19.    subcomponents
20.          Internet: system openNetwork.impl {cs_properties::vulnerability => true;};
21.          client1: system client {cs_properties::securityLevel => 3;}
22.          server1: system server_s.impl;
23.    connections
```

```
24.          client_serv_1: event data port client1.cl_p -> Internet.entryCl;
25.          client_serv_2: event data port Internet.entryServ -> server1.p_serv;
26.          server_cl1: event data port server1.p_serv -> Internet.entryServ;
27.          server_cl2: event data port Internet.entryCl -> client1.cl_p;
28.    annex Behavior_specification { ... }
29. end client_server_arch;
```

Code Snippet 8. *Client-Server description in AADL* – adapted from (Saidane and Guelfi, 2013).

AADL does not offer first-class connectors and defines interfaces, through which the communication between components occurs. Component interfaces consist of (Feiler et al., 2006): data ports, event data ports, event ports, synchronous and explicit access communication mechanisms.

2.9 π-ADL

π-ADL (Oquendo, 2004) is a formal Architecture Description Language designed with a focus on the dynamic perspective of a system (the system's runtime behavior) and successfully addresses the issue of the description of dynamic architectures. It considers both components and connectors first-class entities of the language. Each of these architectural elements is defined in two parts. First the connections are declared. Their role is to ensure communication channels between the interacting elements and they are considered as ports for component declarations and as roles for connector declarations. Next, the behavior of each element is described, as can be seen in the code below, with the use of simple statements. Finally, the architecture is created with the declaration of the instances and their interconnections.

It supports both a textual and a graphical representation. Furthermore, there has been developed a software tool (Cavalcante et al., 2015) for the generation of the executable programming code in the GO (Donovan and Kernighan, 2016) programming language from π-ADL architectural descriptions. Finally, it provides the constructs needed for the successful expression of dynamic and mobile architectures and, as evaluated in (Minora et al., 2012), it is able to support (though the use of tools or other languages like π-AAL (Mateescu and Oquendo, 2006) might be required) both foreseen and unforeseen dynamic reconfigurations.

π-ADL Architectural Description

```
1. component Client is abstraction() {
2.     connection call is out(Integer)
3.     connection wait is in(Integer)

4.     protocol is {
5.           (via call send Integer |
6.            via wait receive Integer)*      }

7.     behavior is {
8.           i is location[Integer]
9.           choose {
```

```
10.                 via call send i
11.                 behavior()
12.           } or {
13.                 storeValue is function(v : Integer) {
14.                       i = v               }
15.                 via wait receive r : Integer
16.                 storeValue(r)
17.                 behavior()
18.           }     }
19. }

20. component Server is abstraction() {
21.    connection request is in(Integer)
22.    connection reply is out(Integer)

23.           [//protocol declarations]
24.           [//behavior specification] }

25. connector Link is abstraction() {
26.    connection fromClient is in(Integer)
27.    connection toServer is out(Integer)
28.    connection fromServer is in(Integer)
29.    connection toClient is out(Integer)

30.           [//protocol declarations]
31.           [//behavior specification]   }

32. architecture ClientServer is abstraction() {
33.    behavior is {
34.           compose {
35.                 c is Client()
36.                 and l is Link()
37.                 and s is Server()
38.           } where {
39.                 c::call unifies l::fromClient
40.                 l::toServer unifies s::request
41.                 s::reply unifies l::fromServer
42.                 l::toClient unifies c::wait
43.           }     }
44. }
```

Code Snippet 9. *Client-Server description in π-ADL* – adapted from (Cavalcante et al., 2014).

2.10 PADL

PADL (Bonta, 2008) is a process algebraic Architecture Description Language with high expressiveness and analyzability. The architectural descriptions are expressed through architectural types (in terms of components and connectors). An architectural type is defined by its behavior (in the form of a sequence of behavioral equations) and its interactions (input/output, synchronous/asynchronous/etc.). As shown in the example implementation of a Client-Server below, the final step in the definition of an architectural type is the declaration of the architectural

topology through the expression of the instances of the previously declared architectural types and their interconnections/attachments. PADL allows for both textual and graphical representation of the architectural description.

The language is, also, integrated in TwoTowers (TwoTowers 5.1 2009), an open-source software tool for the functional verification, security analysis, and performance evaluation of software systems modelled in the ADL Æmilia (TwoTowers 5.1 2009).

PADL2Java (Bonta and Bernardo, 2009) is a software tool 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. A limitation can be considered the fact that the connector classes are predefined, while the port classes (the other architectural element for the establishment of the connection) used, are generated during the PADL translation of the architectural specification.

PADL Architectural Description

```
1. archi_type ClientServer

2.     archi_elem_types

3.           elem_type ClientT
4.                 behavior      Client = send_request.receive_reply.Client
5.                 interactions  output send request
6.                               input receive reply

7.           elem_type ServerT
8.                 behavior Server = receive_request.process_request.send_reply.Server
9.                 interactions  input receive request
10.                              output send reply

11.    archi_topology
12.          archi_elem instances   C : ClientT
13.                                 S : ServerT

14.          archi_interactions
15.          archi_attachments   from C.send_request to S.receive_request
16.                              from S.send_reply to C.receive_reply

17. end
```

Code Snippet 10. *Client-Server description in PADL* – adapted from (Bernardo & Franze, 2002).

2.11 Informal Languages

There is, also, a wide number of modelling languages which offer informal ways for the description of software architectures. The formalities that are met in architecture description languages are omitted in these languages, thus making them more user-friendly and widely applied. Next, a small

portion of them is presented, consisting of languages that have become popular over the years among the software engineering community.

2.11.1 UML

The Unified Modeling Language (UML) (Seidl et al., 2015) is a general-purpose modeling language that has grown in popularity over the last decades and has become one of the most widely used languages in the software engineering community. As seen in (Malavolta et al., 2012), it is preferred by the majority of practitioners for the description of software architectures, over the formal architecture description languages examined in the previous section (Ozkaya, 2014). It defines two views to model different aspects of the system. The *static* view is used to represent the static structure of the system and consists of various diagrams, such as *class* or *component diagrams*. The *dynamic* view is used to represent the behavior of a system during run-time and consists of a different set of diagrams, such as *activity* or *sequence diagrams*.

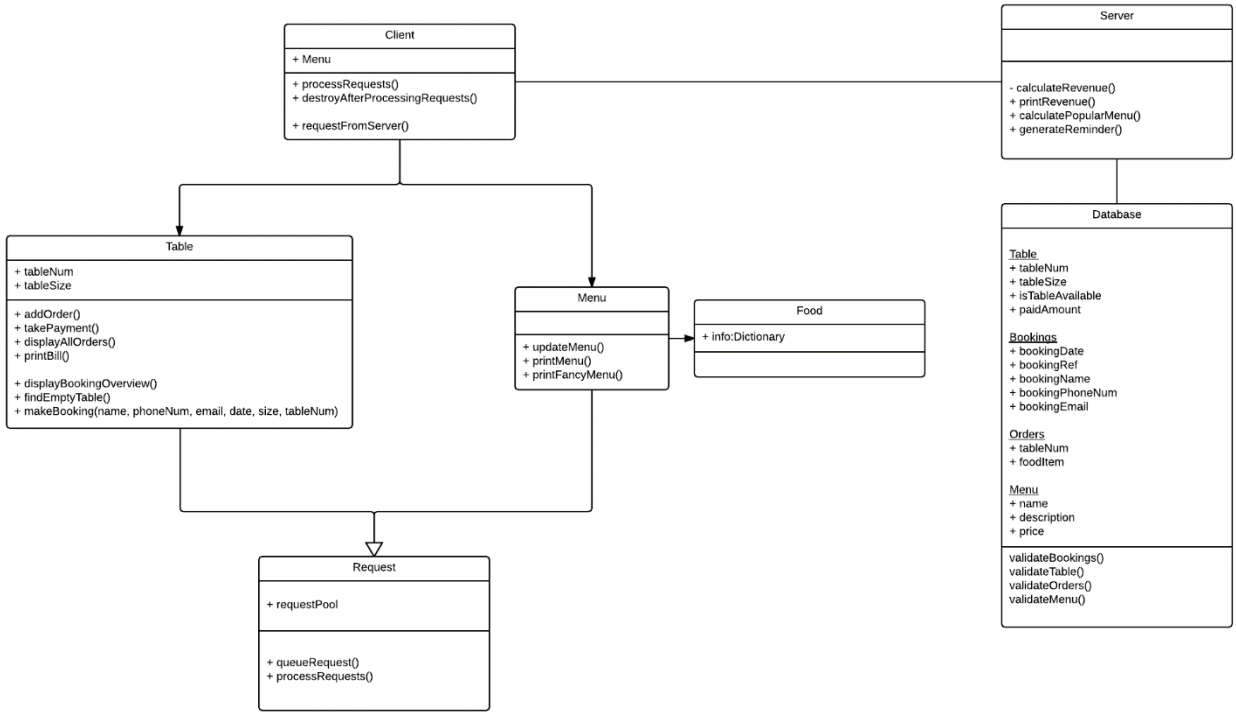


Fig. 7. *Client-Server using the Class UML Diagram* – reprinted from (Software Engineering 2019).

Components can be specified in a *component diagram* in UML. A graph of interconnected components represents the architecture of the system. The interfaces used for the communication are divided in two types; *provided* - services that the component provides, and *required* - services that the component requires from its environment. Connectors are not defined as first-class entities

in UML and the interactions between components are modeled as simple communication channels between their ports, thus making hard to specify complex communication mechanisms (Ozkaya, 2014).

2.11.2 ComponentJ

ComponentJ (Seco and Caires, 2002) is a Java-like programming language oriented to component-based programming and with a focus on the dynamic reconfiguration and evolution of software components. It does not consider connectors as first-class elements of the language and defines three types of first-class entities (Seco et al., 2008): *objects*, *components* and *configurators*. Objects constitute the central computing entity and are described through a set of *methods* – referred to as *services*. Components are used to specify the internal structure and behavior of objects. They declare a set of *ports*, which can be of two types; *required* ports – where the component awaits the result of external services, and *provided* ports – where a set of methods must be implemented in order to be available to other components.

ComponentJ

```
1. component Client {
2.     provides IClient c;
3.     requires IServer s;

4.     methods m {
5.           //method implementation
6.     }
7.     plug m into c;  }

8. component Server {
9.     provides IServer s;
10.    requires IClient c;

11.    methods q {
12.          //method implementation
13.    }
14.    plug q into s;  }

15. ClientServer = compose (
16.          uses c = Client;
17.          uses s = Server;

18.          plug c.c into s.c;
19.          plug s.s into c.s;
20.        );
```

Code Snippet 11. *Client-Server implementation in ComponentJ*.

Configurators are considered to be the basic building blocks in ComponentJ. They describe how existing components are connected, they can define new components and they can be used to modify existing objects. An example of a composition operation can be seen in the code above,

where the configurator ClientServer uses the two previously defined components and connects them appropriately for the creation of a new component.

An advantage of ComponentJ, is the feature of dynamic construction and runtime modification of the structure and behavior of the architectural elements. This results in good support for the description of reconfigurations that occur during run-time.

2.11.3 ArchJava

ArchJava (Aldrich, Chambers, and Notkin, 2002a) is built as an extension to and integrated in the Java programming language. Since often implementation is decoupled from architecture, ArchJava tries to address this problem by providing the means to describe architectural features inside the implementation. In this way it aims at two things. First, the *conformance* of the implementation to the initial architectural constraints defined and, second, to enforce *communication integrity* - i.e. ensuring that components can communicate only with the components they are connected in the defined architecture (Aldrich et al., 2002b).

ArchJava

```
1. public component class Client {
2.     public port c {
3.             provides void sendRequest(String request);
4.             requires Request nextRequest();
5.     }
6.     void processAnswer(String answer) {
7.             nextRequest(answer);
8.     }
9.     void sendRequest(String request) {
10.            ...
11.    }
12.    ...
13. }

14. public component class Server {
15.    //server declarations
16. }

17. public component class ClientServer {
18.    private final Server server = ...;
19.    private final Client client = ...;

20.    connect client.c, Server.s;

21.    public static void main(String args[]) {
22.            new ClientServer();
23.    }
24.    ...
25. }
```

Code Snippet 12. *Client-Server implementation in ArchJava*.

Components are special kind of objects in ArchJava and their communication is ensured through the definition of *ports*, which represent the interaction points of a given component. Ports can declare three sets of methods (Aldrich et al., 2002b): *requires* (provided methods by other components), *provides* (implemented methods by the component and available to other components) and *broadcasts* (similar to required methods, except that they can be connected to any number of implementations and must return void).

In ArchJava, at first, connectors were not first-class elements, but instead the primitive *connect* was used to connect two or more ports, by binding a required method to the appropriate provided one, as shown in the example code above. An extension was created (Aldrich et al., 2003), in order to provide connector abstractions. With the introduction of connectors, ArchJava provides the means for decoupling connection code from application logic, enhancing its reusability.

2.11.4 SysML

SysML (Friedenthal et al., 2014) is a general-purpose architecture modeling language for systems engineering applications (SysML 2018). It is created as an extension to the UML and introduces new fixtures, such as *requirements* and *parametric diagrams* (Ozkaya, 2014). Components in SysML are expressed through *blocks* which are connected to each other with *ports*. The language also supports behavioral specification, but connectors are not first-class entities (since interactions are defined with simple attachments), which makes harder the specification of complex communication mechanisms.

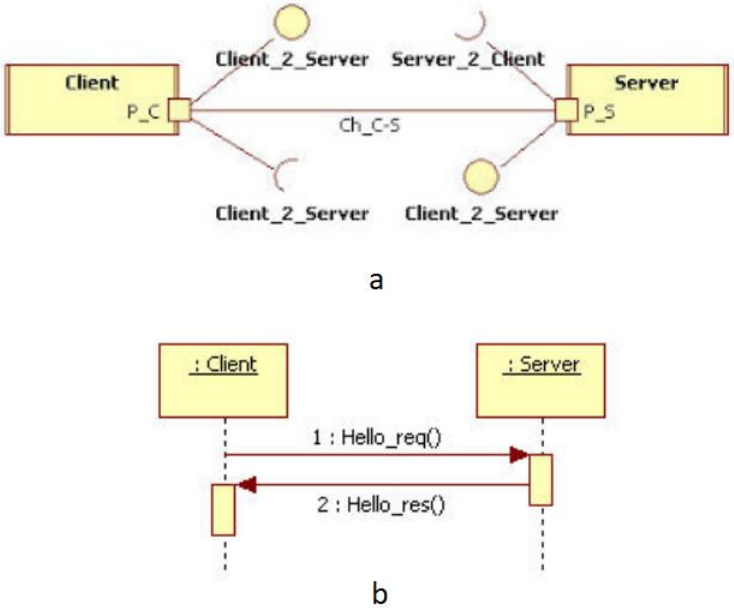


Fig. 8. *A simple (a) structural and (b) behavioral modeling in SysML* – reprinted from (SCIETEC 2010).

2.11.5 SoaML

SoaML (Service-oriented architecture Modeling Language) (SoaML 2019) is another extension to the UML and focuses on Service-Oriented Architectures (SOA) (Erl, 2016). It provides the necessary elements for the modeling of services within a service-oriented architecture. Components can be represented as *Participants* that interact with each other by using *Services* (provided by or requested by). An architecture can be expressed with the use of a UML Collaboration. Additionally, a software tool has been implemented, in the form of an Eclipse plugin, named Eclipse SoaML Tool (Delgado and Gonzalez, 2014). A screenshot of this tool is shown in the figure below.

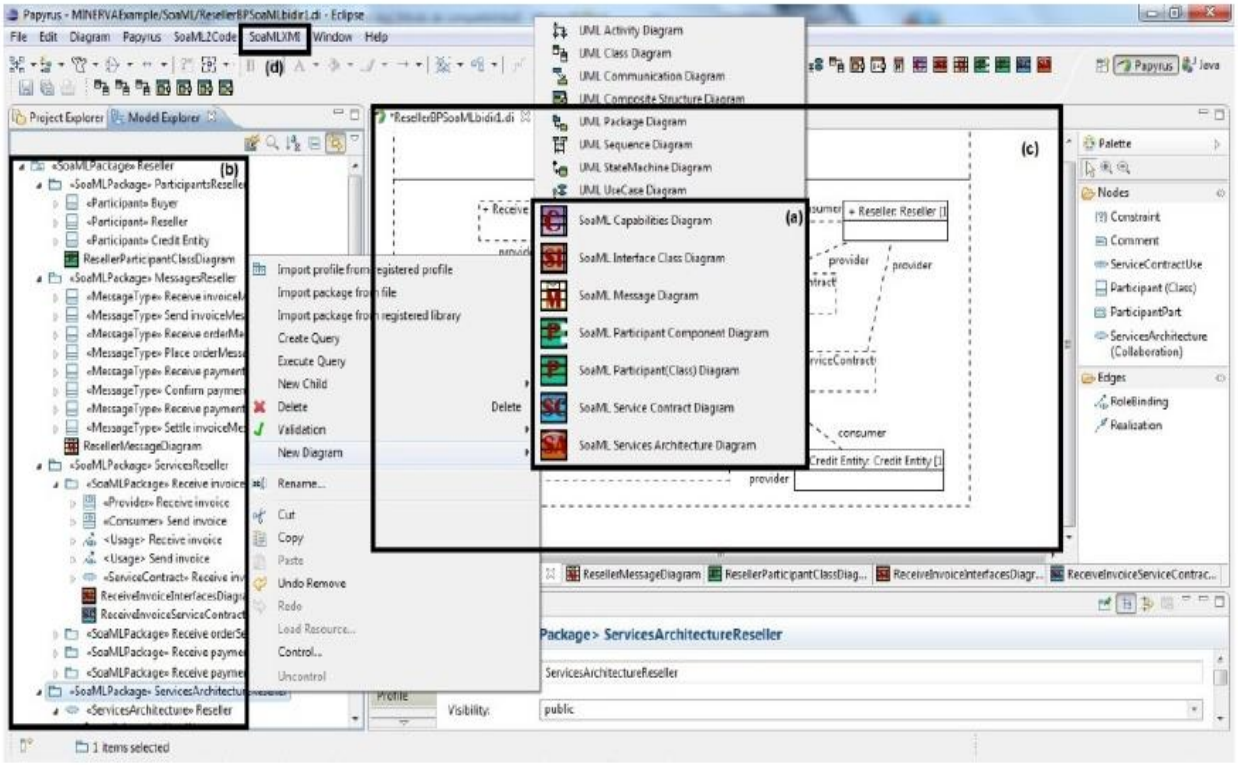


Fig. 9. *Eclipse SoaML Tool* – reprinted from (Delgado and Gonzalez, 2014).

2.12 Conclusion

In the table below, is presented aggregated data regarding the languages examined in this chapter. The indication DNA stands for "does not apply". It is used for the informal languages in the columns Generation and Dynamic Reconfiguration. For the first one, the classification between first and second generation includes only formal languages. For the latter, their capabilities regarding dynamic reconfigurations are out of the scope of this thesis, since the focus of this thesis

is the creation of a formal language with support for dynamic reconfiguration and formal behavior definition.

Language	Generation	High-level Components	Connectors as first-class entities	Formal behavior specification	Dynamic reconfiguration
Darwin	*1st*	✓	**X**	*FSP*	*harder to achieve due to lack of connectors*
Wright	*1st*	✓	✓	*CSP*	*use of extensions, limited to foreseen reconfigurations*
Rapide	*1st*	✓	**X**	*event patterns*	*mostly foreseen reconfigurations*
ACME	*1st*	✓	✓	**X**	*use of external scripts, limited to foreseen reconfigurations*
Koala	*2nd*	✓	**X**	**X**	*harder to achieve due to lack of connectors*
xADL	*2nd*	✓	✓	**X**	*harder to analyze due to lack of formal behavior specification*
AADL	*2nd*	*built-in low-level components*	**X**	*automata*	*harder to achieve due to lack of connectors*
π-ADL	*2nd*	✓	✓	*π-calculus*	*use of extensions, mostly for foreseen reconfigurations*
PADL	*2nd*	✓	**X**	**X**	*harder to achieve due to lack of connectors*
UML	*d.n.a.*	✓	**X**	*state machine diagrams*	*d.n.a.*
ComponentJ	*d.n.a.*	✓	**X**	**X**	*d.n.a.*
ArchJava	*d.n.a.*	✓	✓	**X**	*d.n.a.*
SysML	*d.n.a.*	✓	**X**	*state machine diagrams*	*d.n.a.*
SoaML	*d.n.a.*	✓	**X**	*state machine diagrams*	*d.n.a.*

As presented through the chapter, each of these languages focuses on different aspects when it comes to the description of the architecture of software systems – e.g. Wright in communication mechanisms, Rapide in simulation etc. Though there is a plurality of languages, there are still issues concerning the capability of an architecture description language to capture the dynamism in current software systems, their high degree (in most cases) of formality and the lack of tool support. These three issues constitute the main reasons that led me to the decision to create a new language:

- the dynamism and the need for dynamic reconfiguration in software systems, which has increased in the last decade (e.g. IoT, microservices).

- the lack of an ADL which can capture and express these needs, by providing syntax and language constructs that will be familiar and, relatively, easy to learn and use for practitioners too.

- to provide a set of tools (e.g. an editor, translator, etc.) for architects and stakeholders to ease the use of the language.

Chapter 3

jADL

3.1 Introduction

In the previous chapters have been presented a number of existing architecture description languages, as well as their advantages and disadvantages. Despite their large number (more than 100), research, as the ones conducted for example in (Ozkaya, 2014; Malavolta et al., 2012; Ozkaya and Kloukinas, 2013), indicates that there are still issues regarding the use of such languages. One of the main reasons identified is the need for the adequate expression of the dynamism (the foreseen and unforeseen dynamic reconfigurations) in software systems. Another important reason is the high degree of formality met in most architectural languages, which discourages practitioners from using them. Additionally, a secondary problem could be that these languages, quite often, are not much supported from tools, which could ease their use.

jADL is a formal architecture description language, created in this thesis, for the expression (both structural and behavioral) of static, dynamic and mobile software architectures. It provides the flexibility and expressiveness required in order to express the dynamic reconfigurations of software intensive systems. jADL is based on a Milner's version of an asynchronous process pi-calculus (Milner, 1999), called applied pi-calculus, for studying concurrency and process interaction. It defines a relatively simple syntax and language constructs which can be familiar and easy to learn for developers, since they resemble widely used programming languages. This is done in order to help in promoting the use of architectural languages in software implementation industry processes, which is still very limited. I believe that this can lead to improved and automated ways of creating implementations which are consistent with the initial architecture designed. Additionally, the language is accompanied by tools (e.g. editor) to ease its use, which are presented in chapter 5. Finally, jADL syntax is influenced by good practices of other architectural languages, like ACME and π-ADL presented in the previous chapter.

In the rest of this section the syntax of jADL is presented and explained using the Extended Backus–Naur form (EBNF). The architectural elements of the language are presented in detail; components, connectors, interfaces, communication traits. Additionally, the rest of the language constructs are presented, such as statements, variables and data types supported by jADL. Also, the graphical notation of jADL is shown.

The practical use of the language and its various constructs is illustrated through the description in jADL of a simple Client-Server architecture and a Message Bus Architectural Pattern, through which the capabilities of the language for dynamic reconfiguration of a system are shown.

3.2 jADL Syntax

In jADL, the basic building blocks and first-class architectural entities are components and connectors. Additionally, interfaces and communication traits are considered first-class entities too. jADL allows the creation of both primitive and composite components and connectors. The syntax regarding these elements is presented below, as well as the syntax and the characteristics of the rest of the language constructs.

3.2.1 Components

Following the component-and-connector paradigm, components in jADL represent the computational and data store elements (locus of computation). In order to communicate with their environment, they declare a number of ports, which constitute their single point of interaction. The communication between two components is strictly through the use of connectors (see next subsection). The behavior of each component is defined through the configuration of their *provides* ports (see 3.2.3) and the definition of internal methods which can be accessed only by the given component. Next, is shown the syntax regarding a component declaration in jADL using EBNF.

<*component_declaration*> ::= "**component**" <*id*> "**{**" {
[<*port_declaration*>**]***
[<*trait_aggregation*>**]***
[<*config_statement*>**]***
[<*internal_method*>**]***
} "**}**"

Component declaration in jADL.

3.2.2 Connectors

Connectors in jADL model the communication between the various components and their environment (locus of communication) and only through them can two components communicate. Respectively to a component's ports, they declare a number of roles which are attached to ports so that the communication is ensured. The behavior of each connector is defined through the configuration of their *provides* roles (see next subsections) and their internal methods which are accessed only by the given connectors. Below, is shown the syntax regarding a connector declaration using EBNF.

<*connector_declaration*> ::= "**connector**" <*id*> "**{**" {

[<*role_declaration*>**]***

[<*trait_aggregation*>**]***

[<*config_statement*>**]***

[<*internal_method*>**]***

} "**}**"

Connector declaration 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, as explained in the sections that follow.

3.2.3 Ports & Roles

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 entities. 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.

The previously described characteristics 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). Below, is shown the syntax regarding the declaration of a port and a role in jADL using EBNF.

<*port_declaration*> ::=

("**provides**" | "**requires**") ["**synchronized**"] "**port**" <*interface*> <*id*> "**;**"

<*role_declaration*> ::=

("**provides**" | "**requires**") ["**synchronized**"] "**role**" <*interface*> <*id*> "**;**"

A port and a role declaration in jADL.

jADL views ports and roles as channel of communications. These channels have a data type and interfaces are a flexible and 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 behavior of an architectural element in jADL.

3.2.3.1 Kind

From the declaration of ports and roles above, the keywords *provides* and *requires* are used to declare their *kind*. Every port or role must have a kind. The 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 that will request it. On the other hand, the kind *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 an error is generated.

3.2.3.2 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 10 below provides cases where attachments have more than two architectural elements involved.

While the attachment in 10.a is successful, the one in 10.b 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.

3.2.3.3 Synchronicity

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.

In a 1-N communication, additional problems than those mentioned in the previous subsection might appear when each of the architectural elements participating is part of a different thread. In figure 11 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.

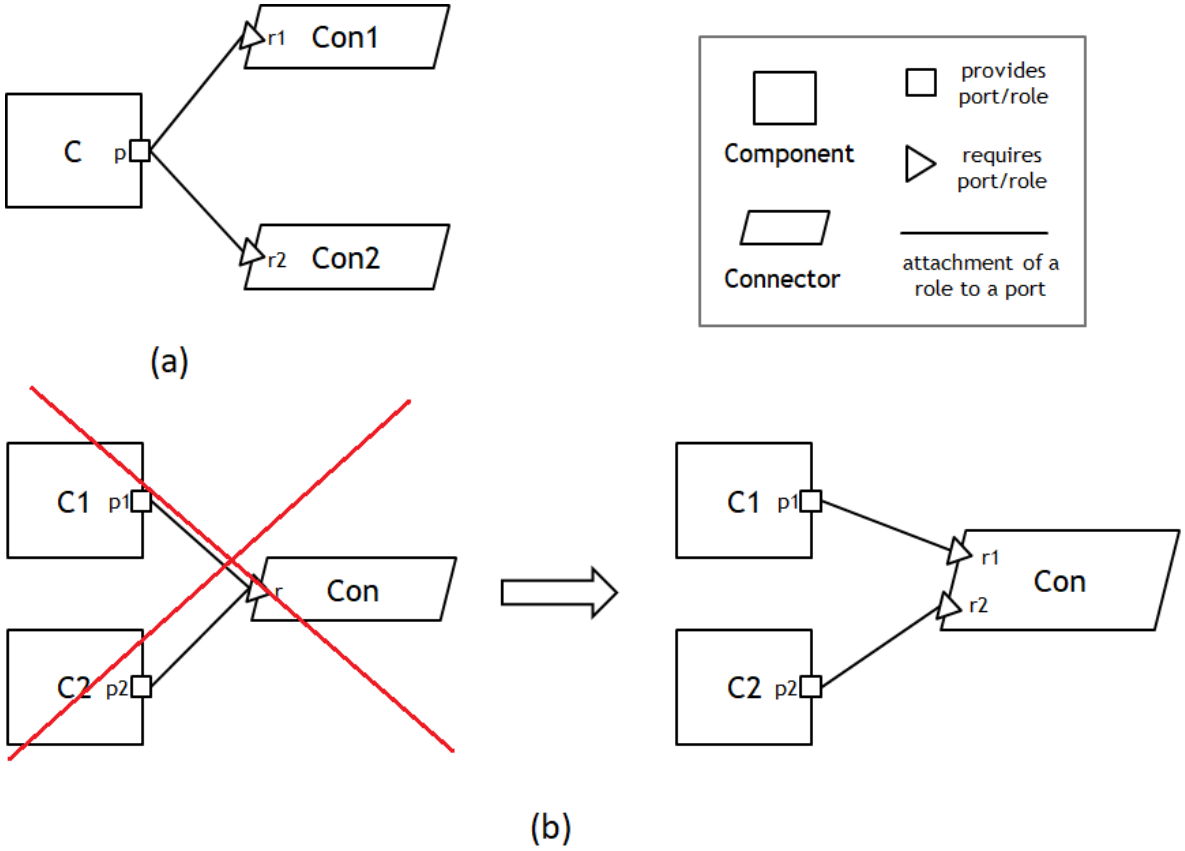


Fig. 10. *Different cases of 1-N communication in jADL.*

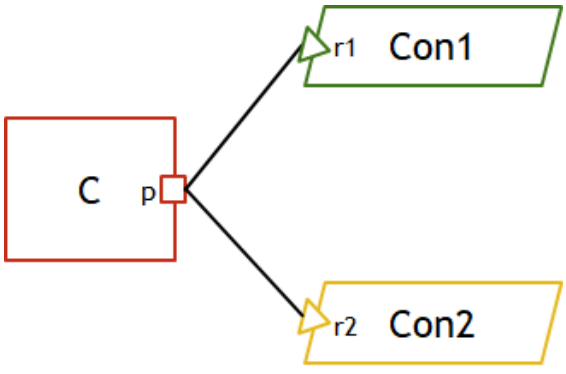


Fig. 11. *1-N communication of three different threads in jADL.*

For example, 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 a part in the description where this size will be available to other elements through its port p and the configuration will be:

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

In connector's *Con1* definition (and respectively to *Con2*) there will be a part in the description where this size will be requested and will look like:

