



Sofia University "St. Kliment Ohridski"

Faculty of Mathematics and Informatics

**jADL, $\mu\sigma$ ADL – Case Study of New Generation ADLs
for Architecting Advanced Software Architectures**

Synopsis

of the PhD Thesis

*Conducted for the purpose of receiving the academic title "Doctor of Philosophy"
(PhD) in the field of 4.6. Informatics and Computer Science*

Submitted by:

Anastasios G. Papapostolu

Advisor:

assoc. prof Dimitar Y. 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

1. Introduction

1.1 Software Architecture.....	1
1.2 Domain Specific Languages.....	2
1.3 Architecture Description Languages.....	2
1.4 Thesis Goals	4
1.5 Publications Related to Thesis	5
1.6 Thesis Structure.....	6

2. Related work

2.1 Introduction	7
2.2 Darwin	7
2.3 Wright.....	7
2.4 Rapide.....	8
2.5 ACME.....	8
2.6 Koala.....	8
2.7 XADL	9
2.8 AADL	9
2.9 π -ADL.....	9
2.10 PADL.....	10
2.11 Informal Languages.....	10
2.12 Conclusion.....	11

3. jADL

3.1 Introduction	14
------------------------	----

3.2 jADL Syntax.....	14
3.3 jADL Graphical Representation	20
3.4 Message Bus Architectural Pattern	21
3.5 Conclusion	26
4. $\mu\sigma$ADL	
4.1 Introduction	27
4.2 MicroService Architectures.....	27
4.3 $\mu\sigma$ ADL Constructs	27
4.4 Designing Microservices Using $\mu\sigma$ ADL and BPMN	29
4.5 Conclusion.....	34
5. Tool Support / Evaluation	
5.1 Introduction	35
5.2 Initial Tool – ANTLR.....	35
5.3 Tool Support.....	36
5.4 Case Study for jADL Evaluation	39
5.5 Conclusion.....	42
6. Conclusion	
6.1 Research Summary	43
6.2 Thesis Contributions	45
6.3 Future Work.....	46
Bibliography	47

List of Figures

Fig. 1. Usage of ADLs in industry – reprinted from a research conducted by (Muccini, 2013).....	4
Fig. 2. Different cases of 1-N communication in jADL	17
Fig. 3. 1-N communication of three different threads in jADL	17
Fig. 4. Graphical notations in jADL	21
Fig. 5. Architecture of the Message Bus Architectural Pattern.....	23
Fig. 6. Inner components of <i>MessageBus</i>	26
Fig. 7. Online shopping process in BPMN – reprinted from (Online Shopping Process 2019).	30
Fig. 8. Graphical representation in jADL of the server component.....	32
Fig. 9. Abstract syntax tree from the description presented in a textual way	35
Fig. 10. (a) error detection, (b) auto-completion... ..	36
Fig. 11. Reprinted and extended from (Cavalcante et al. 2014).....	37
Fig. 12. Graphical representation of the Gas Station system	39

List of Code Snippets

Code Snippet 1. Client component description in jADL	22
Code Snippet 2. Connector (MBAP) description in jADL	24
Code Snippet 3. MBAP description in jADL	25
Code Snippet 4. The translated, in jADL, component and connector.....	29
Code Snippet 5. MicroServices description in $\mu\sigma$ ADL	31
Code Snippet 6. jADL description of the <i>Inventory</i> microservice.....	31
Code Snippet 7. Server description in $\mu\sigma$ ADL	33
Code Snippet 8. Interfaces for the Gas Station system	39
Code Snippet 9. Customer component description	40
Code Snippet 10. Cashier component description	40
Code Snippet 11. Pump component description	41
Code Snippet 12. Architecture of the Gas Station system	41

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. 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). 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.

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 – logical, implementation, process and deployment. The plus-one view consists 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. In this thesis, we will be dealing with the first approach mentioned, the one in (Clements et al., 2011).

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. 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, showing how the software structures correspond to the environment’s structures.

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. *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 changes are performed without a human assistance the architecture is called *autonomic*, or *self-** (Kephart and Chess, 2003).

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. 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. 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.

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.

The main building blocks in most ADLs are *components*, *connectors* and *configurations*. ADLs provide the possibility for both describing the structural specification and the behavioral aspects of the given architecture. ADLs usually present the architecture of a software system in a visual way, as a graph of interconnected elements, 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. 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 reconfigurations.

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) do not consider connectors as first-class entities. Inside components, apart from the computations, the coordination is entangled too.
- *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.

- *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.

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).

