Софийски Университет “Св. Климент Охридски”

Факултет по математика и информатика

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

**Автореферат**

на

дисертационен труд

*Създаден с цел получаване на академична титла “Доктор” (д-р) в професионално направление 4.6 „Информатика и компютърни науки“*

Автор:
**Анастасиос Г. Папапостолу**

Научен ръководител:
**доц. д-р. Димитър Й. Биров**

София, 2019

# Благодарности

Искам да изкажа своите най-големи благодарности на всички, които ме подкрепяха и допринесоха за успешния финал на тази дисертация.

На покойния доцент Димитър Биров, който ми оказа огромна подкрепа и намери начини за решаването на всякакви проблеми с които се сблъсках по време на следването ми, но не можа да го види завършено.

На доцент Александър Димов, който предложи ценни съвети за финалните корекции и помогна за завършването на дисертацията.

На всички служители и преподаватели във Факултета по Математика и Информатика на Софийския Университет за непрекъснатата им подкрепа през годините на моята докторантура.

И накрая, искам също да благодаря на всички членове на комисията за тяхната помощ и съдействие.

# Съдържание

# Списък на фигурите

# Списък на фрагментите от код

# Глава 1

# Въведение

## 1.1 Софтуерна архитектура

Софтуерната архитектура (Shaw and Garlan, 1996; Bass et al., 2013) се е развила през последните десетилетия и се е превърнала в основна инженерна дисциплина. Формалното определение, което според мен най-добре описва значението на софтуерната архитектура е „*набор от структури, необходими за описание на софтуерна система, които съдържат софтуерни елементи, връзки между тях и техните свойства*“ (Clements et al., 2011). Важен аспект на софтуерната архитектура е адекватната документация на архитектурата на дадена система, така че тя да може да се използва по време на проектирането, на процеса на разработка, както и по време на развитието/поддръжката на системата.

Тъй като софтуерните системи стават все по-сложни, един от начините да се създаде ефективна документация е тя да бъде разделена на три части – наречени *перспективи*, всяка от които е придружена от редица изгледи (Clements et al., 2011). Тези три перспективи са *статичната*, *динамичната* и перспектива на *разпределението*. Всяка от тях е обяснена и представена в следващите параграфи на този раздел. Съществуват и други подходи, които се отнасят до успешната документация на софтуерните архитектури като Rational Unified Process (RUP) – пет изгледен (4+1) подход, базиран на класификацията, предложена от Kruchten (Kruchten, 1995).Състои се от четири основни части – логическа, изпълнение, процес и внедряване. Допълнителния изглед се състои от различни случаи на използване и сценарии, свързани с поведението на софтуерната система. Друг подход е този на Rozanski и Woods (Rozanski and Woods, 2005), където те предлагат набор от шест изгледа на документацията на софтуерната архитектура. В тази теза ще обърнем внимание на първия подход, споменат в (Clements et al., 2011).

Първата предложена перспектива е *статичната* перспектива на система. Тя засяга статичните части на системата и помага на архитектите да разсъждават как са структурирани единиците за внедряване на системата. Втората предложена перспектива е перспективата на *разпределение* (или *внедряване*). Тук е описана средата, в която ще бъде внедрена системата, включително зависимостите на системата от средата на изпълнение, показвайки как софтуерните структури съответстват на структурите на средата.

Третата предложена перспектива е *динамичната* перспектива на една система. Тя описва поведението на системата по време на изпълнение, как структурирания набор от елементи взаимодействат динамично един с друг по време на изпълнението на системата. Едно от най-важните неща в нея е изгледът *Компонент-и-Конектор* (*К&К*), където компонентите и конекторите са съставните елементи и са представени техните взаимовръзки, поведение и ограничения. *Компонентите* са елементите за изчисляване и съхранение на данни като комуникацията между тях и средата се осъществява само чрез декларирани портове. Те могат да комуникират помежду си и със средата чрез конектори. *Конекторите* представляват различните форми на комуникация между компонентите или комуникацията на елементите със средата, в която се намират, а декларираните им роли (съответно към портовете на компонента), са техните точки на взаимодействие.

Установява се връзка когато роля на конектор е прикачена към порт на компонент. Архитектурните елементи, техните взаимовръзки и ограничения, които ги засягат, съставляват *топологията* на софтуерната архитектура. Тя може да бъде формализирана като граф на компоненти и конектори, свързани помежду си чрез дъги. Поведението на компонентите и конекторите предоставя на дизайнерите информация за техните функционалности, потока на данни, начина по който те комуникират помежду си и т.н. Взаимодействието между комуникацията, потока на данни и връзката компонент-конектор описва поведението на софтуерната архитектура според топологията. Ако топологията или поведението на системата се променят по време на изпълнение, архитектурата се обозначава като *динамична* или *мобилна*. Когато тези промени се осъществяват без човешка намеса, архитектурата се нарича автономна или само-адаптивна (Kephart and Chess, 2003).