```
// ...
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.

3.2.4 Interfaces

Interfaces are used to define the shape of communication and the behavior 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 (Milner, 1999), 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 the 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], Queue[T], etc. The second type system defines the port and component types.

<*interface_declaration*> ::= "**interface**" <*id*> "**{**"
[<*service_declaration*>]*****
"**}**"

An interface declaration in jADL.

3.2.5 Behavior Specification

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 is defined under the form of statements. All ports and roles that are declared as *provides* must be configured using this statement.

The config statement can be used at runtime as well, for dynamically assigning a behavior. This means that we can reconfigure a port or role behavior and this is one of the mechanisms of jADL to support reconfigurability of the architectural elements during runtime. An example of runtime usage is when a port or role is (re)configured in a trait aggregation declaration (see following subsection).

<*config_declaration*> ::= "**config**" <*id*> "**as**" "**{**"
[<*assigned_interface_service(s)_definition*>]*****
"**}**"

An interface declaration in jADL.

3.2.6 Communication Traits

Communication trait is a complex communication structure in jADL that can group together ports and roles and it is considered a first-class entity. The use of this construct is comprised of two parts; first the declaration of the communication trait, which can be done both inside and outside of another architectural element (component and connector in this case). Second, the aggregation of this trait, which must be done inside of the architectural element that will be using it.

In each trait a varying number of ports or roles can be declared, as long as the type of the architectural elements remains the same - i.e. *each trait can hold only ports or only roles*. By encapsulating the ports and roles in a separate structure and by using the second form of the attach statement (as described in the subsection 3.2.7.1) we provide the possibility for dynamic instantiation of ports and roles at run-time. Without this structure the reconfiguration of an

architectural element in an architectural description would require a series of detach statements, the creation of a new element and a series of attach statements. Communication traits allow to perform such operations with the use of a simple trait aggregation statement and the benefits of using them can be seen (especially) in the case study of the Message Bus architectural pattern described in the final section of this chapter.

<*trait_declaration*> ::=

"**trait**" <*id*> "**{**" [<*port_declaration*>* | <*role_declaration*>*] "**}**"

<*trait_aggregation*> ::=

"**trait**" <*id*> "**aggregate**" <*trait_declaration*> "**{**" <*config_statement*>* "**}**"

A trait declaration and usage in jADL.

Another useful feature of this construct is that the attachments in jADL when two traits are attached are made on the background and there is no need for the explicit declaration of the names of the ports and roles. The introduction of this complex structure enhances significantly the flexibility and the expressiveness of jADL especially when it comes to describing dynamic architectures and dealing with foreseen (that are known in design time) and, mostly, unforeseen (that cannot be known during design time) dynamic reconfiguration problems.

3.2.7 jADL Statements

jADL defines a number of statements in order to provide the means to software architects and various stakeholders to formally, yet with a more practical way, describe a given architecture. In the following subsections, these statements are presented. They concern two aspects of the architecture definition in jADL; the connections (attach/detach, bind) and the behavior definition (select, delay, process).

3.2.7.1 Attach / Detach

The *attach* statement is used for the unification of ports and roles and the creation of a communication channel so that the control and data flow is ensured between the architectural elements. It can accept either a (<*role*>, <*port*>) pair or a (<*trait*>, <*trait*>) pair of arguments. In the first case, as described in section 3.2.3, the checks for the successful unification are: the compatibility of interfaces, the opposing directions of the port and role and the same synchronicity. In the second case, the compiler checks, additionally, the ports and roles of the two traits provided as arguments and either a communication channel is established between the two elements or an error is returned. First, it checks that the one trait consists of ports and the other of roles. The second check performed is whether the two traits consist of the same number of roles and ports of

opposing kinds and whether each pair of opposing kinds has the same interfaces. If both checks are successful, then the attachment is established and the ports and roles are unified. Otherwise, an error will occur and no communication channel will be created for these elements.

The *detach* statement is the "opposite" of the attach statement and it is used to destroy the communication channel that was established between two architectural elements. Similarly to *attach*, it can accept either a (<*role*>, <*port*>) pair or a (<*trait*>, <*trait*>) pair of arguments. These two statements can be used both when defining an architecture and at run-time for dynamically reconfiguring the architecture. The application and use of these statements when it comes to dynamic reconfiguration is shown in the case studies in the final sections of this chapter.

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

| "**(**" <*trait_id*>, <*trait_id*> "**)**"] "**;**"

<*detach_statement*> ::= "**detach**" ["**(**" <*role_id*>, <*port_id*> "**)**" "**;**"

| "**(**" <*trait_id*>, <*trait_id*> "**)**"] "**;**"

Attach and Detach statements in jADL.

3.2.7.2 Delay

The *delay* statement is used to block the execution of an operation within a system for a given period of time. It has two forms for the definition of this period under which it can be declared; by defining an integer value in milliseconds or by using an expression that will stop it as long as it evaluates to false. A simple example where its use is shown, is in the load balancer described in 3.4.1 where the system waits for 10ms until it tries to forward the request, if all servers are busy.

<*delay_statement*> ::= "**delay**" <*int*> "**;**"

| "**delay_until**" <*expression*>* "**;**"

Delay statement in jADL.

3.2.7.3 Select

The *select* statement is used upon the definition of the behavior of an element in its *config* statement. The series of statements to be executed, are chosen from the block that the *when* expression evaluates to true. A simple example of its usage can be seen in the load balancer description presented, where the appropriate server to send the request is being chosen using this statement.

<*select_statement*> ::= "**select**"

["**when**" <*booleanGuard*> "**=>**"] "**{**"

<*statement*>***** "**}**"

["**or**"

["**when**" <*booleanGuard*> "**=>**"] "**{**"

<*statement*>***** "**}**"]*

"**end;**"

Select statement in jADL.

3.2.7.4 Process

The *process* statement is defined using the keyword process. It is used in an architectural description to express that the architectural element containing it continues to operate "*as it is*". For example, in the use case presented in the next sections of a self-adapting load balancing system, as long as the load remains between 20% and 80% the system continues to operate without the addition or removal of servers.

<*process_statement*> ::= "**process**" "**;**"

Process statement in jADL.

3.2.7.5 Bind

The *bind* statement concerns a special case regarding the attachments in jADL. 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 of the usage of bind can be seen in the next figure.

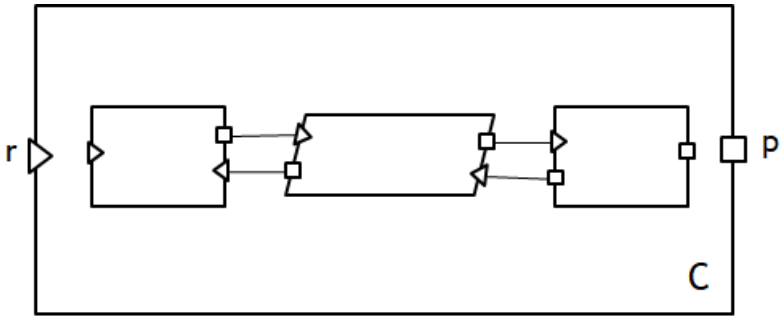


Fig. 12. *A composite component in jADL.*

The *requires* port *r* of the composite component *C* above needs to be "connected" with the *requires* role of an inner component and the provides port *p* respectively. The *attach* statement does not

allow of port to be connected to another port (the same stands for roles too as described above). So, for the definition of the outer ports and roles of composite architectural elements, the *bind* statement is used, since it allows such connections. The only restriction concerning this statement is that *the arguments in the bind statement must be of the same kind*.

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

Bind statement in jADL.

3.2.8 "Simple" Statements

Apart from the special statements presented until now, in jADL standard control flow and loop statements, existing in most programming languages, can be used. We chose a Java-like definition for familiarity to practitioners reasons and these statements include: *if* statement, *for* and *while* loop statements and *assign* statement. These constructs are, mainly, used for the definition of the behavior of an architectural element, inside the *config* statement of a port or role.

3.2.9 Variables and Data Types

jADL defines attributes and local variables as other architecture description languages, which are used to describe constraints over quality attributes architecting. It, also, defines some primitive data types like Integer, String, etc., as well as parameterized by type data structures like List, Hashmap, etc. We allow parametric polymorphism over interfaces and types as well. We employ a typing mechanism to establish shape conformance between ports and roles interfaces.

According to type theory the type inference algorithm exists accompanied with appropriate substitution of variables as a result of algorithm application. Unification between a port and an attached role shapes' is performed and compiler produces possible substitution or otherwise reports typing error. They are expressed with the use of simple statements (as shown in the example of a *hashmap* declaration below), so that they can be both easy to use and familiar to practitioners. For jADL they are components with two ports (one for receiving and one for delivering information) that have predefined services available.

hashmap<*CommTrait, msgs*> *messages* = ***new hashmap***<*CommTrait, msgs*>***();***

3.3 jADL Graphical Representation

jADL as a regular architectural language has two parts: textual representation of an architectural script, as well as a graphical part – graphical representation of the architecture. Graphical representations can ease the communication between the various stakeholders and can provide an easier for a human to grasp overview of the system's architecture. In the figure below the graphical notation of each architectural element in jADL can be seen.

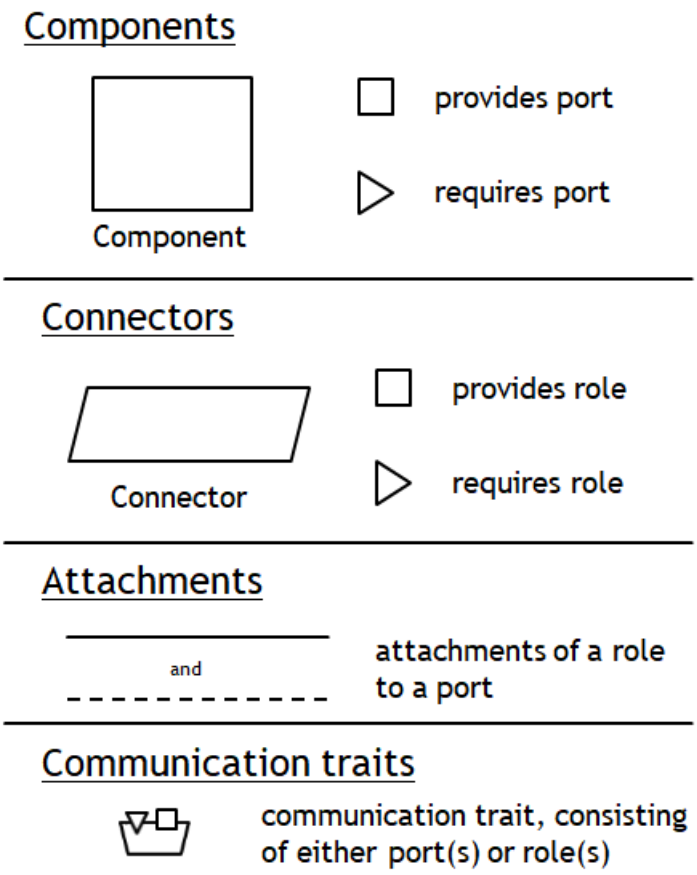


Fig. 13. *Graphical notations in jADL.*

In most of the graphical representations of architectural descriptions of systems that follow in this thesis can be observed that connectors are represented as straight lines (like the ones used for attachments in the figure above). This is done only for the simplification of the graphical part. It should not be considered that the components are connected to each other or confused with other languages that consider connectors as simple communication channels. Behind this line still "lies" a defined connector that connects the two components.

3.4 Client-Server Architecture

In this section we present how a Load Balancer Architectural Pattern (LBAP) (Erl et al., 2013) 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 for simplicity 3 servers are used, but in the next subsection are extended 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.

3.4.1 Load Balancer Architectural Pattern

The client (C) shown in fig. 14 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. 14) defines two kinds of connectors; a secure connector for the client - load balancer communication and a simple one for the load balancer - server communication.

jADL Client Description

```
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 13. *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 it waits for the response at its *provides* port *wait*. When a *provides* port (or role) is defined, jADL specifies configurations that describe its behavior. 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 modelling errors of an application which could occur during run-time.

Furthermore, the first line (code snippet 13) 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.).

jADL Connectors Description

```
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 14. *Connectors description in jADL*.

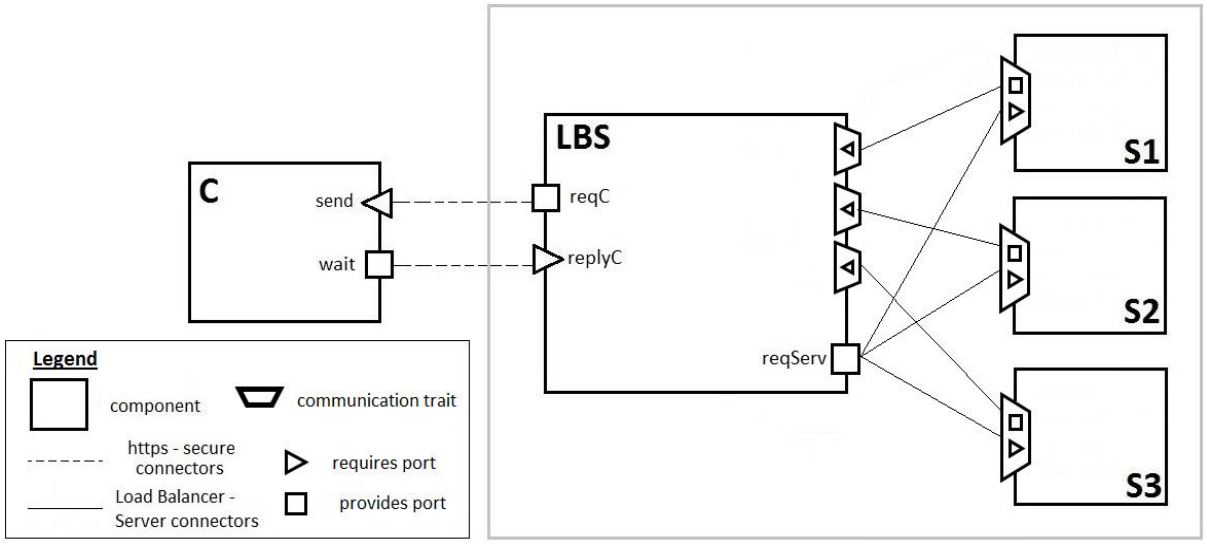


Fig. 14. *Architecture of a simple load balancing system*.

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.

jADL Server Description

```
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.
16.   trait CTraitS aggregate ServCommTrait {
17.     config req as {
18.           service void procRequest (Type data) {
19.                 curLoad=curLoad+1;
20.                 Type resp;
21.                 reply.aResponse(resp);
22.                 curLoad=curLoad-1;
23.           }
24.     }
25.   }
26.
27.   CTraitS comServ = new CTraitS();
28.
29.   while (true) {
30.           select
31.           when (curLoad < maxNum) => {
32.                 process; }
33.           or
34.           when (curLoad == maxNum) => {
35.                 delay_until curLoad == (maxNum-100); }
36.           end;
37.   }
38. }
```

Code Snippet 15. *Server description in jADL*.

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 one 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 15) 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. 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 16, or by explicit declaration – code snippet 15).

jADL Load Balancer Description

```
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.       attribute int m1 = comLB1.rServ.loading();
21.       attribute int m2 = comLB2.rServ.loading();
22.       attribute int m3 = comLB3.rServ.loading();

23.        select
24.          when (m1≤m2 && m1≤m3) => {
25.           comLB1.rServ.procRequest(data); }
26.        or
27.          when (m2≤m1 && m2≤m3) => {
28.           comLB2.rServ.procRequest(data); }
29.        or
30.         when (m3≤m2 && m3≤m1) => {
31.           comLB3.rServ.procRequest(data); }
32.        or {
```

```
33.         delay 10; }
34.       end;
35.
36.     }
37.   }
38. }
```

Code Snippet 16. *Load Balancer Server description in jADL*.