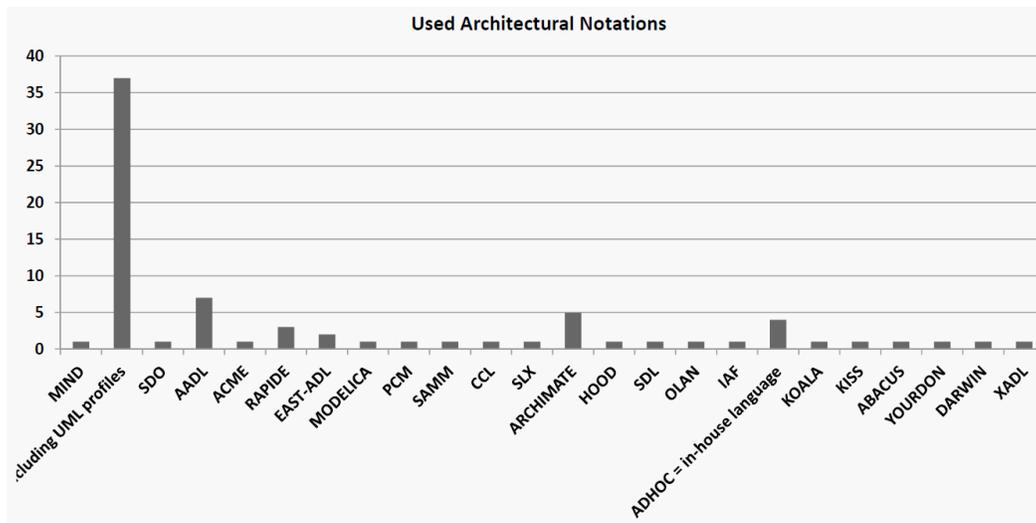


Fig. 1. *Usage of ADLs in industry* – reprinted from a research conducted by (Muccini, 2013).

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.

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 today's 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 $\mu\sigma$ ADL*, 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 $\mu\sigma$ ADL (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 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 formal architecture description languages and informal languages is presented. There is a big number of architecture description languages (120+). A number of researches have been conducted regarding them, as can be seen for example in (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. 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. Other important factors I took into account were their industrial usage and their focus regarding the descriptions (e.g. structural or runtime). Additionally, languages that focus on different capabilities were chosen, like for example AADL (analysis) and Xadl (extensibility). Finally, languages that influenced the 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. Next, the obtained subset of architecture description languages is presented. In the conclusion section, aggregated results regarding the languages examined are shown and the main reasons for the creation of an additional language are outlined.

2.2 Darwin

Darwin (Magee et al., 1995), 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. Connectors are not considered to be first-class entities in Darwin, so there is no such specification when describing an architecture. The interaction mechanisms are encapsulated inside the components, thus making them more complex and harder to reuse.

2.3 Wright

Wright (Allen, 1997) is an Architecture Description Language that, follows the component / connector / configuration style for describing architectures. Components in Wright express

independent computations and are defined in two main sections. 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 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.

2.4 Rapide

Rapide (Luckham, 1996) is an architecture description language with a focus on dynamic architectures and the simulation of architectures. 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. 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 specifications.

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. It follows the classic component/connector/system paradigm and considers connectors as first-class entities. 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. 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 Acme.

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 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. Despite the existence of interfaces, the lack of connectors as first-class entities doesn't allow to describe complex communication mechanisms. Interactions are simply described in the connects section of the declaration of a composite component.

2.7 XADL

xADL (Dashofy et al., 2001) is highly extensible and flexible xml-based architecture description language. Using XML schemas takes advantage and integrates in the language their high interchangeability and modularity. So, it provides the possibility for easy reuse and easy creation of features which can extend the language. 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 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. Additionally, a number of tools has, also, been developed to support the language, like ArchEdit (Kotha, 2004). Due to the multiple schemas specified the actual architectural description can get quite complicated, so non-XML notation can be used too.

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 (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 to choose from, when defining the architecture. There are three categories (Feiler et al., 2006): i) application software, ii) execution platform, and iii) composite, which consists of system types for the specification of composite types. AADL does not offer first-class connectors and defines interfaces, through which the communication between components occurs.

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. It considers both components and connectors first-class entities. Each of these architectural elements is defined in two parts. First the connections are declared. Next, the behavior of each element is described with the use of simple statements. Finally, the architecture is created with the declaration of the instances and their interconnections.

Furthermore, there is a software tool (Cavalcante et al., 2015) for the generation of the executable programming code in the GO (Donovan and Kernighan, 2016) programming language. Finally, it provides the constructs needed for the successful expression of dynamic and mobile architectures and, as seen in (Minora et al., 2012), can support (though the use of other languages might be required) dynamic reconfigurations.

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 and its interactions. 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.

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. Finally, PADL2Java (Bonta and Bernardo, 2009) is a software tool built to translate PADL models into Java implementation code stubs.

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 the previous languages are omitted here. 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. 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 the *dynamic* view is used to represent the behavior of a system during run-time. 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* and *required*.

Connectors are not defined as first-class entities in UML and the interactions between components are modeled as simple communication channels between their ports (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*. 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. *Components* are special kind of objects in ArchJava and their communication is ensured through the definition of *ports*. Ports can declare three sets of methods (Aldrich et al., 2002b): *requires*, *provides* and *broadcasts*. In ArchJava, at first, connectors were not first-class elements, but instead the primitive *connect* was used to connect two or more ports. An extension was created (Aldrich et al., 2003), in order to provide connector abstractions.