## 1.2 Домейн специфични езици

Домейн специфичните езици (Fowler, 2010) (DSLs) са компютърни езици с обикновено ограничена експресивност, създадени специално за справяне с конкретен набор от проблеми в определена област, за разлика от Езиците с Общо Предназначение, които могат да се използват в множество домейни. Домейн специфичните езици се разделят на две главни категории: *вътрешни* (или вградени) и *външни*.

- Вътрешните домейн специфични езици се дефинират с помощта на хост език, за да се даде различно „усещане“ на езика и да се използва по-стандартизиран и по-лесен начин. Основно предимство е, че езикът на хоста покрива нуждите по отношение на граматиката и парсера и могат да се възползват от съществуващи инструменти.
- Външните домейн специфични езици са изградени от основата и изискват персонализиран анализатор за превеждане на синтаксиса в нещо, което компютърът разбира и може да използва. Тъй като са независими от други съществуващи езици, те осигуряват голяма гъвкавост за дефиниране на граматиката по отношение на синтаксиса, операторите, структурата и т.н.

## 1.3 Езици за описание на архитектури

Езиците за описание на архитектури (Medvidovic and Taylor, 2000) (ADLs) са домейн специфични езици, които се използват в областта на софтуерната архитектура и софтуерното инженерство с цел формалното описание на системните архитектури. Те притежават високо ниво на абстракция и обикновено игнорират детайлите при имплементиране от по-ниско ниво. Чрез използването на формални методи, те успяват да верифицират, валидират и да гарантират синтактична и семантична коректност на софтуерната архитектура. Обикновено са предоставени инструменти за да се изпълняват различни действия с архитектурното описание, като симулация, генериране на софтуерни артефакти (напр. части от програмен код) и т.н.

Има много разработени през годините (повече от 120) езици за описание на архитектури, фокусирани върху различни домейни и насочени към различни проблеми. Тъй като има голяма променливост в изискванията на заинтересованите лица в различни сфери, малко вероятно е един единствен език за описание на архитектури, да се справи с всички тях. Следователно тези езици са склонни да се съсредоточат върху проблемите на конкретната област, като предлагат различни варианти за описание/валидиране/анализ и т.н. на архитектурата на софтуерна система. Например Wright (Allen, 1997) предоставя средства за определяне на сложни механизми на взаимодействие.

Основните градивни елементи в повечето такива езици са *компоненти*, *конектори* и *конфигурации*. Езиците за описание на архитектури осигуряват възможност както за описание на структурната спецификация, така и на поведението на дадената архитектура. Обикновено представят архитектурата визуално като графика на взаимосвързани елементи, което представлява топологията на системата.

### 1.3.1 Категории на Езици за описание на архитектури

Съществуващите езици за описание на архитектури могат да бъдат подредени по различни начини и са проведени проучвания по този въпрос, като едно от най-характерните е рамката за класификация и сравнение на (Medvidovic and Taylor, 2000).

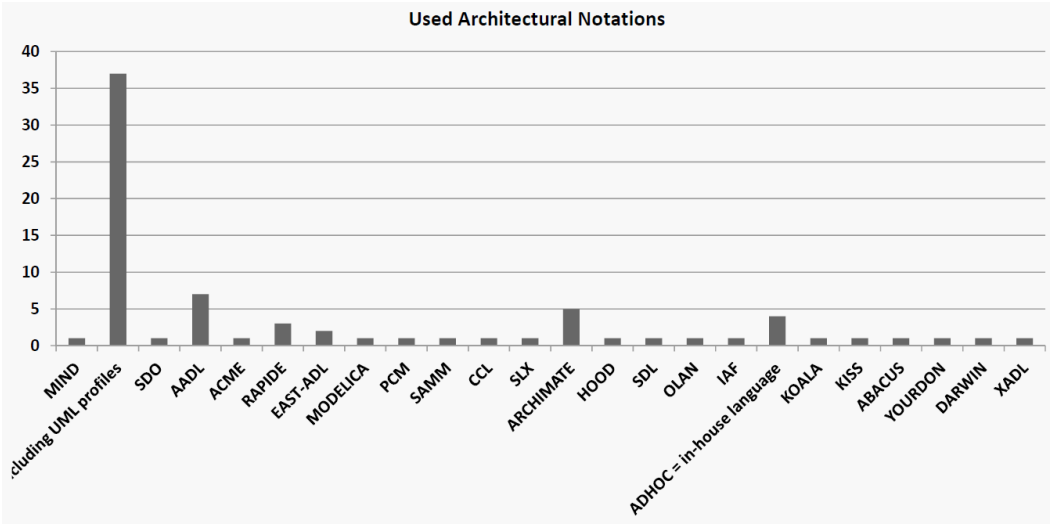
Най-често срещания начин за разделяне на тези езици е на първо и второ поколение. Има две категории – преди 1999 и след 1999. Важен проблем който адресират езиците от второ поколение е описанието на поведението на архитектурата по време на изпълнение и необходимостта от динамични реконфигурации.