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. A new port is created, using the communication trait *CTratitLB*, for each of the three servers. Through these ports the requests are forwarded 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.

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 17). jADL assumes a usual situation where the architectural elements are executed in different processes and multiple threads of execution.

jADL Architecture Description

```
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.   }
35. }
```

Code Snippet 17. *Architecture instantiation in jADL*.

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.

3.4.2 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 the next figure.

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.

jADL Dynamic Load Balancer Description

```
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.
14. config reqServ as {
15.    service void procRequest (Type data) {
16.     requests.get(data).rClient.aResponse(data);
17.    }
18.  }

19. trait CTraitLBS aggregate LBCommTraitS { }

20.  CTraitLBS comLBS1 = new CTraitLBS();
21.  CTraitLBS comLBS2 = new CTraitLBS();
22.  CTraitLBS comLBS3 = new CTraitLBS();

23.  servers.put(comLBS1, "active");
24.  servers.put(comLBS1, "active");
25.  servers.put(comLBS1, "active");

26.  config reqC as {
27.   service void procRequest (Type data) {
28.    float avg = abstract finAvgLoad(servers);

29.    trait CTraitLBC aggregate LBCommTraitC { }

30.    CTraitLBC comLBC = new CTraitLBC();
31.    requests.put(data, comLBC);
32.
33.    select
34.        when (avg<20) => {
35.          x = abstract firstActiveServer(servers);
36.          x.rServ.procRequest(data);
37.          if (count(servers.getKey("active"))>1)
38.            servers.put(x, "inactive");  }
39.    or
40.        when (avg>80) => {
41.          trait CTraitLBS aggregate LBCommTraitS { }

42.          CTraitLBS comLBS = new CTraitLBS();

43.          instance sN = new Server();
44.          instance conn = new Conn2();

45.          servers.put(comLBS, "active");
46.          conns.put(comLBS, conn);
47.          attach(conn.cRes, reqServ);
48.          attach(conn.cReq, comLBS.rServ);
```

```
49.           attach(conn.sRes, sN.comServ.reply);
50.           attach(conn.sReq, sN.comServ.req);

51.           comLBS.rServ.procRequest(data);  }

52.       or {
53.           proccess; }
54.       end;
55.       }
56.     }

57.  while (true) {

58.   for (servers srvK : srvV) {
59.    if (srvV=="inactive" && srvK.loading()==0 && count(servers.getKey("active"))>1){

60.         detach(conns.get(srvK).cRes, reqServ);
61.         detach(conns.get(srvK).cReq, srvKey.rServ);

62.         servers.remove(srvK);
63.         conns.remove(srvK);
64.       }
65.   }
66.   delay 25;
67.  }

68. }
```

Code Snippet 18. *Dynamic Load Balancing System description in jADL.*

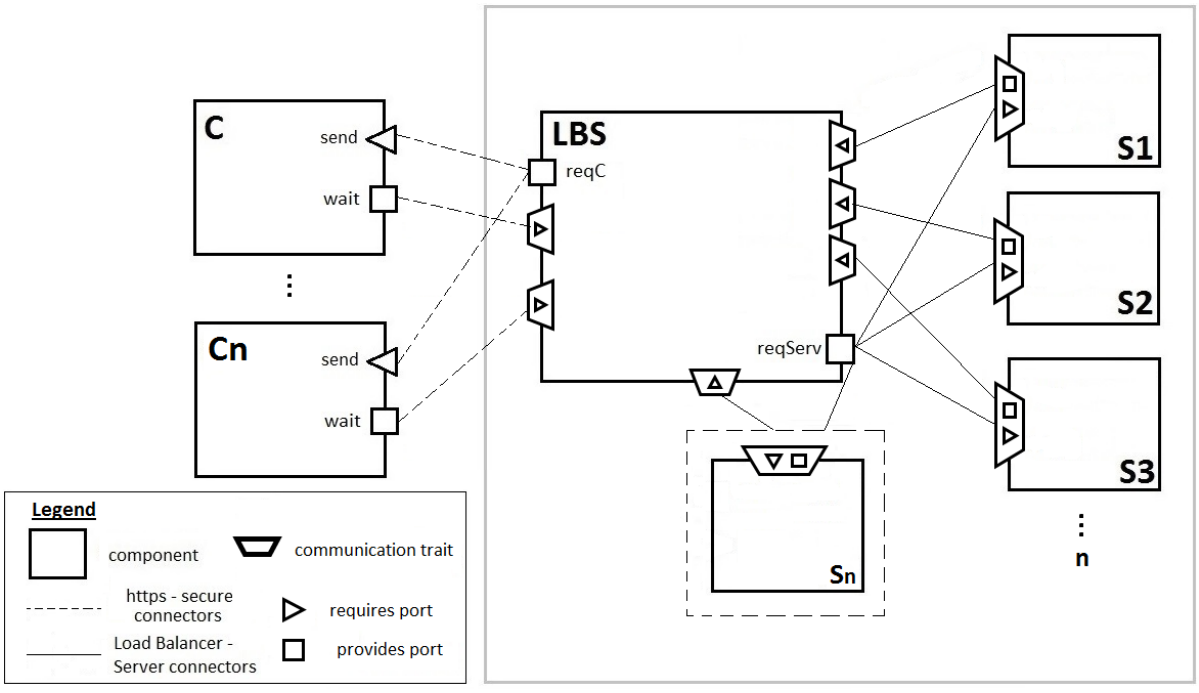


Fig. 15. *Architecture of a self-adapting load balancing system.*

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 18). The keyword *abstract* defines that an algorithm should be implemented there by developers, splitting this way the architecting activities from the implementation/development part. If the average load is below 20% of the allowed 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 18, lines 57-67) 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.

3.5 Message Bus Architectural Pattern

One of the most adopted definitions of an Enterprise Service Bus (ESB) (Keen et al., 2005) is "*a style of integration architecture that allows communication via a common communication bus that consists of a variety of point-to-point connections between providers and users of services*" (Enterprise Service Bus 2013). ESB presents an architectural pattern that outlines the basic set of rules for integrating a varying number of heterogeneous applications together.

It is a widely used concept today since the rapid spread of the internet and the continuously increasing number of services like IoT, cloud computing, etc., require that a lot of different applications can communicate and/or exchange information in a quick, secure and reliable way. The ESB concept defines a pattern that allows different systems to communicate without having any dependencies between them (or even being aware of each other). It provided an adequate response to the need for a different approach than point-to-point integration, which is often hard (or impossible in cases) to manage or evolve over time (highly interdependent modules).

The main advantage of an ESB architecture is the fact that the decoupling between the components that are communicating is increased. They are connected to the bus and not to the actual provider of the service thus eliminating any dependencies between them and lightening the process of addition/removal of components. It constitutes a preferable environment to enforce security since it monitors and mediates all the interactions between the components. Additional advantages and features provided by ESBs can be failover support (e.g. by keeping a cache), load balancing for improved performance, etc.

Here an architectural description of a variation is presented – the Message Bus Architectural Pattern (MBAP). The architecture of the MBAP consists of a connector, which plays the role of the Message Bus, and a varying number (dynamically changing) of components, which play the role of senders and receivers. The architecture of the MBAP can be seen in figure 16.

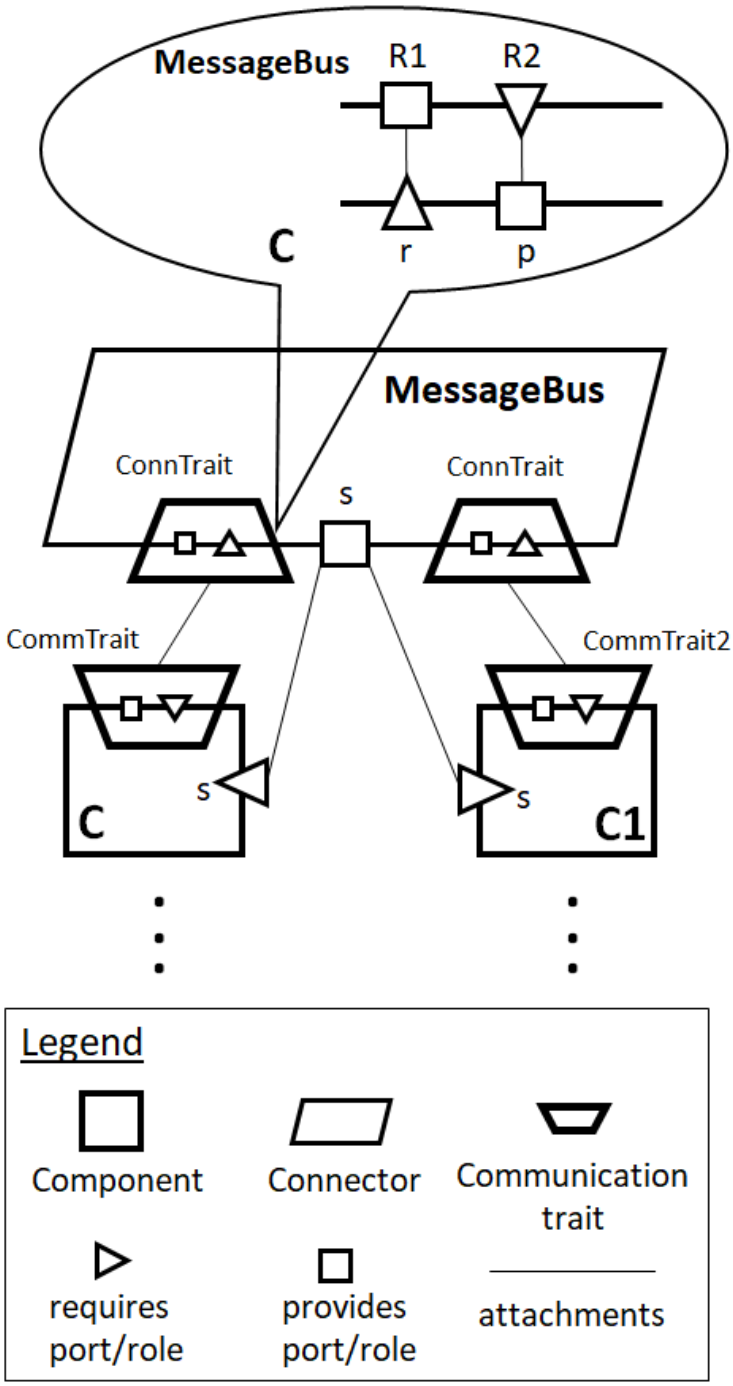


Fig. 16. *Architecture of the Message Bus Architectural Pattern*.

jADL Client Component Description

```
1.  type Message;

2.  // declaration of interfaces
3.  interface ISendMsg {
4.   service void sendMsg (Message msg, CommTrait comT);
5.  }
6.  interface IReceiveMsg {
7.   service void getMsg (Message msg);
8. }
9. interface ISubscribe {
10.  service void subscribeTo (CommTrait comT);
11.  service void unSubscribeFrom (CommTrait comT);
12. }

13. // declaration of a trait
14. trait CommTrait {
15.  provides port IReceiveMsg p;
16.  requires port ISendMsg r;
17. }

18. component C {
19.  requires port ISubscribe s;
20.
21.  trait CTrait aggregate CommTrait {
22.    config p as {
23.      service void getMsg(Message msg){
24.        // proccess the Message
25.        display(msg);
26.      }
27.    }
28.  }
29.
30.  CTrait com1 = new CTrait();

31.  while(true) {
32.    com1.r.sendMsg(com1, "new message");
33.    delay 10;
34.  } }
```

Code Snippet 19. *Client component description in jADL.*

Line 1 in code snippet 19 defines that the architecture is parameterized in terms of the data (messages) exchanged between the components (e.g. XML documents). The keyword *type* used here is an abstraction for the various data types supported in jADL. Next are defined the interfaces assigned to the ports and roles.

The component *C* could be any component of a given application that needs to communicate with its environment by exchanging messages. In its declaration only one port is statically declared – the *requires* port *s*. From this port it can send its request when it needs to subscribe or unsubscribe.

In order to successfully subscribe to the *MessageBus* connector it needs to have the appropriate interfaces at its respected ports. This is achieved through the use of the *CommTrait* trait which is dynamically instantiated at run-time. This trait consists of the required ports (and interfaces) that a component needs to have to subscribe to the *MessageBus*. Once the attachment is successful the component can start sending its messages through its port *com1.r* and receiving messages from the *MessageBus* through its port *com1.p*. Inside the component body the *com1.p* port is configured using the *config* statement. A behavior is assigned regarding the processing of a received message.

At the initialization of the connector there is only one role statically declared – the role named *s*. It is declared as *provides* so that multiple components can be attached to it without encountering any non-determinism problems, since only this kind of ports/roles can handle multiple connections in jADL. The role *s* is assigned the interface *ISubscribe* and is configured as follows.

jADL Connector Description (MBAP)

```
1.  trait ConnTrait {
2.    provides role ISendMsg R1;
3.    requires role IReceiveMsg R2;
4.  }

5.  connector MessageBus {
6.   provides role ISubscribe s;
7.   attribute int maxRoles = 1000;

8.   List<Message> msgs = new List<Message>;
9.   hashmap<CommTrait, msgs> messages = new hashmap<CommTrait, msgs>();
10.  hashmap<CommTrait, ConnTrait> subscribers = new hashmap<CommTrait, ConnTrait>();

11.  trait Comm1 aggreates ConnTrait {
12.    config R1 as {
13.     void sendMsg(Message msg, CommTrait comT){
14.       messages.put(comT, msgs.add(msg));
15.     }
16.    }
17.  }
18.
19.  config s as {
20.    service void subscribeTo (CommTrait comT) {
21.     if (subscribers.size() < maxRoles ){
22.       Comm1 com1 = new Comm1();
23.       attach(com1, comT);
24.       subscribers.put(comT, com1);
25.     }
26.    }

27.    service void unSubscribeFrom (CommTrait comT) {
28.     detach(subscribers.get(comT), comT);
29.     subscribers.remove(comT);
30.     messages.remove(comT);
31.    }
32.   }
```

```
33.   while(true){
34.    for(messages msgKey : msgVal) {
35.     for(subscribers subsKey : subsVal) {
36.      if (subsKey != msgKey) {
37.       subsVal.R2.getMsg(msgVal.get(0));
38.       msgVal.remove(0);
39.      }
40.     }
41.    }
42.    delay 20;
43.   }
44. }
```

Code Snippet 20. *Connector (MBAP) description in jADL*.

There are only two types of requests that are sent to the connector through this role – the first one is from components that request to be attached to the *MessageBus* and the other from components that request to be detached from it. If a new request arrives from a component that needs to be subscribed, a new instance of the connector's trait is instantiated and attached to the component's trait, which is provided as an input argument to the service *subscribeTo*. The described semantic presented is close to the actual implementations used today.

The two data structures (*hashmaps*) are defined in order to manage the subscribers and their incoming messages for distribution. The first one (*subscribers*) consists of references of the components' traits as keys and the references to their attached connector traits as values and is used to manage the subscribers. The connector, using this hashmap, can determine which component is attached to each of the roles defined. This way, it can determine the sender of each new message received, the number of subscribers at any given time, etc. The second data structure (*messages*) is used for handling the messages that the connector receives. It consists of references to the communication trait of each attached component as a key and a List of the messages from each component respectively as values.

As explained in the previous sections these data structures are viewed as components with available services in the ports. Therefore, the inner elements that compose the MBAP can be seen in figure 17.

jADL MBAP Description

```
1. architecture MessageBusArch {

2.   instance msgBus = new MessageBus();
3.   instance comp1 = new C();
4.   instance comp2 = new C();

5.   //attachments
6.   attach(msgBus.s, comp1.s);
7.   attach(msgBus.s, comp2.s);
```

```
8.   comp1.s.subscribeTo(com1);
9.   comp2.s.subscribeTo(com1);

10.  //...

11.  comp1.s.unSubscribeFrom(com1);
12.  comp2.s.unSubscribeFrom(com1);

13.  //...

14.  instance comp3 = new C();
15.  attach(msgBus.s, comp3.s);
16.  comp3.s.subscribeTo(com1);

17.  //...

18.  comp3.s.unSubscribeFrom(com1);

19.  //...

20.  instance compN = new C();
21.  attach(msgBus.s, compN.s);
22.  compN.s.subscribeTo(com1);
23. }
```

Code Snippet 21. *MBAP description in jADL*.

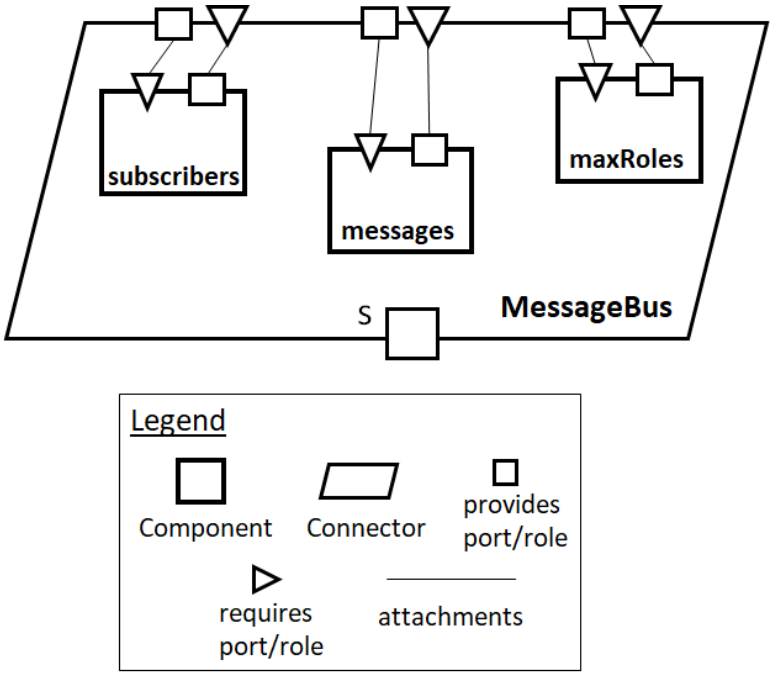


Fig. 17. *Inner components of MessageBus*.

In code snippet 21 the initialization script that needs to be executed after the definition of the elements is shown. At first, they are instantiated and then the topology of the system is defined. Only the initial attachments between the port and role *s* are defined. Once a component sends its request to subscribe to the *MessageBus* (e.g. line 8), the connector handles dynamically the rest of the interactions and ensures the receiving and sending of messages from and to the component.

3.6 Conclusion

In this chapter, the architectural description language jADL, created in this thesis, was presented. The architectural elements and the rest of the constructs of the language were analyzed. The syntax, as shown, is quite simple for architects and stakeholders to learn and can be familiar to practitioners, like for example the operator *new*, which is used for the instantiation of new architectural elements. Whilst being a formal architectural language, it defines an easy and elegant syntax which allows for good expressiveness and flexibility as shown in the previous sections. Its simplicity and familiarity to widely-used programming languages, combined with its capabilities for the expression of dynamic architectures, constitute the main features that can help towards the promotion of ADLs in further use in practice.

Furthermore, two simple case studies are presented in the final sections. First, the description of a Client-Server architecture, which constitutes a widely used model of communication in computing. Initially, a static architecture of a load balancing system, consisting of a client, a load balancing server and three servers, was described to present the capabilities of the language concerning the definition of the separate elements and the whole topology of the system. After the static description, in order to add dynamism, the architecture of a self-adapting load-balancing system is presented, with a varying number of clients and servers. Additionally, the description of the architecture of a Message Bus, a widely used architectural pattern for integrating a varying number of heterogeneous applications, is shown. In both cases, the language constructs provided by jADL (especially the use of communication traits) proved adequate to describe these systems and showed the flexibility of the language when it comes to expressing dynamic reconfigurations.

Chapter 4

μσADL

4.1 Introduction

In this chapter the extension created for jADL, named μσADL, is presented. Its aim is to provide the means for the description of MicroService architectures. It is designed to allow the definition of architectural descriptions with the use of simple structures that hide the formalities met in architecture description languages, which, as mentioned, can be discouraging for architects to use them. By adding an additional layer of abstraction, it omits unnecessary rigorous definitions, providing a practical way to adequately describe software systems that adopt this new architectural style alongside their software qualities.

The rest of this chapter is organized as follows. First, MicroService architectural style is briefly presented. Next, μσADL is presented, by focusing particularly on two issues concerning this architectural style; the communication between microservices and the data storage. Finally, a process is presented of how μσADL can be used in order to reach to an architectural description of a software system, from a Business Process Model and Notation (BPMN) diagram, using an illustrative example.

4.2 MicroService Architectures

Microservice Architecture (Microservices) (Amundsen et al., 2016; Newman, 2015) is a newly adopted architectural style which emerged in the last decade and becomes more and more popular. A number of industry leading companies, have migrated to microservices and initiated the application of this technology of software development with very encouraging and promising results after its appliance to large-scale software systems.

A lot of research has been oriented towards this architectural style in recent years and there is a rapidly growing number of studies concerned with various aspects of microservice architectures, like for example focusing on monitoring and management (Mayer and Weinreich, 2017), or the recovery of a micro-service architecture from an existing software system (Granchelli et al., 2017) etc.

But, as it is pointed out in (Francesco, 2017) there is a lack of such an architecture description language when it comes to specifically describing microservice architectures and architects tend to use languages that describe service-oriented architectures like SoaML, SOMA, etc.

The basic goal of this style is to build applications as a collection of independent and as loosely coupled as possible services, in contrast to monolithic applications, as can be seen in figure 18 where two sample architectures are presented. Although they present similarities with Service-Oriented Architectures (Erl, 2016) (SOA) and in a way can be considered as one form of SOA, there are important differences; e.g. each microservice has its own persistent storage which cannot be modified by other microservices.

Though the microservices architectural style itself is not yet precisely defined, a broadly accepted definition is the one given by Lewis and Fowler (Microservices 2014). They define micro-services as "*an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery*".