2.11.4 SysML

SysML (Friedenthal et al., 2014) is a general-purpose modeling language for systems engineering applications (SysML 2018). It is created as an extension to the UML and introduces new fixtures. Components in SysML are expressed through *blocks* which are connected to each other with *ports*. The language supports behavioral specification, but connectors are not first-class entities.

2.11.5 SoaML

SoaML (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*. Also, a software tool has been implemented.

2.12 Conclusion

In the table below, aggregated data is presented. DNA stands for "does not apply". It is used for the informal languages in the columns Generation (classification includes only formal languages) and Dynamic Reconfiguration (out of the scope of this thesis).

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

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, 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 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. The practical use of the language and its various constructs is illustrated through the description of 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 architectural elements are presented in the sections that follow.

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.

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. 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.

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 element. 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 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*, *kind* and *multiplicity* and *synchronicity* of their connection. These are, also, the factors that define whether an attachment will be successful or not; the two interfaces must be *compatible*, their kinds must be *opposing* and of the *same synchronicity*. 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.

3.2.3.1 Kind

In the declaration of ports and roles, the keywords *provides* and *requires* are used to declare their *kind*. Every port or role must have a kind. 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.

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. Figure 2 provides cases where attachments have more than two architectural elements involved.

While the attachment in 2.a is successful, the one in 2.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.

3.2.3.3 Synchronicity

In jADL, when declaring a port or a role, 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 3 is illustrated this case; the two connectors, each executed in a different thread, might attempt to gain access to the same resource, so concurrency issues will arise. 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, the code should be modified; the keyword *synchronized* must be added to both the port *p* and the roles *r1*, *r2* declarations.

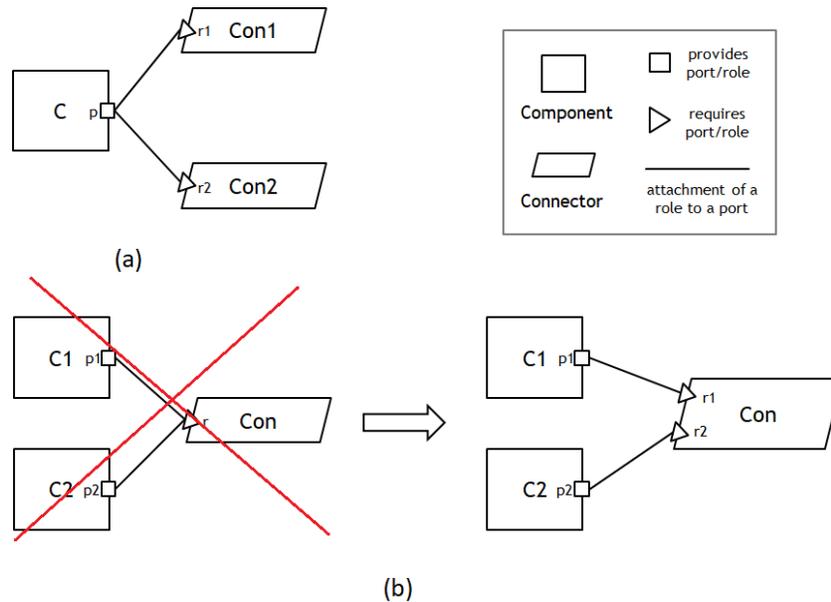


Fig. 2. Different cases of 1-N communication in jADL.

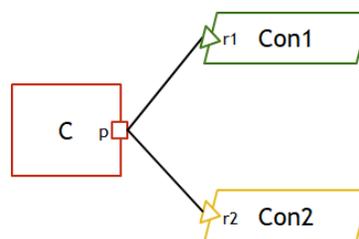


Fig. 3. 1-N communication of three different threads in jADL.

3.2.4 Interfaces

Interfaces are used to define the shape of communication and the behavior of a port or role. 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)*.

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).

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.

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 and the behavior definition.

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.

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. Similar to *attach*, it can accept the same pairs of arguments. These two statements can be used both when defining an architecture and at run-time for dynamically reconfiguring the architecture.

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.

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.

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*”.

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. The only restriction concerning this statement is that *the arguments in the bind statement must be of the same kind*.

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.

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.

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, 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.

3.3 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.