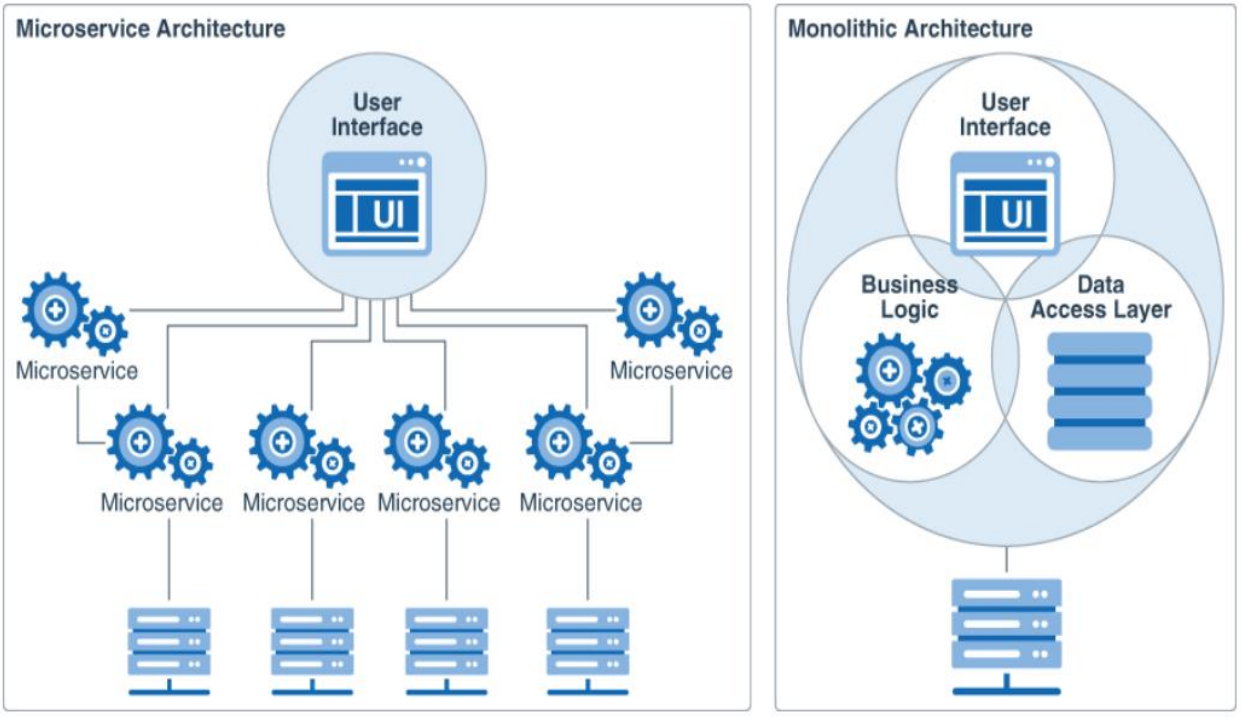


Fig. 18. *Microservice and Monolithic architectures* – reprinted from (Microservices Architecture 2019).

They have, also, outlined common characteristics of this style (Microservices 2014): i) *Componentization via Services*, ii) *Organized around Business Capabilities*, leading to development teams that are cross-functional with a wide range of skills required, iii) *Products not Projects*, meaning that the team that developed a product is responsible for it and after its completion, iv) *Smart endpoints and dumb pipes*, where the appropriate logic is applied upon the receiving of a request and the produced response is forwarded, v) *Decentralized Governance*, leading to a wide choice concerning the implementation technology used, vi) *Decentralized Data*

Management, meaning that each microservice has its own persistent storage, vii) *Infrastructure Automation*, which include automated processes for tests and deployment, and v) *Design for failure*.

4.3 μσADL Constructs

μσADL as a regular architectural language has two parts: textual representation of an architectural script, as well as a graphical part – graphical representation of the architecture. A microservice in μσADL is comprised of *ports*, a set of required *attributes*, (optionally) its private *database* (which can be directly accessed only by itself) and its *behavior*. Ports are defined the same way they are defined in jADL (two types – *provides* and *requires*) and they are configured (if necessary, only for the *provides* kind) using the *config* statement.

Each microservice represents a computational and data store element. Thus, it represents a component when it is translated from μσADL to jADL and all the statements/operators/etc. defined in jADL for components can be used when defining a microservice. In this section, we focus on the communication mechanisms between different microservices and the definition of their databases.

4.3.1 Communication Between Microservices in μσADL

Microservice architecture aims to lead to applications where the coupling is as loose as possible and the cohesion is as strong as possible. An approach usually described as smart endpoints and dumb pipes (Microservices 2014). It can be viewed as the filters in a Unix sense; microservices receive a request, apply the appropriate logic and produce a response. There are two ways primarily used for communication when building an application with microservices; direct communication using light-weight protocols (e.g. REST) or messaging over a lightweight message bus (Microservices 2014).

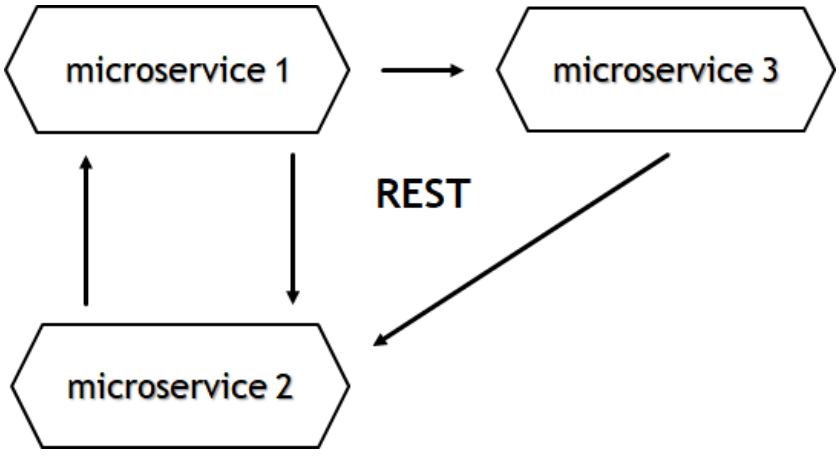


Fig. 19. *Microservices communicating via Synchronous Calls*.

In μσADL this is modelled in the following way. In the first case we have a typical Client-Server architectural pattern where one microservice acts as a client and sends a request to a second microservice (acting as a server), from which it awaits a response. The definition of the attributes concerning their locations (so that they are discoverable) and the ones concerning their protocols (the way the communication will be taking place) is enough for the appropriate connector to be created and the two microservices (in μσADL) or components (in jADL) to be able to communicate.

Generic connectors are used here, provided by jADL, and the generation of such connectors when translating to jADL can be seen in the next section. This way of communication is described in jADL in a similar way to the client-server communication, regarding the load balancer presented in the previous chapter.

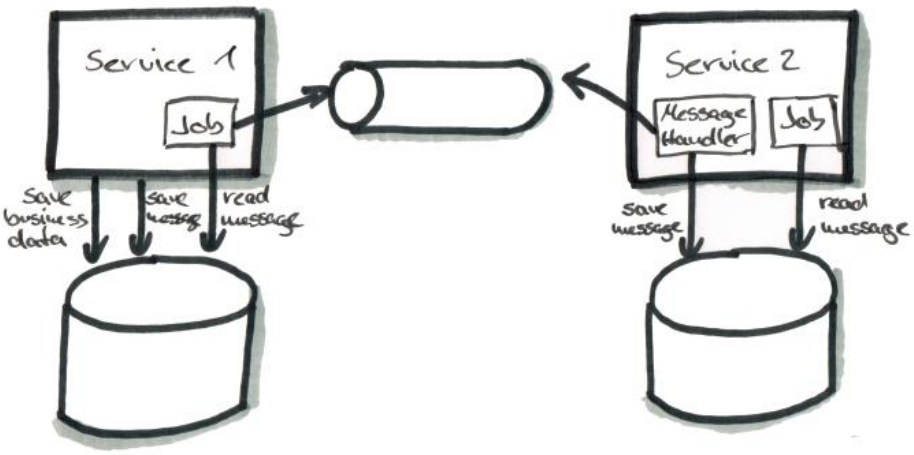


Fig. 20. *Microservices communicating via Messaging* – reprinted from (Microservice Communication Patterns 2018).

The second way of microservice communication is through a lightweight message/event bus. Each microservice that is subscribed to the bus produces messages/events that pushes to the bus and consumes messages/events from it. μσADL provides generic message and event buses for architects to use. Alongside with the use of predefined integrated in jADL communication traits presented in the previous chapter, varying types of buses can be described (e.g. publish/subscribe queues, routing queues etc.). They can be called directly, instantiated and used in the architectural description.

4.3.2 Data Storage in μσADL

In a microservice architecture, when it comes to persistent storage, as mentioned, it is favored a decentralized data management approach. Unlike monolithic applications, where usually a single logical database is preferred, here each microservice manages its own database, either different

instances of the same database technology or entirely different database systems (Polyglot Persistence) (Microservices 2014).

Following this principle, in μσADL we allow for each microservice to define its own instance of a database. Using the keyword *database* and inside the *{ }* the architect can define the necessary attributes for creating the connector he/she desires for a given microservice and a database. This way we provide an elegant way for defining the connections between a microservice and its database in μσADL, hiding all the formal requirements of jADL. Using this simple description, we can then automatically generate the appropriate connector in jADL.

For example, assuming we have a microservice, *ms1*, located in the same location with its database (*localhost*) and we need a JDBC standard connector. The description in μσADL would be:

```
microservice ms1 {
  // port(s) declarations and config(s)
  //...
  database {
    location: localhost;
    connector: JDBC;
    schema: invSchema;
    username: user1;
    password: mypass;
  }
  //...
}
```

The description presented above would result in creating in jADL a new database component and its appropriate connector, so that it can be attached to the microservice. The translated component and connector in jADL would be:

jADL translation

```
1. interface IConnJDBC {
2.   service void sendQuery (sqlString data);
3.   service void getQueryRes (sqlString data);
4. }

5. connector ConnJDBC {
6.    provides role IConnJDBC pClient;
7.    requires role IConnJDBC rClient;
8.    provides role IConnJDBC pDB;
9.    requires role IConnJDBC rDB;

10.   attribute string location = "localhost";
11.   attribute string username = "user1";
12.   attribute string password = "mypass";
```

```
13.   attribute string schema = "invSchema";

14.   config pClient as {
15.     service void getQuery (sqlString data) {
16.       rDB.sendQuery(data);
17.     }
18.   }

19.   config pDB as {
20.     service void sendQuery (sqlString data) {
21.       rClient.getQuery(data);
22.     }
23.   }
24. }

25. component DB {

26.   provides port IConnJDBC pDB;
27.   requires port IConnJDBC rDB;

28.   config pDB as {
29.     service void sendQuery (sqlString data) {
30.       rDB.getQuery(data);
31.       //process the query and send reply
32.     }
33.   }
34. }
```

Code Snippet 22. *The translated, in jADL, component and connector*.

The number of attributes is not limited to the ones shown in this example. The architect can add any number of attributes describing the database and its connection, which are then integrated in the code of the generic connector generated in jADL, as the ones described above.

4.4 Designing Microservices Using μσADL and BPMN

We adopt to the MicroServices style a similar approach to (Oquendo, 2008), which concerned the description of service-oriented architectures. We propose a process as an initial step towards a methodology for describing software systems that are built based upon this architectural style, using μσADL and jADL, consisting of 3 parts:

i) The extraction of an initial architectural sketch from a BPMN representation where each process (or a number of processes depending on the architect's choice) can be modeled as a microservice. At this stage the granularity of each microservice is defined.

ii) The specification of the architecture using μσADL. The language provides the necessary constructs to define both each of the micro-services and the overall architecture of the system (i.e. their communication mechanisms, etc.).

iii) The translation of the μσADL description to jADL description.

4.4.1 Case Study of a Simple Online Shopping System

Business Process Modelling Notation (BPMN) is a standardized visual notation for modelling business processes. In the figure below, a simple process of online shopping through a shopping site is presented.

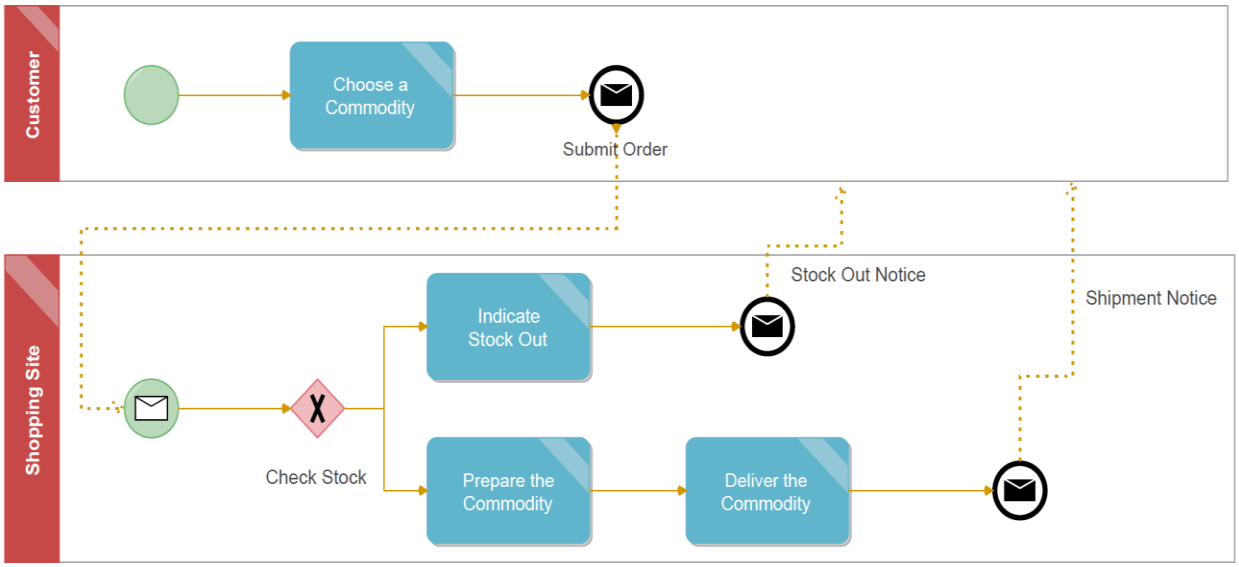


Fig. 21. *Online shopping process in BPMN* – reprinted from (Online Shopping Process 2019).

A customer chooses a commodity and sends its request to the site. The inventory is checked and either an out-of-stock notice is sent or a notice concerning the shipment details. The architecture of this system is dynamic: commodities can be added or removed and the way of delivering the goods may vary from customer to customer. From Software Architecture point-of-view this is a typical Client-Server communication model. The client (customer) sends its request to the server (shopping store) and after the request is processed a response is sent accordingly.

We focus now on the server and how it is organized. A typical approach from the past years would be a monolithic architecture. A single-tiered software application in which the user interface and data access code are combined into a single program from a single platform. It provides certain advantages, like simplicity when developing or deploying and (usually) a relatively easy way for scaling by running multiple instances behind a load balancer. But, as it has been observed, once an application becomes larger and larger significant drawbacks appear; the scaling of components with different resource requirements (CPU or memory intensive) is not possible, an update of a component requires the redeployment of the whole application, it requires long-term commitment to the technology stack (sometimes even to specific versions) chosen at the beginning of the development, etc.

Using the microservices architectural style described above the server can be componentized as follows. Each of the processes can be modeled as a separate microservice – the receiving of an order, the check of the inventory and the shipment information. These three microservices communicate with each other using a message bus, as shown in figure 22.

In architectural terms these various microservices can be viewed as components (locus of computation) and the message bus as a connector (locus of communication). In figure 22 we can see an initial architectural sketch concerning the architecture of the system (extracted from the BPMN model of figure 21) and in figure 23 a formal graphical representation of the server component (translated in jADL) is presented.

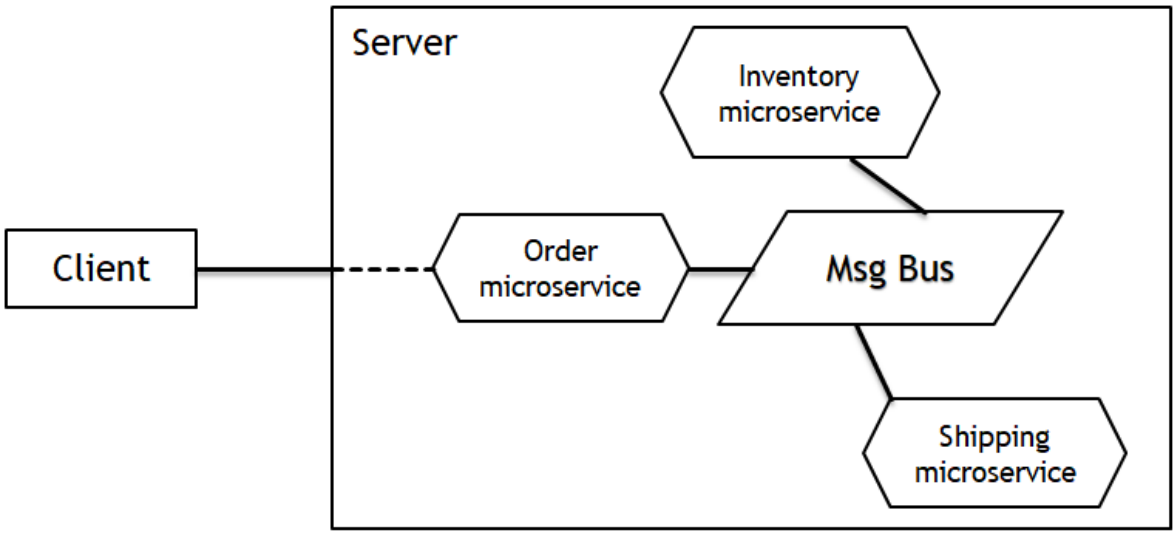


Fig. 22. *Online shopping system architectural sketch*.

The first step of the process is the extraction of an initial architectural sketch of the architecture of the software system from a BPMN model. At this stage the granularity of the microservices in an architecture is up to the architect. The definition of the appropriate granularity is still a field where there is a lot of on-going research, so for simplicity we use three microservices. The client and the composite server component, consisting of the microservices, can be seen in the figure above, which constitutes an informal graphical representation of the architecture of the software system.

The second step of the process proposed concerns the description of the architecture in μσADL. The description of the three microservices (*order*, *inventory* and *shipping*) is presented in code snippet 23. An important difference between the *order* and the other two microservices, is that *order* does not have a private database, unlike the other two. It just pushes a message when an order is accepted and returns a response to the client when the processing has finished.

The script used to instantiate the overall architecture of the server component can be seen in figure 24 - a screenshot from the editor developed for jADL and presented in the next chapter.

The language constructs provided by μσADL proved to be adequate for the description of each microservice and their communication mechanisms. Rigorous and too formal semantics are "hidden" in μσADL and the architect can define the architecture in a simple and elegant way.

During the third step of the process an automatic translation of the description from μσADL to jADL takes place. This is done in order to use the editor built for jADL for the validation of the defined architecture. The generated textual architectural description in jADL of the *Inventory* microservice can be seen in code snippet 24. Similarly, the jADL description of the rest of the architecture can be obtained.

Additionally, the translator to π-ADL (Oquendo, 2004) can be used in order to reach to implementation code stubs in the GO (Donovan and Kernighan, 2016) programming language, by using the PiADL2GO (Cavalcante et al., 2015) generator.

μσADL description

```
1. microservice Shipping {
2.   requires port ISubscribe r;

3.   trait ShipTrait aggregate CommTrait {
4.     config p as {
5.       service void getMsg (Message msg) {
6.         reply(msg);
7.       }
8.     }
9.   }

10.    instance com1 = new CTrait();

11.    database {
12.     location: "localhost";
13.     connector: "MySQL";
14.     schema: "shipSchema";
15.     username: "user1";
16.     password: "mypass1";
17.    }

18.    config pDB as {
19.      service void getQueryRes (type data) {
20.         com1.r.sendMsg(com1, data);
21.      }
22.    }
23. }



24. microservice Order {
25.   provides port IProcess req;
26.   requires port IResponse reply;

27.   trait CTrait aggregate CommTrait {
28.     config p as {
```

```
29.       service void getMsg (Message msg) {
30.         reply(msg);
31.       }
32.     }
33.   }

34.   instance com1 = new CTrait();

35.   config req as {
36.     service void procRequest (type data) {
37.       com1.r.sendMsg(com1, data);
38.     }
39.   }
40. }



41. microservice Inventory {

42.   requires port ISubscribe r;

43.   trait InvTrait aggregate CommTrait {
44.     config p as {
45.       service void getMsg (Message msg) {
46.         reply(msg);
47.       }
48.     }
49.   }

50.   instance com1 = new CTrait();

51.   database {
52.     location: "localhost";
53.     connector: "JDBC";
54.     schema: "invSchema";
55.     username: "user2";
56.     password: "mypass2";
57.   }

58.   config pDB as {
59.     service void getQueryRes (type data) {
60.       com1.r.sendMsg(com1, data);
61.     }
62.   }
63. }
```

Code Snippet 23. *MicroServices description in μσADL*.

Using this architectural style, we achieve decentralized governance, the scaling and/or update of each component can be achieved independently of the rest of the components and the choice of the technology used (programming language, database type, etc.) can be done autonomously for each of the components. Additionally, the adding of new functionalities can be easier, in respect

to monolithic architectures, since we can simply add a new microservice without (usually) having to worry about the other parts (components) of the application.

jADL description

```
1. interface IConnJDBC {

2.   service void sendQuery (type data);
3.   service void getQueryRes (type data);

4. }

5. component Inventory {

6.    requires port ISubscribe r;

7.    trait InvTrait aggregate CommTrait {
8.      config p as {
9.        service void getMsg (type msg) {
10.          reply(msg);
11.       }
12.     }
13.   }

14.   instance com1 = new CTrait();

15.   config pDB as {
16.     service void getQueryRes (type data) {
17.       com1.r.sendMsg(com1, data);
18.     }
19.   }

20. }


21. connector ConnJDBC {

22.   provides role IConnJDBC pClient;
23.   requires role IConnJDBC rClient;
24.   provides role IConnJDBC pDB;
25.   requires role IConnJDBC rDB;

26.   attribute string location = "localhost";
27.   attribute string username = "user2";
28.   attribute string password = "mypass2";
29.   attribute string schema = "invSchema";

30.   config pClient as {
31.     service void getQuery (sqlString data) {
32.       rDB.sendQuery(data);
33.     }
34.   }

35.   config pDB as {
```

```
36.      service void sendQuery (sqlString data) {
37.        rClient.getQuery(data);
38.      }
39.    }
40. }


41. component DBInventory {

42.    provides port IConnJDBC pDB;
43.    requires port IConnJDBC rDB;

44.    config pDB as {
45.      service void sendQuery (sqlString data) {
46.        //process the query and send reply
47.        rDB.getQuery(data);
48.      }
49.    }

50. }



51. component InventoryCont {

52.    requires port ISubscribe r;

53.    instance inv = new Inventory();
54.    instance conn = new ConnJDBC();
55.    instance dbinv = new DBInventory();

56.    attach(inv.com1.r, conn.pClient);
57.    attach(inv.com1.p, conn.rClient);
58.    attach(dbinv.pDB, conn.rDB);
59.    attach(dbinv.rDB, conn.pDB);

60.    bind(r, inv.r);

61. }
```

Code Snippet 24. *jADL description of the Inventory microservice*.

In order for the microservices to communicate the message bus architectural pattern (MBAP), described in the previous chapter of this thesis, was chosen. This is integrated in jADL and the communication trait needed is:

```
trait CommTrait {
  provides port IReceiveMsg p;
  requires port ISendMsg r;
}
```

When a microservice (or a component in jADL) uses this trait, it needs to define the behavior of the provides *p* port as shown in code snippet 23.

The script used to instantiate the overall architecture of the server component (the microservices and the message bus used for their communication) can be seen in figure 24 - a screenshot from the editor developed for jADL and presented in the next chapter.

After the instantiation of the server component we can define the architecture of the online shopping system which comprises of the server previously described, a client and a connector. Such a Client-Server architecture is adequately described in jADL in the load balancing architecture of the previous chapter.

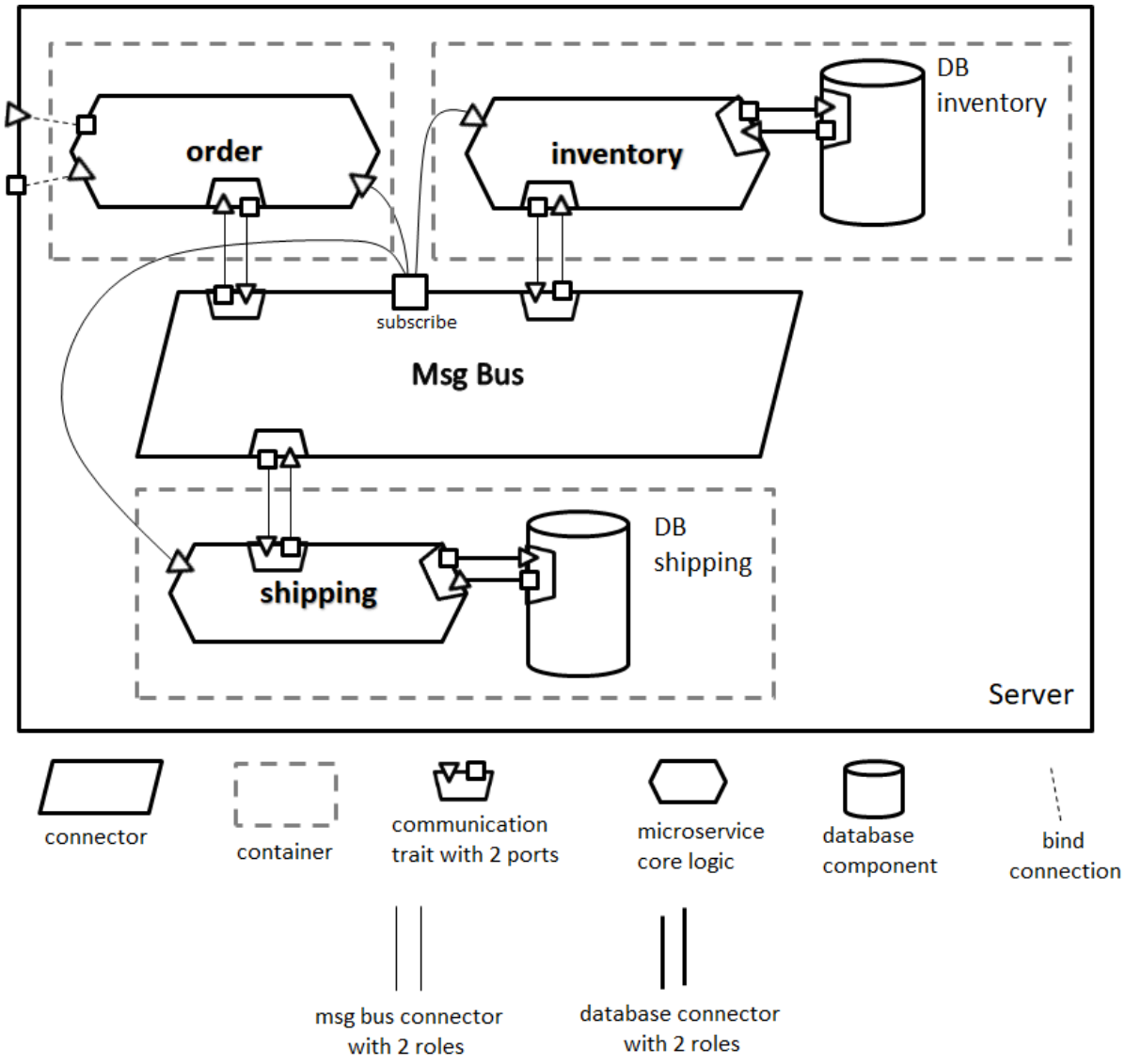


Fig. 23. *Graphical representation in jADL of the server component.*

Using a simple generic BPMN model that describes business processes concerning an online shopping store to describe the software system in μσADL, we have reached to the formal description of the architecture in jADL.

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

    instance order = new Order();
    instance inventory = new Inventory();
    instance shipping = new Shipping();
    instance msgBus = new MessageBus();

    attach(msgBus.s, order.r);
    attach(msgBus.s, inventory.r);
    attach(msgBus.s, shipping.r);

    order.r.subscribeTo(com1);
    inventory.r.subscribeTo(com1);
    shipping.r.subscribeTo(com1);
}
```

Fig. 24. *Description of the server in jADL.*

4.4.2 Dynamic Reconfiguration

A common characteristic of microservice architectures is the need for dynamic reconfiguration - i.e. the change (foreseen or unforeseen) of the topology of a software system during run-time.

Continuing with the previous example, the *Inventory* process can change in the future, hence a new instance of the reconfigured microservice must replace the old one. The language constructs *attach* and *detach* allow to easily describe such a change at the instance level of the given architecture. A script that can handle such a reconfiguration is shown below:

```
component Server {
  //...
  detach(mbus.s, inventory.r);
  instance inventory2 = new Inventory2();
  attach(mbus.s, inventory2.r);
  inventory2.r.subscribeTo(com1);
  //...
}
```

Scalability is another important quality attribute when it comes to microservices. Though it can be challenging, since it can demand the handling of various components, in μσADL one way to address this issue is to use the dynamic load balancer architectural pattern implemented in jADL.

In a similar way that the message bus is used in the previous section, the architect can use the default implementation or modify the behavior. Instead of the servers presented there, the instances of each microservice can be managed by such a load balancer, after configuring the behavior

concerning the upscaling of each microservice. By applying this to our example, the code concerning the server component becomes:

jADL description

```
1. component Server {

2.    provides port IProcess req;
3.    requires port IResponse reply;

4.    instance mbus = new MessageBus();

5.    instance myLB = new DynamicLB();
6.    instance inv2 = new Inventory();

7.    attach(mbus.s, myLB.r);
8.    attach(myLB.p, inv2.r);
9.    myLB.r.subscribe();

10.   instance myLBs = new DynamicLB();
11.   instance order2 = new Order();

12.   attach(mbus.s, myLBs.r);
13.   attach(myLBs.p, order2.r);
14.   myLBs.r.subscribe();

15.   instance myLBr = new DynamicLB();
16.   instance ship2 = new Shipping();

17.   attach(mbus.s, myLBr.r);
18.   attach(myLBr.p, ship2.r);
19.   myLBr.r.subscribe();
20. }
```

Code Snippet 25. *Server description in jADL*.

4.5 Conclusion

The extension of jADL, named μσADL, was presented in this chapter. An extension designed in order to ease the description of software systems that follow the microservice architectural style. The language constructs concerning the communication between the microservices and their data storage were analyzed.

One of the main goals of μσADL is to add an additional layer of abstraction, compared to jADL, where the rigorous and too formal requirements of jADL can be "hidden". When it comes to the persistent storage regarding a microservice, this is achieved with the use of the *database* declaration. As shown in the previous sections of this chapter, a simple declaration consisting of *name:value* tuples is used. In 4.3.2 such a declaration can be seen, as well as, the resulting connector and database component. As for the communication between microservices,

communication traits and the MBAP (since message buses constitute one of the main ways for their communication) discussed in the previous chapter, can be used. This further automates and eases the description of software systems built using microservices.

Additionally, a process was presented regarding the practical application of μσADL. It concerns a proposed way for reaching to a formal architectural description of a software system, by starting from BPMN diagram(s). As shown in the case study presented in this chapter, regarding an online shopping system, by following the three steps of the process proposed this can be achieved. With the use of a series of simple and elegant statements (code snippet 23) in μσADL, a detailed formal representation of the architecture in jADL (code snippet 24) can be obtained. Formal definitions regarding simple architectural elements (e.g. simple connectors for exchanging data, generic database components, etc.) can be omitted or significantly reduced and simplified, thus providing a more practical and user-friendlier way of describing software architectures. This can help towards the use of architectural languages in industrial software implementation processes, since BPMN are widely used in practice.

Finally, dynamic reconfiguration issues regarding the language were presented. With the use of the appropriate language constructs and the load balancer and MBAP descriptions from the previous chapter, simple foreseen and unforeseen dynamic reconfiguration and scalability issues can be addressed successfully. Though simple cases can be treated, the architectural style of microservices has appeared during the last several years and there is still a lot of on-going research around it. So, future work regarding μσADL concerns the improvement and/or introduction of additional processes/constructs/declarations/etc. to further support scalability, to provide a standardized way for defining the granularity of each microservice, etc.

Chapter 5

Tool Support / Evaluation

5.1 Introduction

In the previous chapters the jADL architecture description language and its extension for MicroService architectures, μσADL, have been presented. In order to support these languages a prototype tool has been developed. The tool aims in easing their use by providing the means for error-checking in the descriptions, automatic transformations etc.

The first thing that needs to be built is a parser for the language and then continue with the building of the tools mentioned. Two different frameworks were used during the time of this research; first we started with ANTLR and then moved to Xtext. Both of them are presented in the next sections. Additionally, a case study is presented for the evaluation of the language and the tools developed.

5.2 Initial Tool – ANTLR

The first parser that was created during this research was built using ANTLR (ANother Tool for Language Recognition) (ANTLR 2014). As an input ANTLR requires the definition of a grammar. The definition rules are described using an EBNF form and an example of such a rule for a component declaration is:

```
componentDeclaration
    :  'component' Identifier typeParameters?
       ('extends' type)?
       ('implements' typeList)?
       componentBody
    ;
```

The whole grammar used for the definition of jADL can be found in the appendix. The grammar defined at the time does not fully correspond to the final version of the grammar presented in the previous sections, because ANTLR was used at the beginning of this research while still experimenting with the grammar definition. For example, in the above definition we can see that in the component declaration, a component could be 'extended' by 'inheriting' elements from another component, which was later removed.

Additionally, it is important to note that the rules defined in the Appendix concern only the definition of the basic architectural elements in jADL and the special statements concerning the language. For the rules concerning simple statements, expressions, etc. another grammar was adopted to our needs. Since jADL defines statements in a Java-like way, the grammar defined from (antlr/codebuff 2013) was taken and modified accordingly.

Upon the definition of the grammar, ANTLR can automatically, using a simple command, generate a parser that can build and walk parse trees. It offers different options for the analysis of the abstract syntax tree. In our approach we used the Visitor Design pattern (Gamma et al., 1994) and through ANTLR, a visitor interface and a blank visitor implementation object can be automatically generated.

After the compilation - in our approach we chose the Java programming language (Urma et al., 2014) - of the created files from the translation of the grammar, using a command line interface we can type the jADL architectural description and view the generated abstract syntax tree, as shown in the figure below. The command used is:

grun jADLfull2 compilationUnit -tree

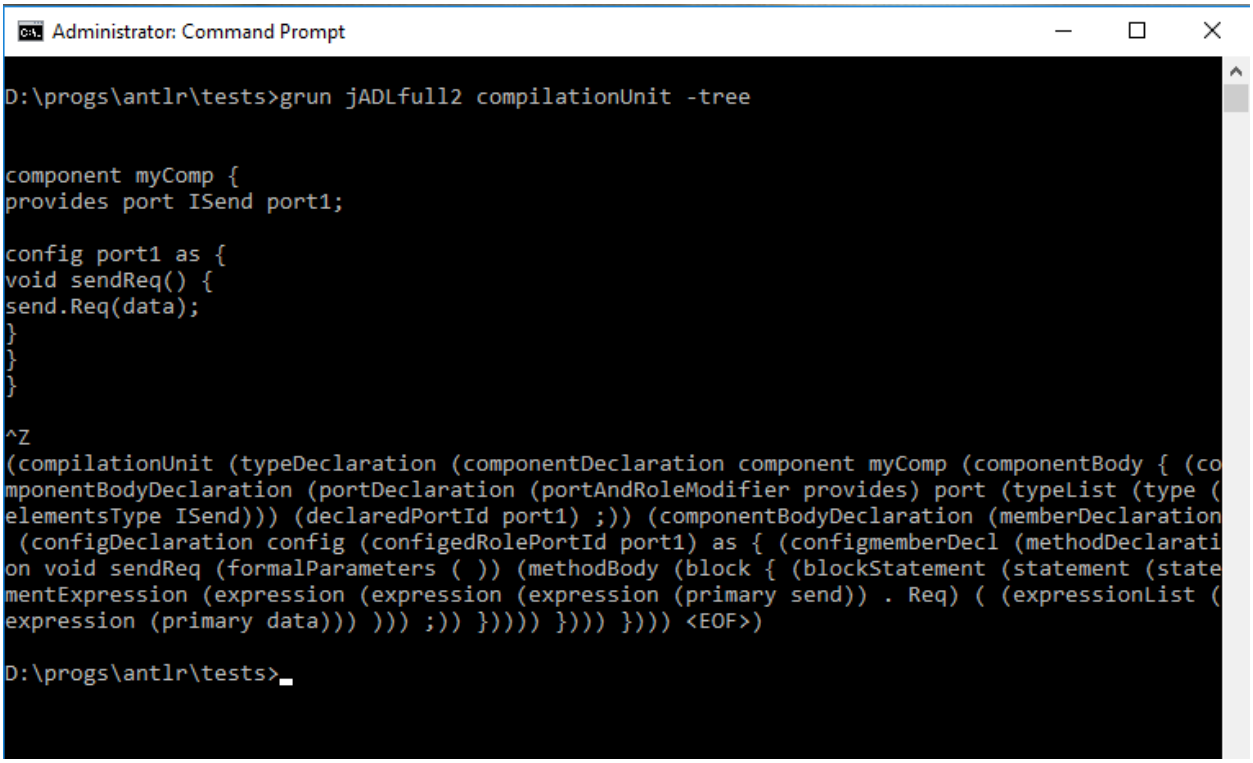


Fig. 25. *Abstract syntax tree from the description presented in a textual way*.

where *grun* loads an ANTLR library (org.antlr.v4.gui.TestRig), *jADLfull2* is the name of the grammar we used to generate and compile the files previously mentioned, *compilationUnit* is the

starting rule (the entry point) of the grammar and the final (optional) argument *-tree* is one of the options provided by ANTLR for further processing the description.

Also, the different tokens from a description can be extracted. In the figure below, we can see the various tokens from the previous example. The difference in the command used this time is that the optional argument is different (-tokens shows the tokens extracted while -tree in the previous case showed the abstract syntax tree) and that now the description is read from a file (the path at the end of the command defines the location of the input file containing the description).

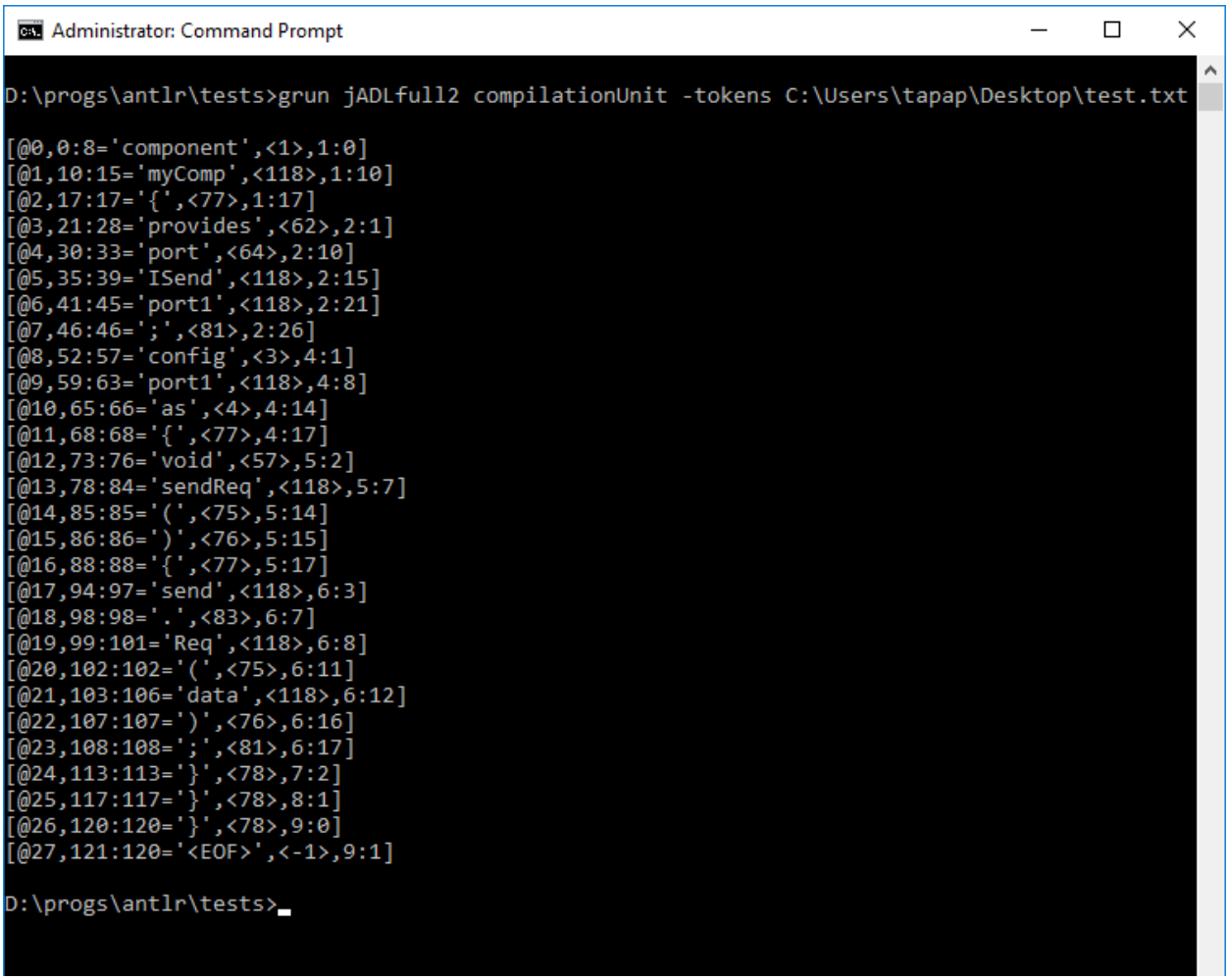


Fig. 26. *Reading from a file and extracting the tokens*.

Additionally, ANTLR provides a graphical user interface for viewing the generated tree. By changing the optional argument in the previous command from -tokens to -gui, we can obtain a "pretty-printed" version of the abstract syntax tree, as shown in the next figure.

Finally, despite of the advantages that ANTLR provides for the building of a parser (and other options not presented here, like, for example, the possibility for the integration in a Java program)

we chose to change the framework and moved to Xtext, which is presented in the section that follows.

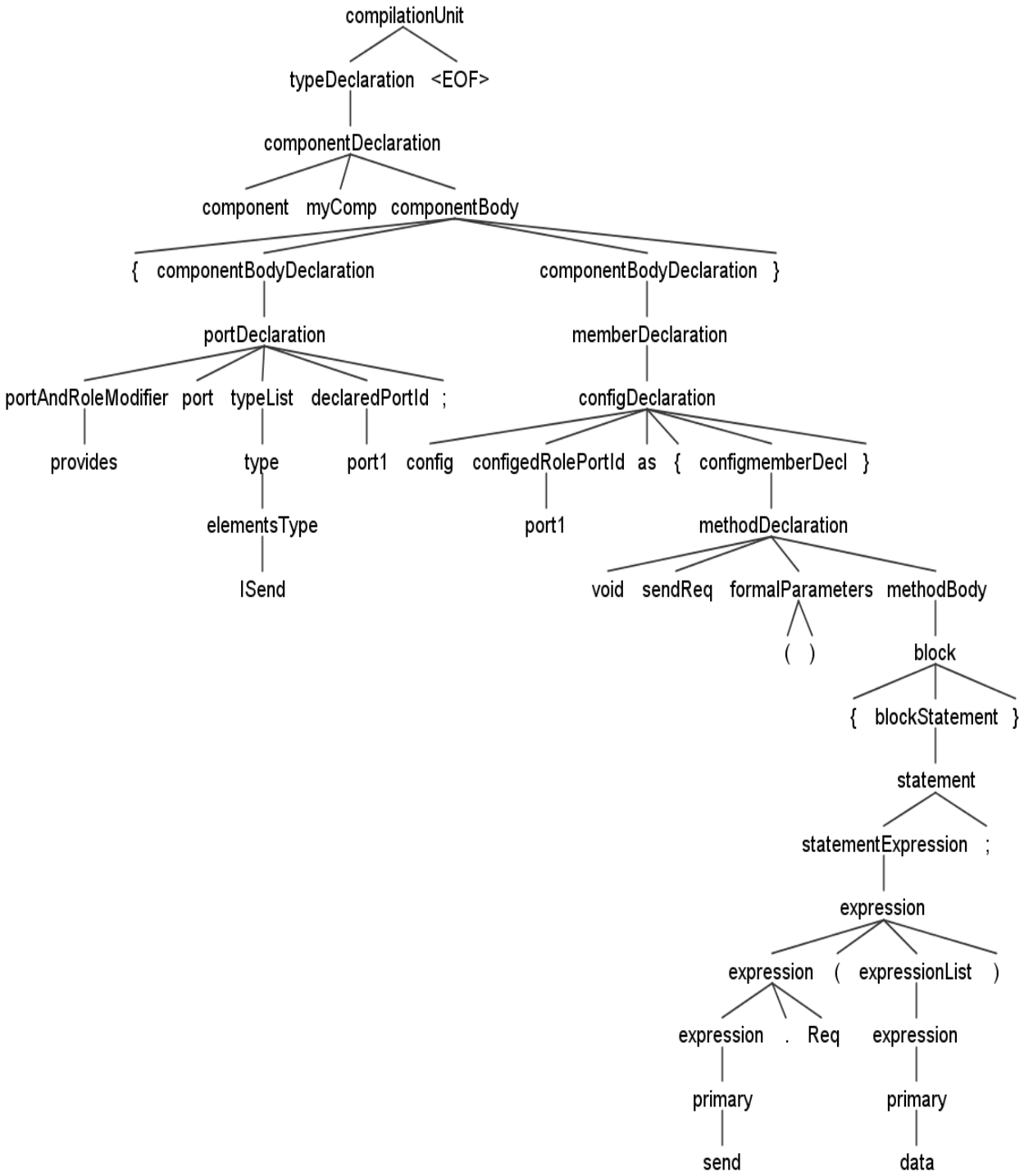


Fig. 27. *Abstract syntax tree from the description presented in a graphical way.*

5.3 Tool Support

Continuing this research, we decided to change and move on to the Xtext framework (Efftinge and Spoenemann, 2018). It is developed as a plugin for Eclipse and it provides valuable tools for the design of domain specific languages. It offers the possibility of automatically obtaining a parser and an editor for Eclipse, by the definition of the grammar rules of the language. The editor created, offers standard functionalities (e.g. auto-completion), as shown in the next section. The framework supports, also, the writing of programs in Xtend (Bettini, 2013). It is a Java-like language that can be used for adding additional code/functionality to grammar rules, defined in the language provided by Xtext, for the writing of validation rules or the creation of generators, etc.

5.3.1 Editor

First, an editor for specifying jADL architectural descriptions was created. As mentioned, the Xtext framework provides automatically this option, upon the definition of the grammar of the language. A screenshot from the editor obtained is depicted in figure 28. The rules are defined in a similar way to the one shown for ANTLR in the first section. For example, the grammar rule in jADL for component declaration would be:

componentDeclaration:
'component' name=ID '{'
componentBodyDecl += componentBodyMembers '}' ;*

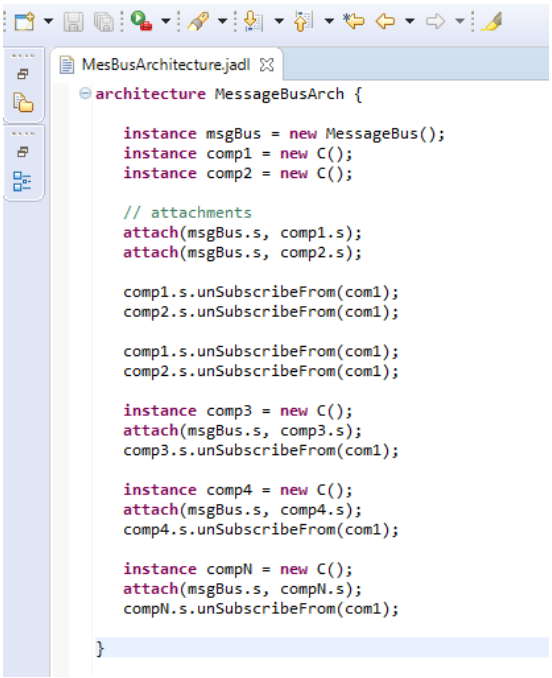


Fig. 28. *Editor for jADL descriptions.*

An important difference now, is that Xtext does not allow for recursion in the grammar rule definitions, unlike ANTLR. In the appendix, can be found the rest of the grammar rules used for the Xtext definition.

After defining the grammar for jADL the Eclipse editor was generated. Since it is an Eclipse editor, as shown in the screenshots in figure 29, it automatically supports typical functionalities for editors. In the figure, a syntax error detection (a) and an auto-completion proposal (b) are presented.

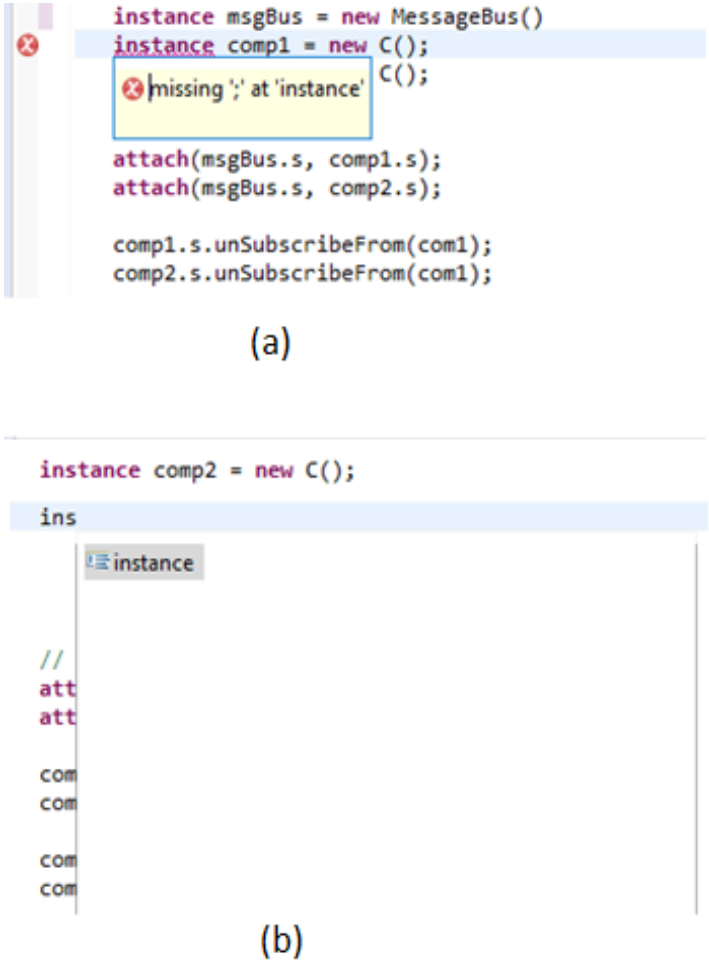


Fig. 29. *(a) error detection, (b) auto-completion.*

Additionally, there are helpful tools that can be added to such an Eclipse editor, in the form of Eclipse plugins. In figure 30, such a tool is shown. We added to the editor a node model outline plugin, which presents in a graphical tree-like structure the rules that are matched from the parsing of the architectural description.

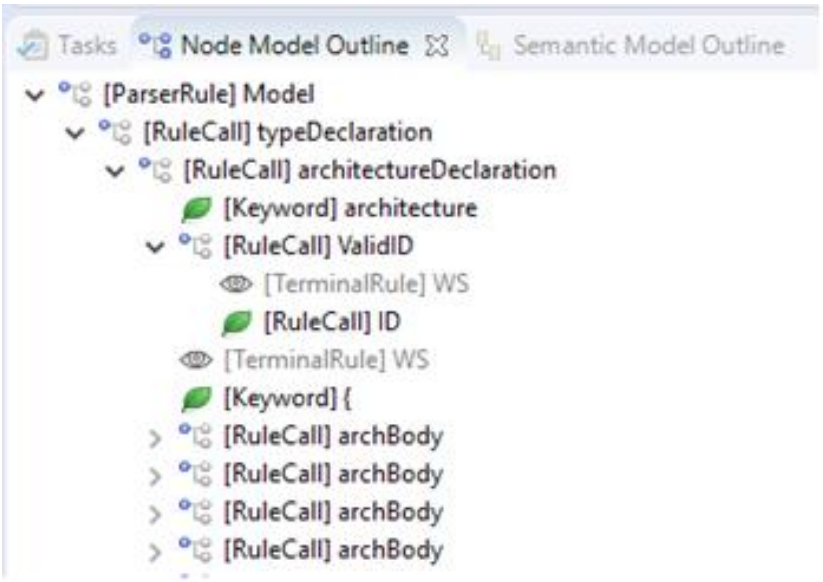


Fig. 30. *Node model outline plugin for Eclipse*.

5.3.2 Translator for π-ADL

For experimentation during the time of this research and as a first step towards the generation of software artefacts, we proposed a way for reaching to GO programming code from a jADL description, using π-ADL as an intermediate ADL. As shown in (Cavalcante et al., 2014), there is a generator of GO code from π-ADL specification. Therefore, we built a transformer to automate the process of the transformation from jADL to π-ADL description. In figure 31, an extended version of the table in (Cavalcante et al., 2014) is presented. Information has been added concerning the architectural elements in jADL, respectively to π-ADL and the GO programming language. The transformation process concerning each one of them is explained through this section.

Components & Connectors. Both ADLs consider components and connectors as first-class entities and follow the classical component/connector/system paradigm. In both cases, they are defined using the *component* and *connector* keywords followed by an identifier. Inside these declarations both ADLs define how the element will communicate with its environment (ports/roles/interfaces in jADL, connections in π-ADL) and what its behavior will be (through behavior in π-ADL and the *config* statement in jADL). They both correspond to *Functions* (*goroutine*) in the GO programming language.

Behavior. Exactly one behavior must be mandatorily declared in order to specify the behavior of each architectural element in π-ADL. Using the keyword *behavior*, the behavior is defined inside a block of code as a series of instructions/statements (e.g. type/variable declarations, function calls, etc.) (Cavalcante et al., 2014). In jADL the behavior is defined in a different way; with the use of the *config* statement the behavior of each provides port/role is defined as a set of statements. Any

additional behavior can be defined inside the body of the architectural element. So, in order to transform the behavior, we collect the *config* declarations (i.e. the services they provide) and any additional behavioral statements. Then, by using the choice statement we enclose them inside a behavioral block of π-ADL code.

π-ADL	jADL	Go
Component	Component	Function (*goroutine*)
Connector	Connector	Function (*goroutine*)
Behavior	Behavior	Body of function (*goroutine*)
Connection	Connections (*ports*/*roles*/*interfaces*)	Channel
Architecture	Architecture	Main function
Declaration of connections	Declaration of connections	Maps of channels
Unification of connections	Unification of connections	Channels as parameters to goroutines

Fig. 31. *Reprinted and extended from (Cavalcante et al. 2014).*

Connections. In π-ADL, both in components and connectors *connections* are defined. These connections are typed and constrained to the scope of the architectural element. They have an identifier, the direction of the connection (in/out) and an existing type. In jADL, on the other hand, we distinguish between the connections concerning components (ports) and the ones concerning connectors (roles). The fact that we can define N services in each port's/role's interface adds unnecessary complexity when it comes to transforming them to connections in π-ADL. Thus, at this stage, when it comes to transforming the description from jADL to π-ADL, we allow for exactly one service per interface (code snippet 26). This eases the process of extracting the type for each connection. The other two properties of each connection are parsed from the jADL description; the identifier and the direction (provides/requires – in/out) from the port or role declaration. These specifications correspond to *Channels* in the GO language.

Architecture. Both jADL and π-ADL, after defining each concrete element, define in a separate architectural declaration the topology of this architectural description. They both define the appropriate instances and how they are connected. In jADL this is defined using the keywords *instance* and *new*, while in π-ADL the keyword *is* (Cavalcante et al. 2014) is used. For the attachments between them in jADL, we use the *attach* statement and in π-ADL the keyword *unifies* is used between the two connections. It is important to notice that the unifications in π-ADL should be written in a specific way (from an output connection of an element to an input connection of

another). Thus, when transforming an attach statement to a unification statement in π-ADL, it is important to extract the direction of the port/role in the attach statement so that it can be placed on the correct side of the unification statement. An a*rchitecture* declaration corresponds to *Main Function* in the GO language.

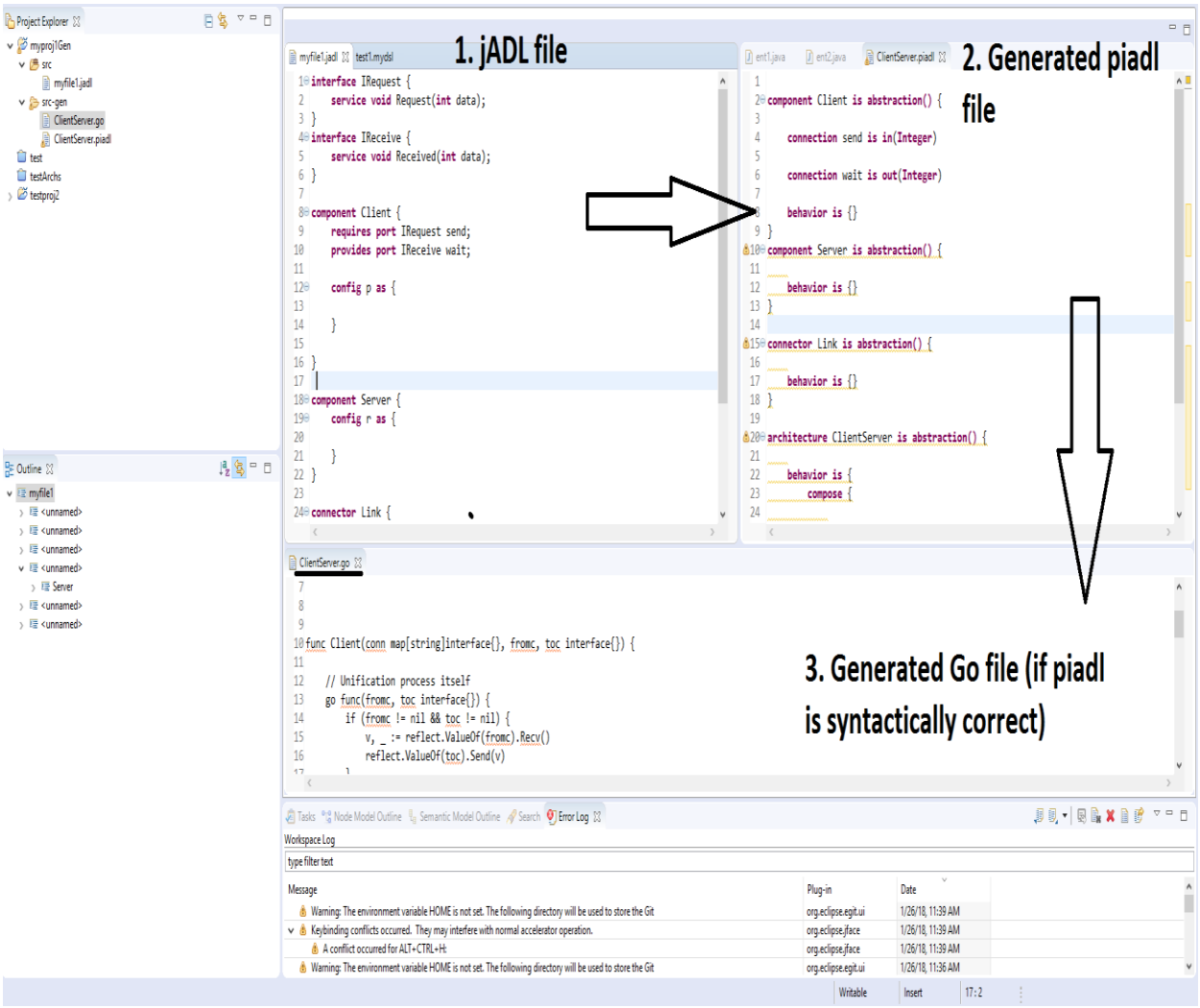


Fig. 32. *Translator to π-ADL for GO generation.*

Declaration and Unification of connections. The declarations and unifications of connections are defined in the *architecture* for both jADL and π-ADL, as explained above. When it comes to the GO programming language, the declarations of such connections correspond to *Maps of Channels* and their unifications to *Channels as Parameters* to *goroutines*.

Following the example in (Cavalcante et al., 2014), we define a similar simple jADL architectural description as shown in code snippet 26. In order for the generator to work, the syntax of the resulting π-ADL description must be syntactically correct.

jADL Description