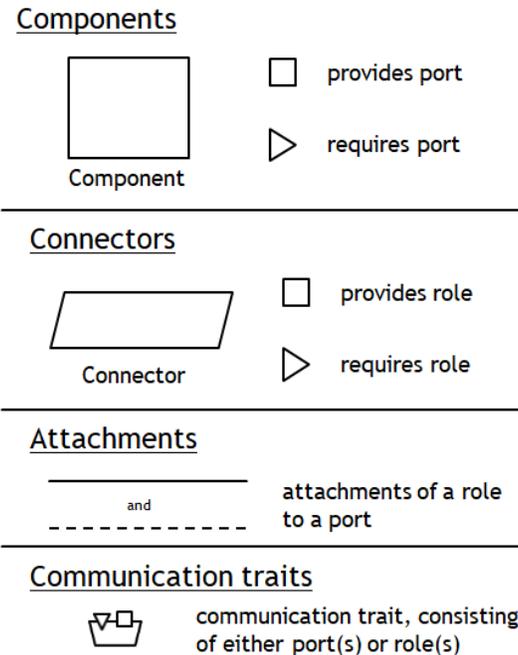


Fig. 4. Graphical notations in jADL.

3.4 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 5.

jADL Client Component Description

```
1.  type Message;

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

12. trait CommTrait {
13.   provides port IReceiveMsg p;
14.   requires port ISendMsg r;
15.  }

16. component C {
17.   requires port ISubscribe s;
18.
19.   trait CTrait aggregate CommTrait {
20.     config p as {
21.       service void getMsg(Message msg){
22.         // process the Message
23.         display(msg);
24.       }
25.     } }

26.   CTrait com1 = new CTrait();
27.   while(true) {
28.     com1.r.sendMsg(com1, "new message");
29.     delay 10;
30.  } }
```

Code Snippet 1. *Client component description in jADL.*

Line 1 in code snippet 1 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.

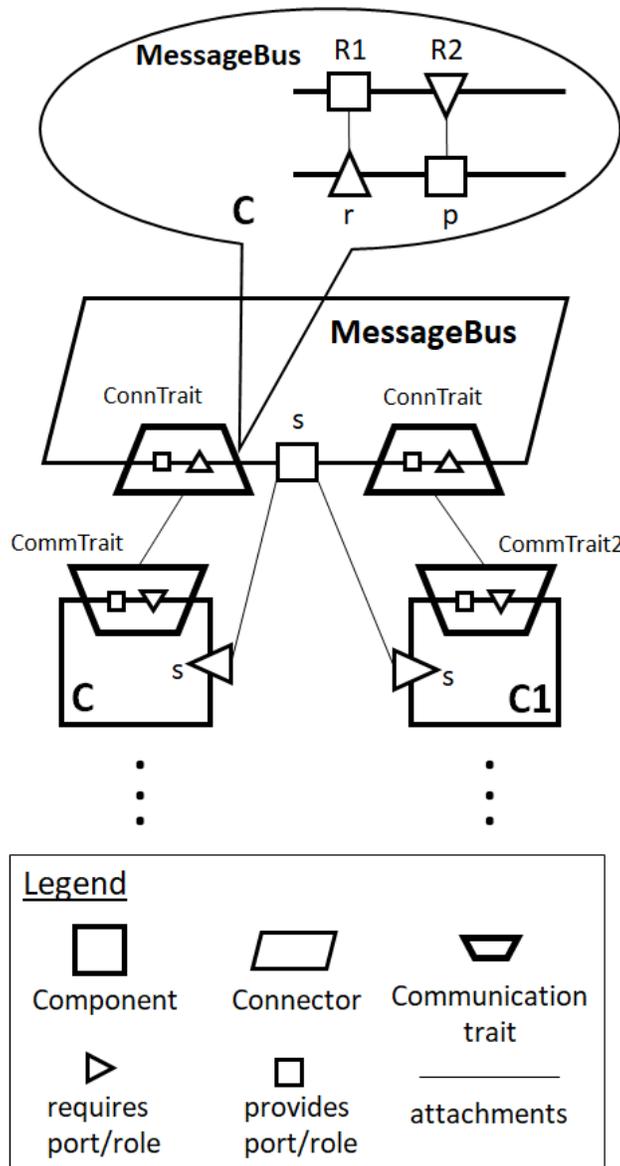


Fig. 5. Architecture of the Message Bus Architectural Pattern.

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 aggregates 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.   delay 20;
41. } }

```

Code Snippet 2. 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 6.

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); }
```

Code Snippet 3. MBAP description in jADL.

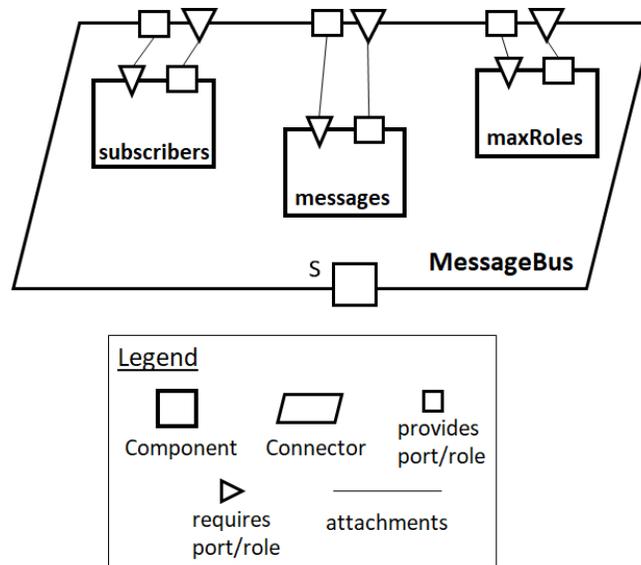


Fig. 6. Inner components of *MessageBus*.

In code snippet 3 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.

3.5 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 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, a case study is presented in the final section. The description of the architecture of a Message Bus, a widely used architectural pattern for integrating a varying number of heterogeneous applications, is shown. The language constructs provided by jADL (especially the use of communication traits) proved adequate to describe the system and showed the flexibility of the language when it comes to expressing dynamic reconfigurations.

Chapter 4

$\mu\sigma$ ADL

4.1 Introduction

In this chapter the extension created for jADL, named $\mu\sigma$ ADL, 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.

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 with very promising results. There is, also, a rapidly growing number of studies concerned with various aspects of microservice architectures, e.g. (Mayer and Weinreich, 2017). 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*".

As it is pointed out in (Francesco, 2017) there is a lack of 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.

4.3 $\mu\sigma$ ADL Constructs

$\mu\sigma$ ADL 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 $\mu\sigma$ ADL 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 and they are configured (if necessary) using the *config* statement.

Each microservice represents a computational and data store element. Thus, it represents a component when it is translated from $\mu\sigma$ ADL to jADL and all the statements/operators/etc. defined in jADL for components can be used when defining a microservice.

4.3.1 Communication Between Microservices in $\mu\sigma$ ADL

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 or messaging over a lightweight message bus (Microservices 2014).

In $\mu\sigma$ ADL 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 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. Alongside with the use of predefined integrated in jADL communication traits presented in the previous chapter, varying types of buses can be described. They can be called directly, instantiated and used in the architectural description.

4.3.2 Data Storage in $\mu\sigma$ ADL

In a microservice architecture, when it comes to persistent storage, as mentioned, it is favored a decentralized data management approach. Each microservice manages its own database.

Following this principle, in $\mu\sigma$ ADL 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. Using this simple description, we can then automatically generate the appropriate connector in jADL. For example, assuming we have a microservice, located in the same location with its database (*localhost*) and we need a JDBC standard connector. The description in $\mu\sigma$ ADL would be:

```
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:

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

23. component DB {
24.  provides port IConnJDBC pDB;
25.  requires port IConnJDBC rDB;
26.  config pDB as {
27.    service void sendQuery (sqlString data) {
28.      rDB.getQuery(data);
29.      //process the query and send reply
30.    }
31.  }
```

Code Snippet 4. *The translated, in jADL, component and connector.*

4.4 Designing Microservices Using $\mu\sigma$ ADL 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 $\mu\sigma$ ADL and jADL, consisting of 3 parts:

- i) The extraction of an initial architectural sketch from a BPMN representation where each process can be modeled as a microservice.

- ii) The specification of the architecture using $\mu\sigma$ ADL. 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 automatic translation of the $\mu\sigma$ ADL 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 figure 7, a simple process of online shopping through a site is presented.

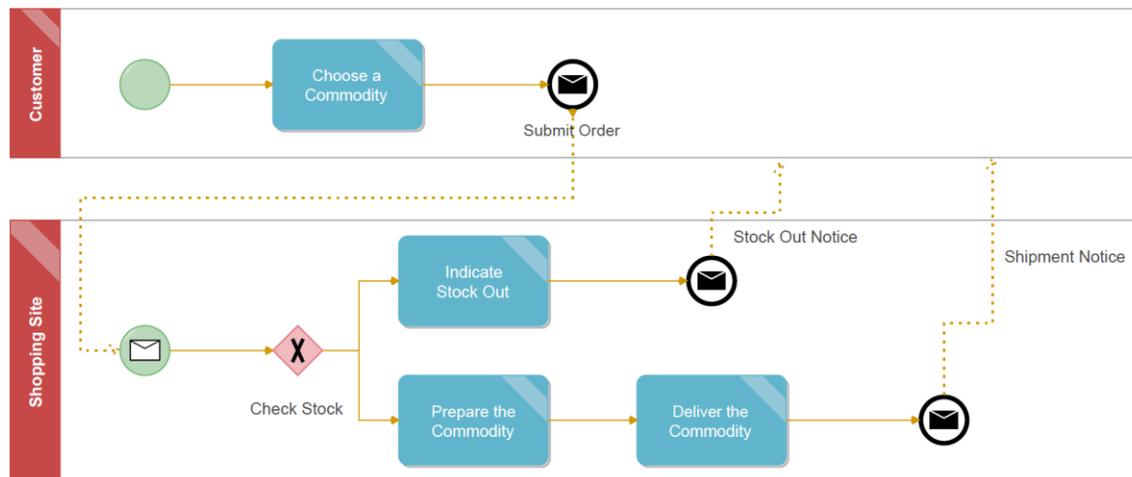


Fig. 7. *Online shopping process in BPMN* – reprinted from (Online Shopping Process 2019).