```
1. interface IRequest {
2.   service void Request (int data);
3. }

4. interface IReceive {
5.   service void Received (int data);
6. }

7. component Client {
8.   requires port IRequest send;
9.   provides port IReceive wait;
10.   attribute int x = 0;

11.   config wait as {
12.     service void Received (int data) {
13.       x = data;
14.     }
15.   }

16.   while(true) {
17.     send.Request(myRand);
18.     delay 500;
19.   }
20. }

21. connector Link {
22.   provides role IRequest cReq;
23.   requires role IReceive cRes;
24.   provides role IResponse sRes;
25.   requires role IProcess sReq;

26.   config cReq as {
27.     service void aRequest (int data) {
28.       sReq.pRequest(data);
29.     }
30.   }

31.   config sRes as {
32.     service void aResponse (int data) {
33.       cRes.Received(data);
34.     }
35.   }
36. }

37. interface IResponse {
38.   service void Response (int data);
39. }

40. interface IProcess {
41.   service void procRequest (int data);
42. }
```

```
43. component Server {
44.   provides port IProcess req;
45.   requires port IResponse reply;
46.   attribute int data2 = 0;

47.   config req as {
48.     service void procRequest(int data) {
49.       data2 = data+1;
50.       reply.Response(resp);
51.     }
52.   }
53. }

54. architecture ClientServer {

55.   instance client = new Client();
56.   instance server = new Server();
57.   instance link = new Link();

58.   attach(link.cRes, client.wait);
59.   attach(link.cReq, client.send);

60.   attach(link.sRes, server.reply);
61.   attach(link.sReq, server.req);
62. }
```

Code Snippet 26. *Client-Server description in jADL*.

In code snippet 27 the π-ADL description obtained can be seen. We should note a significant difference between the generated code and the one in (Cavalcante et al., 2014) - the lack of *protocols* here. Protocols in π-ADL are used to enforce the value types that must be transmitted and the order in which the sending/receiving operations must be performed. During the implementation of this transformer we chose to omit them for two reasons; first, they are optional and, second, the type is declared during the connection declaration and the order of the sending/receiving operations is adequately defined inside the behavior of each architectural element.

Since the π-ADL architectural description is semantically and syntactically correct, the generation of GO programming code was successful. The generated code is presented in code snippet 28.

π-ADL Description

```
1. component Client is abstraction() {
2.   connection send is out(Integer)
3.   connection wait is in(Integer)

4.   behavior is {
5.     x is location[Integer]
6.     choose {
7.       via send send x
8.       behavior()
```

```
9.     } or {
10.       via wait receive v : Integer
11.       x = v
12.       behavior()
13.     }
14.   }  }

15. component Server is abstraction() {
16.   connection req is in(Integer)
17.   connection reply is out(Integer)

18.   behavior is {
19.     data2 is location[Integer]
20.     via req receive data : Integer
21.     data2 = data + 1
22.     via reply send data2
23.     behavior()
24.   }
25. }

26. connector Link is abstraction() {
27.     connection cReq is in(Integer)
28.     connection cRes is out(Integer)
29.     connection sRes is in(Integer)
30.     connection sReq is out(Integer)

31.   behavior is {
32.     choose {
33.       via cReq receive x : Integer
34.       via sReq send x
35.       behavior()
36.     } or {
37.       via sRes receive y : Integer
38.       via sReq send y
39.       behavior()
40.     }
41.   }
42. }

43. architecture ClientServer is abstraction() {
44.   behavior is {
45.     compose {
46.       client is Client()
47.       and server is Server()
48.       and link is Link()
49.     } where {
50.       link::cRes unifies client::wait
51.       client::send unifies link::cReq
52.       server::reply unifies link::sRes
53.       link::sReq unifies server::req
54.     }
55.   }
56. }
```

Code Snippet 27. *Client-Server description in π-ADL*.

GO (generated)

```
1. package main
2.
3. import (
4.     "reflect"
5. )
6.
7.
8. func Client(conn map[string]interface{}, fromc, toc interface{}) {
9.
10.    // Unification process itself
11.    go func(fromc, toc interface{}) {
12.           if (fromc != nil && toc != nil) {
13.                  v, _ := reflect.ValueOf(fromc).Recv()
14.                  reflect.ValueOf(toc).Send(v)
15.           }
16.    }(fromc, toc)
17.
18. }
19.
20.
21. func Server(conn map[string]interface{}, fromc, toc interface{}) {
22.
23.    // Unification process itself
24.    go func(fromc, toc interface{}) {
25.           if (fromc != nil && toc != nil) {
26.                  v, _ := reflect.ValueOf(fromc).Recv()
27.                  reflect.ValueOf(toc).Send(v)
28.           }
29.    }(fromc, toc)
30.
31. }
32.
33.
34. func Link(conn map[string]interface{}, fromc, toc interface{}) {
35.
36.    // Unification process itself
37.    go func(fromc, toc interface{}) {
38.           if (fromc != nil && toc != nil) {
39.                  v, _ := reflect.ValueOf(fromc).Recv()
40.                  reflect.ValueOf(toc).Send(v)
41.           }
42.    }(fromc, toc)
43.
44. }
45.
46.
47. func main() {
48.    client := map[string]interface{}{
49.           "send" : make(chan int64),
50.           "wait" : make(chan int64),
51.    }
52.    server := map[string]interface{}{
53.           "req" : make(chan int64),
```

```
54.           "reply" : make(chan int64),
55.     }
56.     link := map[string]interface{}{
57.           "cReq" : make(chan int64),
58.           "cRes" : make(chan int64),
59.           "sRes" : make(chan int64),
60.           "sReq" : make(chan int64),
61.     }
62.
63.
64.     go Link(link, client["wait"], link["cRes"])
65.     go Client(client, link["cReq"], client["send"])
66.     go Server(server, link["sRes"], server["reply"])
67.     go Link(link, server["req"], link["sReq"])
68. }
```

Code Snippet 28. *Generated GO programming code*.

5.4 Case Study for jADL Evaluation

For the further evaluation of the language a case study was considered. It concerns a gas station system, and it is comprised of 3 components; a customer, a cashier and a pump component, as shown in figure 33. The system works as follows; the customer component deposits a payment and requests a pump number from the cashier component. If the transaction is successful, the cashier component checks for an available pump and returns it to the customer component. The customer then requests the release of this pump from the pump component. Upon the receiving of a request from a customer, the pump component first verifies the payment of a given customer with the cashier component and then releases the pump. The architecture described was adapted to the one presented in (Naumovich et al., 1997), alongside the modification from (Ozkaya, 2016). The modification concerns the ports and attachments between the customer and the pump components. In the first one, there was a port for each customer in the pump component, whilst in the second and in the one presented here there is one port for multiple customers to connect.

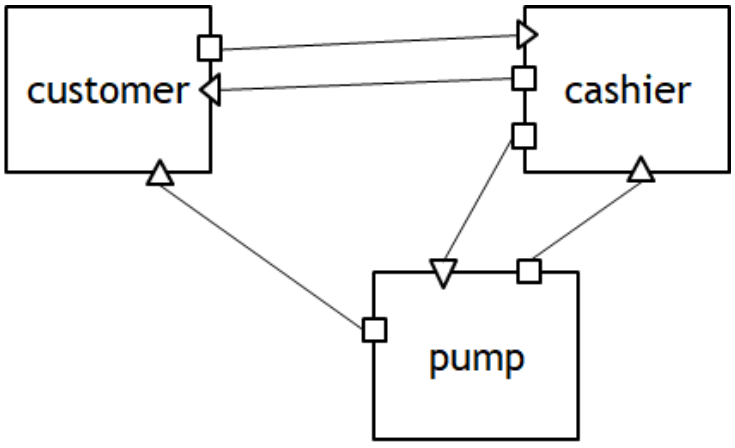


Fig. 33. *Graphical representation of the Gas Station system*.

First, the interfaces used, are declared in code snippet 29. The first one, *ICustomer*, is used for the communication between the customer and both the pump and the cashier. The second one, *IGas*, is used for the communication between the cashier and the pump component. We consider 2 interfaces, since we distinguish between the first type of communication (external customer) and the second where the cashier communicates with the *pump* component (internal communication). The services *payment* and *getGas* are used from the customer to make a payment to the cashier and to make a request for gas to the pump.

jADL Interfaces Description

```
1. interface ICustomer {

2.     service int payment(float amnt);
3.     service void getGas(int custId, int pumpId);
4.     service void getCustPump();
5. }

6. interface IGas {

7.     service int getPump();
8.     service boolean checkOrder(int custId);
9. }
```

Code Snippet 29. *Interfaces for the Gas Station system*.

The *customer* component, shown in code snippet 30, consists of three ports: *rCash*, *pCash* and *rPump*. Through its *rCash* port it requires a *pumpID*, after the successful completion of a payment. Once the payment is processed (in the cashier component) it sends a request to the pump component (line 8).

jADL Customer Description

```
1. component Customer {

2.     requires port ICustomer rCash;
3.     provides port ICustomer pCash;
4.     requires port ICustomer rPump;

5.     config pCash as {
6.            service void getCustPump() {
7.                   int pumpID = rCash.payment(amnt);
8.                   rPump.getGas(custID, pumpID);
9.            }
10.    }
11. }
```

Code Snippet 30. *Customer component description*.

Continuing with the *Cashier* component (code snippet 31), which consists of four ports: *rCust*, *pCust*, *rPump* and *pPump*. Through its *pCust provides* port, it accepts a request from customer

regarding a payment. After checking the amount, it requires information for the next available pump from the pump component, and sends it to the customer. Also, the *pPump* port is configured for providing a response to the pump component, concerning the payment status (successful or not) of a client.

jADL Cashier Description