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. Using the microservices architectural style 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.

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 second step of the process proposed concerns the description of the architecture in $\mu\sigma$ ADL. The description of the microservice *shipping* is presented in code snippet 5. 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 language constructs provided by $\mu\sigma$ ADL proved to be adequate for the description of each microservice and their communication mechanisms. Rigorous and too formal semantics are "hidden" in $\mu\sigma$ ADL and the architect can define the architecture in a simple and elegant way.

During the third step of the process a translation of the description from $\mu\sigma$ ADL 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 (partially) seen in code snippet 6.

$\mu\sigma$ ADL 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.sendMessage(com1, data);
21.    }
22.  }
23. }
```

Code Snippet 5. *MicroServices* description in $\mu\sigma$ ADL.

jADL description

```
1. component Inventory {
2.   requires port ISubscribe r;
3.   trait InvTrait aggregate CommTrait {
4.     config p as {
5.       service void getMsg (type msg) {
6.         reply(msg);
7.       } } }
8.   instance com1 = new CTrait();
9.   config pDB as {
```

```

10.     service void getQueryRes (type data) {
11.         com1.r.sendMsg(com1, data);
12.     } } }
13. component DBInventory {
14.     provides port IConnJDBC pDB;
15.     requires port IConnJDBC rDB;
16.     config pDB as {
17.         service void sendQuery (sqlString data) {
18.             //process the query and send reply
19.             rDB.getQuery(data);
20.         } } }
21. component InventoryCont {
22.     requires port ISubscribe r;
23.     instance inv = new Inventory();
24.     instance conn = new ConnJDBC();
25.     instance dbinv = new DBInventory();
26.     attach(inv.com1.r, conn.pClient);
27.     attach(inv.com1.p, conn.rClient);
28.     attach(dbinv.pDB, conn.rDB);
29.     attach(dbinv.rDB, conn.pDB);
30.     bind(r, inv.r); }

```

Code Snippet 6. *jADL* description of the Inventory microservice.

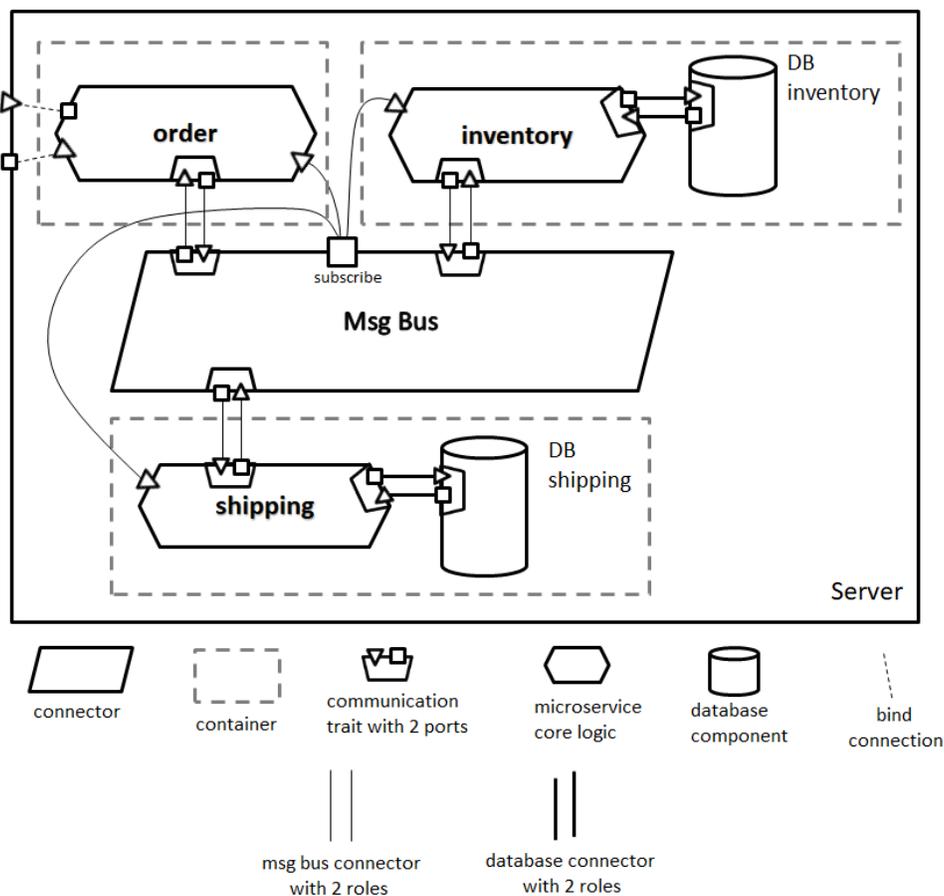


Fig. 8. *Graphical representation in jADL of the server component.*

In order for the microservices to communicate the message bus architectural pattern (MBAP), described in the previous chapter of this thesis, was chosen. 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 5.

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.

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

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 $\mu\sigma$ ADL 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();    }
```

Code Snippet 7. *Server description in jADL.*

4.5 Conclusion

The extension of jADL, named $\mu\sigma$ ADL, was presented in this chapter. An extension designed in order to ease the description of software systems that follow the microservices architectural style.