```
1. component Cashier {

2.     requires port ICustomer rCust;
3.     provides port ICustomer pCust;
4.     requires port ICustomer rPump;
5.     provides port ICustomer pPump;

6.     config pCust as {
7.            service int payment(float amnt) {
8.                   if (check(amnt))
9.                          return rPump.getPump();
10.           }
11.    }

12.    config pPump as {
13.           service boolean checkOrder(int custid) {
14.                  if (check(custid))
15.                         return true;
16.                  else
17.                         return false;
18.           }
19.    }   }
```

Code Snippet 31. *Cashier component description*.

The *pump* component is shown next, in code snippet 32. Through its port *pCash* it sends the next available pump, upon each request from the cashier component. In the configuration of the *pCust* port the requests from a customer (*custID*) for the release of a pump (*pumpID*) is handled. If the check regarding the payment of the client is successful (line 12) the pump is released for the customer.

jADL Pump Description

```
1. component Pump {

2.     provides port ICustomer pCust;
3.     requires port IGas rCash;
4.     provides port IGas pCash;

5.     config pCash as {
6.            service int getPump() {
7.                   return nextpumpId;
8.            }
9.     }
```

```
10.    config pCust as {
11.           service void getGas(int custId, int pumpId){
12.                  if (rCash.checkOrder(custid))
13.                         releasePump(pumpId);
14.
15.           }
16.    }   }
```

Code Snippet 32. *Pump component description*.

Next, the architecture instantiation is presented in code snippet 33. Note that the *SimpleConn* and *SimpleConn2* connectors are not previously described in the section. This is because we considered them to be simple connectors for data exchange between the components in this case study. Thus, they are similar with the connector *Link*, presented in the previous section of this chapter. The first one (*SimpleConn*) consists of four roles and is used for the communication between the customer and the cashier and the pump and the cashier. The other (*SimpleConn2*) is used for the communication between the customer and the pump component. Each of the elements is instantiated (lines 2-7) and, finally, the attachments are defined (lines 8-17).

jADL Description

```
1. architecture GasStation {
2.   instance cust = new Customer();
3.   instance pump = new Pump();
4.   instance cash = new Cashier();

5.   instance Cust2Cash = new SimpleConn();
6.   instance Cust2Pump = new SimpleConn2();
7.   instance Pump2Cash = new SimpleConn();

8.   attach(Cust2Cash.p1, cust.rCash);
9.   attach(cash.pCust, Cust2Cash.r1);
10.   attach(Cust2Cash.r2, cust.pCash);
11.   attach(cash.rCust, Cust2Cash.p2);

12.   attach(Cust2Pump.p1, cust.rPump);
13.   attach(pump.pCust, Cust2Pump.r1);

14.   attach(Pump2Cash.p1, pump.rCash);
15.   attach(cash.pPump, Pump2Cash.r1);
16.   attach(Pump2Cash.r2, pump.pCash);
17.   attach(cash.rPump, Pump2Cash.p2);
18.
19. }
```

Code Snippet 33. *Architecture of the Gas Station system*.

From the above definitions of the three components and the overall architecture of the gas station system is shown that jADL provides the language constructs for adequately expressing the behavior of each of the components. This is achieved with the use of simple statements, like for example lines 7,8 in the customer component description. Moreover, the use of well-known by

practitioners constructs, like the *new* operator, and constructs which semantic meanings are pretty self-explanatory (like the *attach* statement used for declaring an attachment between a port and a role) can furtherly ease the adoption and use of the language, thus helping towards the promotion of the practical usage of architecture description languages.

5.5 Conclusion

In this chapter, the tool created for the language and a case study for its evaluation were presented. First, the initial tool build with the use of ANTLR was presented and some of its features were shown, like e.g. the extraction of a visual representation of the abstract syntax tree from a jADL textual description. Next, the framework was changed, as we moved to the Eclipse Xtext framework, mainly due to the automated capabilities offered by the framework and its integration with Eclipse. An editor was obtained after the definition of the grammar, which, as presented, has integrated standard editor functionalities (e.g. syntax error-detection) and, also, can be further extended with various plugins provided for Eclipse. Additionally, the translator to π-ADL is presented, which was used for experimentation regarding the generation of code. π-ADL has a tool for the generation of GO programming code from its architectural descriptions and that is why was used as an intermediate language. In the second part of this chapter, a case study was presented for the evaluation of jADL. The case study presented in (Naumovich et al., 1997) was chosen, as it was additionally modified by (Ozkaya, 2016). It concerns a gas station system, and it is comprised of 3 components; a customer, a cashier and a pump component. The description in jADL of each component and the overall architecture of the gas station system were presented and explained. The language proved to provide adequate support for the description of the architecture of the system.

Chapter 6

Conclusion

6.1 Research Summary

This research started from the point where we tried to determine and then address the issue regarding architecture description languages (ADLs) and their usage. ADLs are domain specific languages used in the domain of software architecture and software engineering. They describe software architectures from a higher level and they ignore lower level implementation details. They can provide the means for the validation and verification of a given architecture. Research surrounding architectural languages, like (Ozkaya, 2016; Malavolta et al., 2012; Minora et al., 2012), indicated as two important problems: (1) the high-degree of formality met in these languages and (2) their support for dynamic reconfigurations. Other issues could be the lack of such a language to describe particular architectural styles, like microservices (Francesco, 2017), or the lack of adequate tool support.

An extensive analysis of the literature was performed, as described in chapter 2. Due to the large number of existing ADLs, a subset had to be selected for this analysis. Based on the research findings mentioned, one of the criteria was their support for dynamic reconfiguration. The second one, was their support for user-defined connectors. We believe the latter to be an important aspect of ADLs, since it would allow decoupling between computation and communication. Then the subset was chosen and the results are presented throughout chapter 2.

The results of both these findings helped to define the following goals of this thesis in chapter 1:

- *the creation of a new Architecture Description Language, named jADL, which would formally describe dynamic architectures, but, also, by using a relatively simple syntax.*
- *the support from jADL of new architectural styles, like microservices.*
- *the support of the language with the development of a tool.*

In chapter 3, jADL is presented and explained in detail. Its language constructs and whole syntax is discussed. The syntax defined resembles widely used programming languages (e.g. the *new* operator for the instantiation of architectural elements) and is presented through EBNF. Despite the resemblance in some constructs, it is a formal architecture description language, whilst at the same time tries to address the problem that developers consider architectural languages to be too

formal to be used in practice. It provides both a textual and a graphical way to represent architectures. Additionally, in the end of the chapter, the practical use of the language and its various constructs is illustrated through the description in jADL of a simple Client-Server architecture and a Message Bus Architectural Pattern, through which the capabilities of the language for dynamic reconfiguration of a system are shown. One of the language constructs that jADL introduces, communication trait, proved most useful for handling dynamic reconfiguration. It constitutes a complex communication structure that can group together ports and roles, that can be used both during design and run-time.

In chapter 4, the second goal defined is addressed. μσADL is presented, an extension to jADL specifically for designing software architectures that follow the microservices architectural style. As mentioned in (Francesco, 2017), the lack of an architecture description language for the formal description of microservices results in architects using modelling languages for SOA, like SoaML. μσADL provides simple language constructs that can adequately describe microservice architectures. By providing an additional layer of abstraction, rigorous and too formal definitions are omitted or hidden behind simple statements. A simple series of value assignments leads to the creation of formal generic architectural elements as shown in chapter 4. Additionally, a process for the practical application of μσADL is proposed. Starting from a BPMN representation and by following three simple steps a formal architectural description can be obtained. An illustrative example is presented concerning the description of an online shopping system, through which is shown that μσADL provides the means for simply, yet also formally, describing microservice architectures. Though, there are still matters to be addressed as outlined in the next section concerning the scalability or the definition of the granularity for each microservice for example.

In chapter 5, the tool support for jADL and a case study for its evaluation are presented. First, the tool created using the Xtext framework is shown. Taking advantage of the features offered from Xtext an editor was created where jADL architectural descriptions are specified. The editor comes with integrated typical functionalities like e.g. auto-completion. Additionally, a transformer from jADL to π-ADL (for which a generator of GO programming code is built) was, also, created in order to be used for experimentation with the generation of software artefacts. The case study presented at the end of the chapter for the evaluation of jADL showed that the language provides the necessary constructs to describe the architecture of the system required.

6.2 Thesis Contributions

In the previous section a brief summary of the research conducted is presented. The theoretical and practical contributions of this thesis are:

- *Literature review and analysis*. Through the literature review the main problems, regarding architecture description languages (ADLs) and their usage are shown. In the analysis performed and presented in chapter 2, the advantages and disadvantages of a subset of existing ADLs are discussed.

- *The development of a new ADL*, named *jADL*, that:

    - *can support dynamic software architectures* and provides the means to handle dynamic reconfigurations of a given architecture.

    - *provides an easy to adopt and use syntax* for practitioners. The high degree of formality constitutes one of the major problems surrounding the usage of such languages and jADL provides a simple and familiar to developers syntax.

    - *can support the description of modern architectural styles*, such as the microservices architectural style, as shown in chapter 4.

- *The design and development of a tool to support the use of jADL*. In chapter 5, the editor created for the architectural descriptions in jADL is shown, alongside the translator implemented for π-ADL.

- *The description of common and widely used architectural patterns*. In chapter 3, a self-adapting load balancing and a message bus architectural patterns are described.

- *A case study for the evaluation of the language created*. A common case study used in the domain of software architecture is presented in chapter 5.

- *A process for the conversion of BPMN models to jADL models*. In chapter 4, through an illustrative example is shown how we can reach to a jADL specification starting from a BPMN model. This can help towards the promotion of the use of ADLs, since BPMN are widely used in practice.

6.3 Future Work

As discussed in the previous section, the main objectives set at the beginning have been achieved and the following directions for further research can be outlined as future work:

- the development of a graphical user interface for the visual definition of architectures, based on the graphical representation of jADL shown in chapter 3.

- the development of a compiler/generator for jADL, so that no intermediate language will be needed.

- experimentation with more case studies for the further and in-depth evaluation of the language.

- provide the means for validation of various aspects of microservices, such as granularity and scalability.

Appendix

A.1 jADL Syntax

```
<architecture_declaration> ::=
                "architecture" <id> "{" {
                      [<component_declaration>]*
                      [<connector_declaration>]*
                      [<interface_declaration>]*
                      [<trait_declaration>]*     }
                "}"
<interface_declaration> ::=
                "interface" <id> "{"
                      [<service_declaration>]*
                 "}"
<component_declaration> ::=
                "component" <id> "{" {
                      [<port_declaration>]*
                      [<config_statement>]*
                      [<internal_method>]*
                } "}"
<connector_declaration> ::=
           "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>")" ";"

<config_statement> ::=

          "config" <roleOrPortId> "as" "{"

               <behavior_implementation>

          "}" ";"

<attach_statement> ::=

          "attach" "("<role_id>, <port_id>")" ";"

<detach_statement> ::=

          "detach" "(" < role_id >, <port_id> ")" ";"

<select_statement> ::=

          "select"

               ["when" <booleanGuard> "=>" ] "{"

                    statement(s)          "}"

               [ "or"

                    ["when" <booleanGuard> "=>" ] "{"

                    statement(s)        "}" ]*

               "end;"

<trait_declaration> ::=

          "trait" <id> "{"

          [ <port_declaration>* | <role_declaration>* ]   "}"
```

<*trait_aggregation*> ::=

"**trait**" <*id*> "**aggregate**" <*trait_declaration*> "**{**"

<*config_statement*>* "**}**"

<*delay_statement*> ::=

"**delay**" <*int*> "**;**"

| "**delay_until**" <*expression*>* "**;**"

<*process_statement*> ::=

"**process**" "**;**"

<*attribute_declaration*> ::=

"**attribute**" <*type*> <*id*> "**=**" <*val*> "**;**"

<*service_declaration*> ::=

"**service**" <*type*> <*id*> "**(**" <args>* "**)**"

["**{**" <*statements*>+ "**}**"

| "**;**"]

<*args*> ::=

<*type*> <*id*> ("**,**" <*type*> <*id*>)*

<*statements*> **::=**

<*jADL_statements*>

| <*reg_statements*>

<*jADL_statements*> ::=

<*attach_statement*>

| <*detach_statement*>

| <*delay_statement*>

| <*process_statement*>

| <*select_statement*>

| <*bind_statement*>

<*reg_statements*> **::=**

<*if_statement*>

| <*while_statement*>

| <*for_statement*>

| <*do_while_statement*>

| <*assign_statement*>

<*types*> **::=**

<*simple_types*>

| <*complex_types*>

<*simple_types*> **::=**

“**string**”

| “**int**”

| “**type**”

| “**float**”

<*complex_types*> **::=**

<*hashmap_declaration*>

| <*array_declaration*>

| <*list_declaration*>

<*val*> **::=**

<*string_val*>

| <*int_val*>

| <*float_val*>

<*id*> **::=**

[Aa-Zz]*[0-9]*

A.2 Tool Realization Using ANTLR

In this section the grammar rules used for the building of the parser with ANTLR are presented. As mentioned, they concern only the definition of the basic architectural elements in jADL and the special statements concerning the language. For the rules concerning simple statements, expressions, etc. the grammar from (antlr/codebuff 2013) was adopted to our needs.

```
grammar jADLfull;

// starting point for parsing a file
compilationUnit
   :  typeDeclaration* EOF
   ;

typeDeclaration
   :  interfaceDeclaration
   |  componentDeclaration
   |  connectorDeclaration
   |  ';'
   ;

componentDeclaration
   :  'component' Identifier typeParameters?
      ('extends' type)?
      ('implements' typeList)?
      componentBody
   ;

connectorDeclaration
   :  'connector' Identifier typeParameters?
      ('extends' type)?
      ('implements' typeList)?
      connectorBody
   ;

portDeclaration
   :  portAndRoleModifier 'port' ('synchronized')? typeList+ declaredPortId (',' declaredPortId)* ';'
   ;

declaredPortId
   :  Identifier
   ;

roleDeclaration
   : portAndRoleModifier 'role' ('synchronized')? typeList+ declaredRoleId (',' declaredRoleId)* ';'
   ;
```

```
declaredRoleId
   :  Identifier
   ;

portAndRoleModifier
   :  'provides'
   |  'requires'
   ;

attributeDeclaration
   :  'attribute' type Identifier '=' variableInitializer ';'
   ;

typeParameters
   :  '<' typeParameter (',' typeParameter)* '>'
   ;

typeParameter
   :  Identifier ('extends' typeBound)?
   ;

typeBound
   :  type ('&' type)*
   ;

interfaceDeclaration
   :  'interface' Identifier typeParameters? ('extends' typeList)? interfaceBody
   ;

typeList
   :  type (',' type)*
   ;

componentBody
   :  '{' componentBodyDeclaration*  '}'
   ;

connectorBody
   :  '{' connectorBodyDeclaration* '}'
   ;

componentBodyDeclaration
   :  portDeclaration
   |  attributeDeclaration
   |  memberDeclaration
   |  blockStatement    ;
```

```
connectorBodyDeclaration
   :  roleDeclaration
   |  attributeDeclaration
   |  memberDeclaration
   |  blockStatement
   ;

interfaceBody
   :  '{' interfaceBodyDeclaration* '}'
   ;

configmemberDecl
   :  methodDeclaration
   |  fieldDeclaration
   ;

memberDeclaration
   :  methodDeclaration
   |  genericMethodDeclaration
   |  fieldDeclaration
   |  constructorDeclaration
   |  genericConstructorDeclaration
   |  interfaceDeclaration
   |  configDeclaration
   ;

methodDeclaration
   :  (type|'void') Identifier formalParameters ('[' ']')*
      ('throws' qualifiedNameList)?
      (   methodBody
      |   ';'
      )
   ;

configDeclaration
   :  'config' configedRolePortId (',' configedRolePortId)* 'as' '{' configmemberDecl* '}'
   ;

configedRolePortId
   :  Identifier
   ;

fieldDeclaration
   :  type variableDeclarators ';'
   ;
```

```
interfaceBodyDeclaration
   :  interfaceMemberDeclaration
   |  ';'
   ;

interfaceMemberDeclaration
   :  constDeclaration
   |  interfaceMethodDeclaration
   |  genericInterfaceMethodDeclaration
   |  interfaceDeclaration
   ;

interfaceMethodDeclaration
   :  (type|'void') Identifier formalParameters ('[' ']')*
      ('throws' qualifiedNameList)?
      ';'
   ;

formalParameters
   :  '(' formalParameterList? ')'
   ;

formalParameterList
   :  formalParameter (',' formalParameter)* (',' lastFormalParameter)?
   |  lastFormalParameter
   ;

formalParameter
   :  variableModifier* type variableDeclaratorId
   ;

lastFormalParameter
   :  variableModifier* type '...' variableDeclaratorId
   ;

methodBody
   :  block
   ;

qualifiedName
   :  Identifier ('.' Identifier)*
   ;

literal
   :  IntegerLiteral
   |  FloatingPointLiteral
```

```
    |   CharacterLiteral
    |   StringLiteral
    |   BooleanLiteral
    |   'null'
    ;

// STATEMENTS / BLOCKS

block
    :   '{' blockStatement* '}'
    ;

blockStatement
    :   localVariableDeclarationStatement
    |   statement
    |   typeDeclaration
    ;

localVariableDeclarationStatement
    :    localVariableDeclaration ';'
    ;

localVariableDeclaration
    :   variableModifier* type variableDeclarators
    ;

statement
    :   block
    |   ';'
    |   statementExpression ';'
    |   Identifier ':' statement
    |   connectStatement
    |   bindStatement
    |   selectStatement
    |   processStatement
    |   delayStatement
    ;

connectStatement
    :   'connect' '(' expression ',' expression ')' ';'
    ;

bindStatement
    :   'bind' '(' expression ',' expression ')' ';'
    ;
```

```
selectStatement
   :  'select'
         ('when' expression '=>')? '{'
            statement* '}'
      ( 'or'
         ('when' expression '=>')? '{'
            statement* '}' )*
      'end;'
   ;

delayStatement
   :  'delay' IntegerLiteral ';'
   |  'delay until' expression ';'
   ;

processStatement
   :  'process' ';'
   ;

PLUG: 'plug';
INTO: 'into';
PROVIDES: 'provides';
REQUIRES: 'requires';
PORT: 'port';
ROLE: 'role';
ATTRIBUTE: 'attribute';
CONNECT: 'connect';
BIND: 'bind';
```

A.3 Tool Realization Using Xtext

```
grammar org.xtext.Jadl09b with org.eclipse.xtext.common.Terminals

generate jadl09b "http://www.xtext.org/Jadl09b"

//start from here...
Model:
        types+=typeDeclaration*;

typeDeclaration :
        interfaces+=interfaceDeclaration
   | components+=componentDeclaration
   | connectors+=connectorDeclaration
   | archs=architectureDeclaration
   //| stmnts+=XJStatementOrBlock
   | 'archelement' name=QualifiedName ';'
   | traits+=traitDeclaration
;

QualifiedName:
        ID('.'ID)*
;


//interfaces
interfaceDeclaration:
        'interface' name=ID '{'
                interfaceBodyDecl=interfaceBody '}'
;

interfaceBody :
         interfaceBodyDeclaration+=interfaceMethodDeclaration+
;

//interfaceBodyMembers :
//        interfaceMethodDeclaration
//;

interfaceMethodDeclaration :
        'service' type=types name=ID
```

```
        paramsDecl=formalParameters ('[' ']')* ';'
;

formalParameters :
        {formalParameters} '(' paramsDecl=formalParameterList? ')'
;

formalParameterList :
        paramDecl=formalParameter (',' paramsDecl+=formalParameter)*
;

formalParameter :
        type=types name=variableDeclaratorId
;

variableDeclaratorId :
        name=ID ('[' ']')*
;


//components
componentDeclaration:
        'component' name=ID '{'
        componentBodyDecl+=componentBodyMembers* '}'
;

//componentBody :
//        {componentBody} componentBodyDeclaration+=componentBodyMembers+

componentBodyMembers :
        ports+=portDeclaration
        | jADlst+=jADlstatements
   | attrs+=attributeDeclaration
   | configs+=configDeclaration
;

portDeclaration :
        {portDeclaration}        type=portAndRoleType        'port'        ('synchronized')?
InterfaceImpl=[interfaceDeclaration] name=ID ';'
;
```

```
portAndRoleType :
        'provides'
        | 'requires'
;


//connectors
connectorDeclaration:
        'connector' name=ID '{'
        connectorBodyDecl+=connectorBodyMembers* '}'
;

//connectorBody :
//        {connectorBody} connectorBodyDeclaration+=connectorBodyMembers+
//;

connectorBodyMembers :
        roles+=roleDeclaration
        | jADLst+=jADlstatements
   | attrs+=attributeDeclaration
   | configs+=configDeclaration
   //| connTraitAggr
;

roleDeclaration :
        {roleDeclaration}        type=portAndRoleType        'role'        ('synchronized')?
Interimpl=[interfaceDeclaration] name=ID ';'
;


//configurations, methods
configDeclaration :
        'config' name=ID 'as' '{'
                configBody=configBodyMembers
        '}'
;

configBodyMembers:
        {configBodyMembers} configMembers+=(methodDeclaration | jADlstatements)*
;
```

```
methodDeclaration :
	'service' type=types name=ID paramsDecl=formalParameters ('[' ']')* '{'
		methodBody=methodBodyMembers
	'}'
;

methodBodyMembers:
	{methodBodyMembers} methodMembers+=jADlstatements*
;

//traits
traitDeclaration:
	'trait' name=ID '{'
		(ports+=portDeclaration)*
		(roles+=roleDeclaration)*
	'}'
;


//attributes
attributeDeclaration :
	'attribute' attrType=types name=ID '=' attrInit=(ID | initVals) ';'
;

types:
	'string'
	| 'int'
	| 'float'
	| 'boolean'
	| 'type'
	| 'void'
;

initVals:
	STRING
	| INT
	| FLOAT
	| Boolean
;
```

```
Boolean:
	'true'
	| 'false'
;

FLOAT:
	INT '.' INT
;

//jADL special statements
jADlstatements :
	jadltype=attachStatement
	| jadltype=detachStatement
	| jadltype=bindStatement
	| jadltype=selectStatement
	| jadltype=delayStatement
	| jadltype=processStatement
	| jadltype=newElementInst
	//| jadltype=elemTraitAggr
	| jadltype=Expression ';'
;

newElementInst:
	'instance' name=ID '=' 'new'  type=ID '()' ';'
;

attachStatement :
	{attachStatement} 'attach' '(' attachParam1=Expression ',' attachParam2=Expression ')' ';'
;

detachStatement :
	{detachStatement} 'detach' '(' detachParam1=Expression ',' detachParam2=Expression ')' ';'
;

bindStatement :
	{bindStatement} 'bind' '(' bindParam1=Expression ',' bindParam2=Expression ')' ';'
;

selectStatement :
	{selectStatement} => 'select' '{'
      ('when' => expressionDecl=jADlstatements '=>')?
```

```
        statementDecl1+=jADlstatements+ '}'
   ( 'or' '{'
      ('when' => expressionOrDecl+=jADlstatements '=>')?
        statementDecl2+=jADlstatements+ '}' )*
   'end;'
;

delayStatement :
        {delayStatement} 'delay' delayValue=INT ';'
   | {delayStatement} 'delay_until' delayStatement=Expression
;

processStatement :
        {processStatement} 'process' ';'
;


//architecture
architectureDeclaration :
        'architecture' name=ID '{'
                archBodyDecl=archBodyMembers '}'
;

archBodyMembers :
        {archBodyMembers} archElements+=jADlstatements*
;

// Expression grammar adapted from the tools from (Cavalcante et al., 2014)

Expression:
        LogicalExpression
;

LogicalExpression returns Expression:
        EqualityExpression
        ({LogicalExpression.left=current} op=('||' | '&&')
        right=EqualityExpression)*
;

EqualityExpression returns Expression:
        RelationalExpression
```

```
        ({EqualityExpression.left=current} op=('==' | '!=')
        right=RelationalExpression)*
;

RelationalExpression returns Expression:
        ArithmeticExpression
        ({RelationalExpression.left=current} op=('>=' | '<=' | '>' | '<')
                right=ArithmeticExpression)*
;

ArithmeticExpression returns Expression:
        Term
        (({SumOperation.left=current} op='+' | {MinusOperation.left=current} op='-')
        right=Term)*
;

Term returns Expression:
        Factor
        ({Term.left=current} op=('*' | '/' | 'mod') right=Factor)*
;

Factor returns Expression:
        '(' Expression ')' |
        UnaryExpression |
        AtomicElement
;

UnaryExpression returns Expression:
        {NegationExpression} => '!' expression=AtomicElement
;

AtomicElement returns Expression:
        LiteralElement | {VariableRef} variable=ID
        //| {CallRef} calledElement=ID ('.'next+=ID)* ( '(' ( ('.')? nextArg+=ID)* ')' )
;

LiteralElement returns Expression:
        IntegerLiteral | RealLiteral | StringLiteral | BooleanLiteral
;

IntegerLiteral returns Expression:
```

```
	{IntegerLiteral} value=INT
;

RealLiteral returns Expression:
	{RealLiteral} value=FLOAT
;

StringLiteral returns Expression:
	{StringLiteral} value=STRING
;

BooleanLiteral returns Expression:
	{BooleanLiteral} value=('true' | 'false')
;

//terminal REAL:
//	INT '.' INT
//;
```

```
/*
 * generated by Xtext 2.10.0
 */
package org.xtext.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext

import org.xtext.jadl09b.architectureDeclaration

import org.xtext.jadl09b.interfaceDeclaration
import org.xtext.jadl09b.interfaceBody
import org.xtext.jadl09b.interfaceMethodDeclaration
import org.xtext.jadl09b.formalParameters
import org.xtext.jadl09b.formalParameterList
import org.xtext.jadl09b.formalParameter

import org.xtext.jadl09b.componentDeclaration
import org.xtext.jadl09b.portDeclaration

import org.xtext.jadl09b.connectorDeclaration
import org.xtext.jadl09b.roleDeclaration

/**
 * Generates code from your model files on save.
 *
 *	See	https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-
generation
 */

class Jadl09bGenerator extends AbstractGenerator {

//global vars and fileToWrite...
	int i=0
	portDeclaration currPort

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext
context) {
	for (e : resource.allContents.toIterable.filter(architectureDeclaration)) {
```

```
                fsa.generateFile(
            e.name + ".piadl",
            e.compile(resource))
        }
        i=0
    }

//architecture and file definition
    def compile(architectureDeclaration e, Resource resource) '''
        «FOR f : resource.allContents.toIterable.filter(componentDeclaration)»
                        «f.compileComp»
        «ENDFOR»

         «FOR g : resource.allContents.toIterable.filter(connectorDeclaration)»
                        «g.compileConn»
        «ENDFOR»

                architecture «e.name» is abstraction() {

                        behavior is {
                                compose {

                        «FOR f : resource.allContents.toIterable.filter(componentDeclaration)»
                                «IF (i++>0)» and «ENDIF» «f.name.substring(0, 1)» is «f.name»()
                                        «ENDFOR»
                        «FOR f : resource.allContents.toIterable.filter(connectorDeclaration)»
                                «IF (i++>0)» and «ENDIF» «f.name.substring(0, 1)» is «f.name»()
                                        «ENDFOR»
                                } where {
                                }
                        }
                }
    '''

//components, ports, etc...
    def compileComp(componentDeclaration f) '''
        component «f.name» is abstraction() {
            «FOR g : f.componentBodyDecl»
                                «FOR p : g.ports»
                                        «p.compilePorts»
                                «ENDFOR»
```

```
                «ENDFOR»

        behavior is {}
      }
   '''

   def compilePorts(portDeclaration p) '''
        «{ currPort = p; "" }»
                connection «currPort.name» is «IF
(currPort.type.equals('provides'))»out«ELSE»in«ENDIF»(«currPort.interfaceImpl.compileTypes
»
   '''

//extract types from interfaces for ports & roles
   def compileTypes(interfaceDeclaration z) '''
      «z.interfaceBodyDecl.compileInterfaceBody»
   '''

   def compileInterfaceBody(interfaceBody z1) '''
          «FOR z2 : z1.interfaceBodyDeclaration»
              «z2.compileInterfaceMethods»
          «ENDFOR»
        '''

   def compileInterfaceMethods(interfaceMethodDeclaration z2) '''
        «z2.paramsDecl.compileInterfaceParams»
   '''

   def compileInterfaceParams(formalParameters z3) '''
        «z3.paramsDecl.compileInterfaceParams2»
   '''

   def compileInterfaceParams2(formalParameterList z4) '''
        «z4.paramDecl.compileInterfaceParams3»
   '''

   def compileInterfaceParams3(formalParameter z5) '''
      «switch z5.type {
        case 'int' : "Integer)"
        case 'type' : "Any)"
        default : "Could not get type...)"
```

```
        }»
    '''

//connectors, roles, etc...
    def compileConn(connectorDeclaration g) '''
        connector «g.name» is abstraction() {
            «FOR h : g.connectorBodyDecl»
                                        «FOR i : h.roles»
                                    «i.compileRoles»
                                        «ENDFOR»
                                    «ENDFOR»

            behavior is {}
        }
    '''

    def compileRoles(roleDeclaration j) '''

        connection «j.name» is «IF
(j.type.equals('provides'))»out«ELSE»in«ENDIF»(«j.interimpl.compileTypes»
    '''
}
```

Bibliography

[Aldrich et al., 2002a] Aldrich, J., Chambers, C. and Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering (ICSE '02), ACM, New York, USA, pp. 187–197.

[Aldrich et al., 2002b] Aldrich, J., Chambers, C. and Notkin, D. (2002). Architectural Reasoning in ArchJava. In: ECOOP 2002 — Object-Oriented Programming. ECOOP 2002. Lecture Notes in Computer Science, vol 2374. Springer, Berlin, pp. 334-367.

[Aldrich et al., 2003] Aldrich, J., Sazawal, V., Chambers, C. and Notkin, D. (2003). Language Support for Connector Abstractions. In: ECOOP 2003 – Object-Oriented Programming. Lecture Notes in Computer Science, vol 2743. Springer, pp. 74–102.

[Allen et al., 1998] Allen, R., Douence R. and Garlan, D. (1998). Specifying and analyzing dynamic software architectures. In: Fundamental Approaches to Software Engineering. FASE 1998. Lecture Notes in Computer Science, vol 1382. Springer, Berlin, Heidelberg.

[Allen, 1997] Allen, J. (1997). A Formal Approach to Software Architecture. PhD Thesis. School of Computer Science, Carnegie Mellon University.

[Amirat and Oussalah, 2009] Amirat, A. and Oussalah, M. (2009). First-Class Connectors to Support Systematic Construction of Hierarchical Software Architecture. In: Journal of Object Technology, 8(7), pp.107-130.

[Amundsen et al., 2016] Amundsen, M., McLarty, M., Mitra, R. and Nadareishvili, I. (2016). Microservice Architecture - Aligning Principles, Practices, and Culture. O'Reilly Media.

[ANTLR 2014] ANother Tool for Language Recognition 2014, Terence Parr, accessed 11 June 2019, <https://www.antlr.org/>

[antlr/codebuff 2013] GitHub - antlr/codebuff: Language-agnostic pretty-printing through machine learning 2013, accessed 11 June 2019, <https://github.com/antlr/codebuff>

[Architecture Analysis and Design Language 2015] Architecture Analysis and Design Language 2015, accessed 12 March 2019, <http://www.aadl.info/aadl/currentsite/>

[Barros, 2005] Barros, T. (2005). Formal Specification and Verification of Distributed Component Systems. PhD Thesis. Universite de Nice-Sophia Antipolis.

[Bass et al., 2013] Bass L., Clements, P. and Kazman, R. (2013). Software Architecture in Practice (SEI Series in Software Engineering), 3rd Edition, Addison-Wesley Professional.

[Batista et al., 2005] Batista, T., Joolia, A. and Coulson., G. (2005). Managing dynamic reconfiguration in component-based systems. In: Software Architecture. EWSA 2005. Lecture Notes in Computer Science, vol 3527, Springer, Berlin, pp. 1-17.

[Beck and Andres, 2004] Beck, K. and Andres, C. (2004). Extreme Programming Explained: Embrace Change (The XP Series), 2nd Edition, Addison-Wesley.

[Bernardo and Franze, 2002] Bernardo, M. and Franze, F. (2002). Architectural Types Revisited: Extensible And/Or Connections. In: Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol 2306. Springer, Berlin, Heidelberg, pp. 113-127.

[Bettini, 2016] Bettini, L. (2016). Implementing Domain-Specific Languages with Xtext and Xtend. Second Edition. Packt Publishing.

[Bonta and Bernardo, 2009] Bonta, E. and Bernardo, M. (2009). PADL2Java: A Java code generator for process algebraic architectural descriptions. In: Joint Working {IEEE/IFIP} Conference on Software Architecture 2009 and European Conference on Software Architecture 2009, (WICSA/ECSA), UK, pp. 161-170.

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

[Cavalcante et al., 2014] Cavalcante, E., Oquendo, F. and Batista, T. (2014). π-ADL: A Formal Description Language for Software Architectures. Technical Report - UFRN-DIMAp-2014-102-RT. Departamento de Informática e Matemática Aplicada. Universidade Federal do Rio Grande do Norte

[Cavalcante et al., 2015] Cavalcante, E., Batista, T. and Oquendo, F. (2015). Supporting Dynamic Software Architectures: From Architectural Description to Implementation. In: Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA '15). IEEE Computer Society, Washington, USA, pp. 31-40.

[Clements et al., 2011] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R. and Stafford, J. (2011). Documenting Software Architecture: Views and Beyond, 2nd ed., Addison-Wesley, USA.

[Clements, 1996] Clements, P. (1996). A survey of architecture description languages. In: Proceedings of the 8th International Workshop on Software Specification and Design, IWSSD ’96, Washington, USA.

[Dashofy et al., 2001] Dashofy, E.M., van der Hoek, A. and Taylor, R.N. (2001). A highly-extensible, XML-based architecture description language. In: Proceedings Working IEEE/IFIP Conference on Software Architecture, pp. 103-112.

[Dashofy et al., 2002] Dashofy, E.M., van der Hoek, A. and Taylor, R.N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In: Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, Orlando, USA, pp. 266–276.

[Delgado and Gonzalez, 2014] Delgado, A. and Gonzalez, L. (2014). Eclipse SoaML: A Tool for Engineering Service Oriented Applications. In: Pre-proceedings of International Conference on Advanced Information Systems Engineering (CAISE ’14) Forum, Thessaloniki, Greece.

[Donovan and Kernighan, 2016] Donovan, A. and Kernighan, B. (2016). The Go Programming Language. Addison-Wesley Professional Computing Series.

[Efftinge Spoenemann 2018] Efftinge, S. and Spoenemann, M. (2018). Xtext - Language Engineering Made Easy. Eclipse.org., accessed 11 May 2019, <https://eclipse.org/Xtext/>

[Enterprise Service Bus 2013] Enterprise Service Bus, Technical Article, Oracle Technology Network 2013, accessed 15 July 2019, <http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html>

[Erl et al., 2013] Erl, T., Puttini, R. and Mahmood, Z. (2013). Cloud Computing: Concepts, Technology & Architecture. Prentice Hall.

[Erl, 2016] Erl, T. (2016). Service-Oriented Architecture (paperback): Concepts, Technology, and Design (The Prentice Hall Service Technology Series from Thomas Erl). Prentice Hall.

[Feiler et al., 2006] Feiler, P., Gluch, D. and Hudak, J. (2006) The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report, CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, USA.

[Fowler, 2010] Fowler, M. (2010). Domain-Specific Languages. Addison-Wesley Professional, USA.

[Francesco, 2017] Francesco, P. (2017). Architecting Microservices. In: Proceedings of 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, pp. 224-229.

[Friedenthal et al., 2014] Friedenthal, S., Moore, A. and Steiner, R. (2014). A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Publishers Inc., 3rd Edition, San Francisco, CA, USA.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., Vlissides, J. and Booch, G. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

[Garlan et al., 1997] Garlan, D., Monroe, R. and Wile., D. (1997). ACME: An Architecture Description Interchange Language. In: Proceedings of CASCON 97, Toronto, pp. 169-183.

[Garlan et al., 2000] Garlan, D., Monroe, R. and Wile, D. (2000). Acme: Architectural Description of Component-Based Systems. In: Foundations of Component-Based Systems, Cambridge University Press, Springer-Verlag, London, UK, pp. 47-68.

[Granchelli et al., 2017] Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L. and Salle, A.D. (2017). Towards Recovering the Software Architecture of Microservice-Based Systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, pp. 46-53.

[Imperial College of Science, Technology and Medicine, 1997] Imperial College of Science, Technology and Medicine. (1997). The Darwin Language, Version 3d. Technical Report. Department of Computing.

[Kamal and Avgeriou, 2007] Kamal, A.W. and Avgeriou, P. (2007). 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.

[Keen et al., 2005] Keen, M., Adinolfi, O., Hemmings, S., Humphreys, A., Kanthi, H. and Nottingham, A. (2005). Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6, IBM RedBooks.

[Kephart and Chess, 2003] Kephart, J. and Chess, D. (2003). The Vision of Autonomic Computing. In: Computer 36 (1), pp. 41-50.

[Kotha, 2004] Kotha, S.P. (2004). xADL – A Better way to Describe Architecture, Mtech, CSE, IIT Kanpur.

[Kruchten, 1995] Kruchten, P. (1995). Architectural Blueprints — The “4+1” ViewModel of Software Architecture. In: IEEE Software 12 (6), pp. 42-50.

[Luckham, 1996] Luckham, D.C. (1996). Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford University, Stanford, USA.

[Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S. and Kramer, J. (1995). Specifying Distributed Software Architectures. In: Proceedings of the 5th European Software Engineering Conference. Springer-Verlag, London, UK, pp. 137-153.

[Magee et al., 1999] Magee, J., Kramer, J. and Giannakopoulou, D. (1999). Behaviour Analysis of Software Architectures. In: Software Architecture. WICSA 1999. IFIP — The International Federation for Information Processing, vol 12. Springer, Boston, MA.

[Malavolta et al., 2012] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P. and Tang, A. (2012). What industry needs from architectural languages: A survey. In: IEEE Transactions on Software Engineering.

[Mateescu and Oquendo, 2006] Mateescu, R. and Oquendo, F. (2006). π-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. In: SIGSOFT Softw. Eng. Notes 31, 2, pp. 1-19.

[Mayer and Weinreich, 2017] Mayer, B. and Weinreich, R. (2017). A Dashboard for Microservice Monitoring and Management. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, pp. 66-69.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R.N. (2000). A classification and comparison framework for software architecture description languages. In: IEEE Trans, Software Eng., 26(1), pp. 70–93.

[Microservice Communication Patterns 2018] Microservice Communication Patterns 2018, Tom Hombergs, accessed 20 July 2019, <https://reflectoring.io/microservice-communication-patterns/>

[Microservices 2014] Microservices: a definition of this new architectural term 2014, M. Fowler and J. Lewis, accessed 5 July 2019, <http://martinfowler.com/articles/microservices.html>

[Microservices Architecture 2019] Learn About the Microservices Architecture 2019, Oracle, accessed 5 July 2019, <https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html>

[Milner, 1999] Milner, R. (1999). Communicating and Mobile Systems: The Pi Calculus. Cambridge University Press.

[Minora et al., 2012] Minora, L., Buisson, J., Oquendo, F. and Batista, T.V. (2012). Issues of Architectural Description Languages for Handling Dynamic Reconfiguration. In: 6eme Conference francophone sur les architectures logicielles (CAL '12), Montpellier, France, pp. 69-80.

[Monroe, 1998] Monroe., R. (1998). Capturing Software Architecture Design Expertise with ARMANI. Technical Report CMU-CS-163, Carnegie Mellon University, Pittsburgh, USA.

[Muccini, 2013] Muccini, H. (2013). Lecture: Introduction to ADLs. DISIM, University of L’Aquila.

[Naumovich et al., 1997] Naumovich, G., Avrunin, S., Clarke, A., and Osterweil, J. (1997). Applying static analysis to software architectures. In: ESEC/SIGSOFT FSE, volume 1301 of Lecture Notes in Computer Science, pages 77–93. Springer.

[Newman, 2015] Newman, S. (2015). Building Microservices - Designing Fine-Grained Systems. O'Reilly Media.

[Online Shopping Process 2019] Online Shopping Process BPMN Template 2019, accessed 15 July 2019, <https://www.edrawsoft.com/template-online-shopping-process-bpmn.php>

[Oquendo, 2004] Oquendo, F. (2004). π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. In: ACM Software Engineering Notes Volume 29, Issue 3, pp. 1-14.

[Oquendo, 2008] Oquendo, F. (2008). π-ADL for WS-Composition: A Service-Oriented Architecture Description Language for the Formal Development of Dynamic Web Service Compositions. In: Proceedings of the Second Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2008), pp. 1-14.

[Ozkaya and Kloukinas, 2013] Ozkaya, M. and Kloukinas, C. (2013). Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. In: Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 177-184.

[Ozkaya, 2014] Ozkaya, M. (2014). A Design-by-Contract based Approach for Architectural Modelling and Analysis. Post-Doctoral Thesis. London City University.

[Ozkaya, 2016] Ozkaya, M. (2016). What is software architecture to practitioners: A survey. In: 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), Rome, Italy.

[Rozanski and Woods, 2005] Rozanski, N. and Woods, E. (2005). Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives, Addison-Wesley Professional.

[Saidane and Guelfi, 2013] Saidane, A. and Guelfi, N. (2013). Towards Test-Driven and Architecture Model-Based Security and Resilience Engineering. In: H. Singh and K. Kaur, ed., Designing, Engineering, and Analyzing Reliable and Efficient Software, pp. 163-188.

[Schwaber and Beedle, 2001] Schwaber, K. and Beedle, M. (2001). Agile Software Development with Scrum (Series in Agile Software Development). Pearson.

[SCIETEC 2010] SCIETEC 2010, SCIETEC: Modeling a Network protocol with UML / SysML, accessed 12 March 2019, <http://scietec.blogspot.com/2010/05/modeling-network-protocol-with-uml.html>

[Seco and Caires, 2002] Seco, J. and Caires, L. (2002). ComponentJ: The Reference Manual. Departamento de Informatica, Universidade Nova de Lisboa. Technical Report, UNL-DI-6-2002.

[Seco et al., 2008] Seco, J., Silva, R. and Piriquito, M. (2008). Component J: A component-based programming language with dynamic reconfiguration, Computer Science and Information Systems, Volume 5 (2), pp. 63–86.

[Seidl et al., 2015] Seidl, M., Scholz, M., Huemer, C. and Kappel, G. (2015). UML @ Classroom: An Introduction to Object-Oriented Modeling (Undergraduate Topics in Computer Science). Springer.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc., USA.

[Shaw et al., 1996] Shaw, M., DeLine, R. and Zelesnik, G. (1996). Abstractions and Implementations for Architectural Connections. In: Proceedings of the 3rd International Conference on Configurable Distributed Systems, Annapolis, USA, pp. 2-10.

[SoaML 2019] SoaML 2019, The Service Oriented Architecture Modeling Language Specification Version 1.0.1, accessed 12 March 2019, <https://www.omg.org/spec/SoaML/>

[Software Engineering 2019] Software Engineering 2019, accessed 12 March 2019, <https://softwareengineering.stackexchange.com/questions/310420/improving-the-design-of-a-simple-restaurant-client-server-architecture-uml-diag>

[SysML 2018] SysML 2018, SysML Open Source Project, accessed 12 March 2019, <https://sysml.org/>

[Taylor et al., 2009] Taylor, R., Medvidovic, N. and Dashofy E. (2009). Software architecture: Foundations, theory, and practice. Wiley, John & Sons, United Kingdom.

[The Acme Studio Homepage 2009] The Acme Studio Homepage 2009, accessed 25 March 2019, <http://www.cs.cmu.edu/~acme/AcmeStudio/index.html>

[TwoTowers 5.1 2009] TwoTowers 5.1 2009, M. Bernardo, accessed 10 March 2019, <http://www.sti.uniurb.it/bernardo/twotowers/>

[Urma et al., 2014] Urma, R.G., Fusco, M. and Mycroft, A. (2014). Java 8 in Action: Lambdas, Streams, and functional-style programming. Manning Publications.

[van Ommering et al., 2000] van Ommering, R.C., van der Linden, F., Kramer, J. and Magee, J. (2000). The koala component model for consumer electronics software. In: IEEE Computer, 33(3), pp. 78–85.

[Wright tools] Wright tools, accessed 25 March 2019, <http://www.cs.cmu.edu/afs/cs/project/able/www/wright/wright_tools.html>

[xADL Concepts and Info 2003] xADL Concepts and Info 2003, Eric M. Dashofy, accessed 9 May 2019, <http://isr.uci.edu/projects/archstudio-4/www/xarchuci/guide.html>

[XML Authority 2017] XML Authority 2017, XML Program for Reporting - XML Authority - Authority Software, accessed 12 March 2019, <https://authoritysoftware.co.uk/authority-suite/xml-authority/>

[XMLSpy 2019] XMLSpy 2019, XML Editor: XMLSpy | Altova, accessed 12 March 2019, <https://www.altova.com/xmlspy-xml-editor>

Declaration of Originality

I declare that this thesis, conducted for the purpose of receiving the academic title “Doctor of Philosophy” (PhD) in the field of 4.6 Informatics and Computer Science, entitled "*jADL, μσADL – Case Study of New Generation ADLs for Architecting Advanced Software Architectures*" is my work and in its development no foreign publications and developments have been used in violation of their copyright. Quotations of all sources of information, text, illustrations, tables, figures are marked by standards. The results and contributions of this thesis obtained are original and are not borrowed from research and publications in which I have no participation.