One of the main goals of $\mu\sigma$ ADL 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. As for the communication between microservices, communication traits and the MBAP 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 $\mu\sigma$ ADL. 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, by following the three steps of the process proposed this can be achieved. With the use of a series of simple and elegant statements in $\mu\sigma$ ADL, a detailed formal representation of the architecture in jADL can be obtained. Formal definitions regarding simple architectural elements 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 promotion of the use of architectural languages in industrial software implementation processes, since BPMN diagrams are widely used in practice.

Though simple cases of dynamic reconfigurations can be treated, as shown in this chapter, 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 $\mu\sigma$ ADL 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

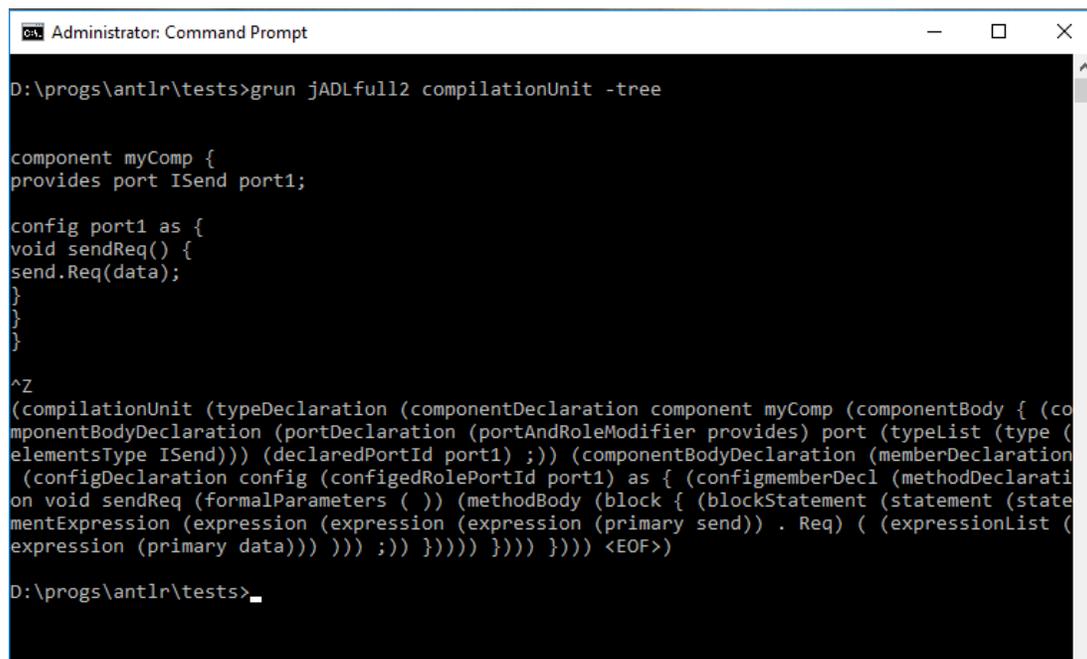
Tool Support / Evaluation

5.1 Introduction

In the previous chapters jADL and its extension for microservices, $\mu\sigma$ ADL, 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. Two different frameworks were used during the time of this research; first ANTLR and then Xtext. Both of them are presented alongside with a case study for the evaluation of the language.

5.2 Initial Tool – ANTLR

The first parser that was created during this research was built using ANTLR (ANTLR 2014). As an input it requires the definition of a grammar using an EBNF form. The grammar defined at the time does not fully correspond to the final version of the grammar presented in the previous sections, since ANTLR was used early in this research while still experimenting with the grammar.



```
Administrator: Command Prompt
D:\progs\antlr\tests>grun jADLfull2 compilationUnit -tree

component myComp {
  provides port ISend port1;

  config port1 as {
  void sendReq() {
  send.Req(data);
  }
  }
}

^Z
(compilationUnit (typeDeclaration (componentDeclaration component myComp (componentBody { (co
mponentBodyDeclaration (portDeclaration (portAndRoleModifier provides) port (typeList (type (
elementsType ISend))) (declaredPortId port1) ;)) (componentBodyDeclaration (memberDeclaration
 (configDeclaration config (configuredRolePortId port1) as { (configmemberDecl (methodDeclarati
on void sendReq (formalParameters ( )) (methodBody (block { (blockStatement (statement (state
mentExpression (expression (expression (expression (primary send)) . Req) ( (expressionList (
expression (primary data))) ))) ;)) })))) }))) <EOF>)
```

Fig. 9. Abstract syntax tree from the description presented in a textual way.

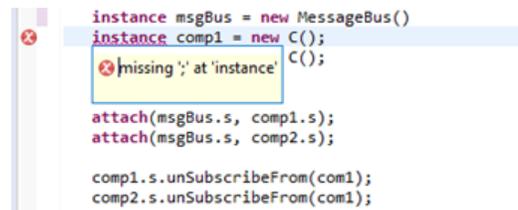
Upon the definition of the grammar, ANTLR can automatically, generate a parser that can build and walk parse trees. It offers different options for the analysis of the abstract syntax tree. In our approach we used the Visitor Design pattern (Gamma et al., 1994). After the compilation of the created files, using a command line interface we can type the jADL architectural description and view the generated abstract syntax tree. 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.

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 framework supports, also, the writing of programs in Xtend (Bettini, 2013), a Java-like language that can be used for adding additional code/functionality to grammar rules.

5.3.1 Editor

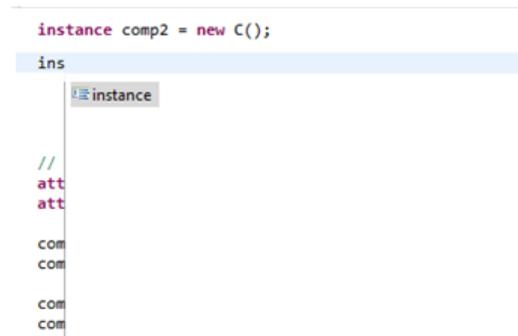
After defining the grammar for jADL the Eclipse editor was generated. It automatically supports typical functionalities for editors (e.g. auto-completion).



```
instance msgBus = new MessageBus()
instance comp1 = new C();
missing ';' at 'instance' C();
attach(msgBus.s, comp1.s);
attach(msgBus.s, comp2.s);

comp1.s.unsubscribeFrom(comp1);
comp2.s.unsubscribeFrom(comp1);
```

(a)



```
instance comp2 = new C();
ins
instance
//
att
att
com
com
com
com
```

(b)

Fig. 10. (a) error detection, (b) auto-completion.

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 11, 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 (<i>goroutine</i>)
Connector	Connector	Function (<i>goroutine</i>)
Behavior	Behavior	Body of function (<i>goroutine</i>)
Connection	Connections (<i>ports/roles/interfaces</i>)	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. 11. 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. 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 *architecture* declaration corresponds to *Main Function* in the GO language.

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. In order for the generator to work, the syntax of the resulting π -ADL description must be syntactically correct. We should note a significant difference between the generated code and the one in (Cavalcante et al., 2014) - the lack of *protocols*. 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.

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 12. 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.

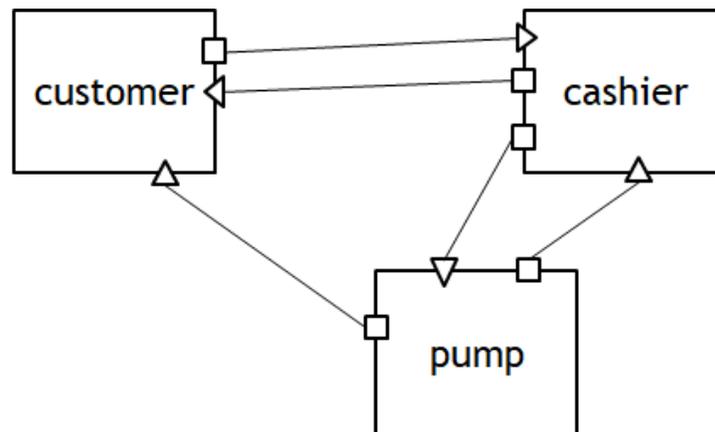


Fig. 12. Graphical representation of the Gas Station system.

First, the interfaces used, are declared in code snippet 8. 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 8. Interfaces for the Gas Station system.

The *customer* component, shown in code snippet 9, 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 9. *Customer component description.*

Continuing with the *Cashier* component (code snippet 10), 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.
7.     config pCust as {
8.         service int payment(float amnt) {
9.             if (check(amnt))
10.                return rPump.getPump();
11.         }
12.
13.     config pPump as {
14.         service boolean checkOrder(int custid) {
15.             if (check(custid))
16.                return true;
17.             else
```

```

17.             return false;
18.         }
19.     } }

```

Code Snippet 10. *Cashier component description.*

The *pump* component is shown next, in code snippet 11. 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 11. *Pump component description.*

Next, the architecture instantiation is presented in code snippet 12. 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. 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 12. *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 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. $\mu\sigma$ ADL 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. $\mu\sigma$ ADL 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 $\mu\sigma$ ADL 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 $\mu\sigma$ ADL 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.

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://reflecting.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-architecture-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>>