Конекторите представляват важни архитектурни структури които ADLs трябва да дефинират като първокласни елементи. Следователно важна таксономия по отношение на езиците за описание на архитектури е тази, която ги класифицира според това дали поддържат дефиниране на конектори. Тази таксономия е предложена от (Amirat and Oussalah, 2009):

- *Езици за описание на архитектури с неявни конектори*. Те не поддържат конектори, защото нарушават композиционната природа на софтуерната архитектура. Езици като Darwin (Magee et al., 1995) не считат конекторите за първокласни единици. Вътре в компонентите, освен изчисленията, координацията също е описана.
- *Езици за описание на архитектури с предварително определен набор от конектори*. UniCon (Shaw et al., 1996) е пример за такъв език. Конекторите са дефинирани предварително и са вградени в езика. Въпреки че повторната използваемост е подобрена в сравнение с предишната категория, все още има ограничения.
- *Езици за описание на архитектури* с *явни типове конектори*. Повечето езици за описание на архитектури попадат в тази категория, като разглеждат конекторите като първокласни единици на езика. Изчисленията са описани вътре в компонентите и конекторите описват механизма на взаимодействие между тях, като по този начин разделят изчисленията от координацията и насърчават повторната използваемост.

### 1.3.2 Използване на Езици за описание на архитектури

Въпреки големия брой и предимствата на езиците за описание на архитектури, особено по време на проектиране, използването им извън академичните среди все още е ограничено. Що се отнася до практиката, високата степен на формалност на тези езици ги прави трудни за изучаване и за интегриране в индустриалните процеси. Предпочитат се неформални езици, като UML или инструменти за чертаене на диаграми от типа box-line, както е посочено от редица проучвания като (Ozkaya, 2016; Malavolta et al., 2012).




Фиг. 1. *Използване на ADLs в индустрията* – от изследване на (Muccini, 2013).

Друг проблем на тези езици е тяхната поддръжка относно динамични реконфигурации както е посочено от (Minora et al., 2012). Освен това поддръжката на инструментите за тези езици е сравнително слаба, особено в сравнение с езиците за програмиране или неформални

езици като UML. Също така, за модерни софтуерни архитектурни стилове като микроуслуги, във (Francesco, 2017) се посочва липса на език за описание на архитектури за формалното им описание, затова архитектите са склонни да използват неформални езици за моделиране, като SoaML (SoaML 2019).

## 1.4 Цели на дисертацията

В предишния раздел бяха очертани проблемите около ADLs. В тази дисертация се опитвам да адресирам основно два проблема: високата степен на формалност в синтаксиса и проблемите, когато става въпрос за изразяване на динамични архитектури, за да има принос за тяхната продължаваща еволюция.

*Целта на тази дисертация е да създаде нов език за описание на архитектури, наречен jADL, който осигурява средствата за формално описание на динамични и мобилни софтуерни архитектури със сравнително прост синтаксис*. Такъв език трябва да осигурява необходимите средства и езикови конструкции така че динамичните реконфигурации срещани в днешните системи, да бъдат адекватно изразени. Освен това, трябва да поддържа съвременни архитектурни стилове (микроуслуги) и да бъде придружен от инструмент, който би улеснил използването му. Целите на това изследване са:

- Създаване на език за описание на архитектури от ново поколение за дефиниране на съвременни и динамични архитектури, наречен jADL, който ще трябва да:
    - Дефинира такъв синтаксис и структура на езика, които ще помогнат при повишаването на степента на използването на ADLs извън академичните среди, като са прости и добре познати на разработчиците на софтуер.
    - Предоставя средства и езикови конструкции за изразяването на динамични реконфигурации на дадена софтуерна архитектура.
    - Поддържа описанието на съвременни архитектурни стилове, като микроуслуги.
- Валидиране на езика чрез описание на добре познати и широко използвани архитектурни модели (пр. Message Bus) и по-сложни съвременни архитектури.
- Разработване на инструмент за улесняване на използването на езика.

## 1.5 Публикации, свързани с дисертацията

Повечето от представените резултати са публикувани в различни конференции. Следва списък с публикациите свързани с дисертацията, разделени в списание и сборници от конференции.

Публикация в списание:

- 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

Публикации в конференции:

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

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

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

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

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

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

## 1.6 Структура на дисертацията

Структурата на останала част от дисертацията е:

- В следващата глава е представен обзор относно езиците за описание на архитектура. Разглеждат се редица от тях по отношение на начина, по който описват архитектурата.

- В глава 3 е представен езикът за описание на архитектура jADL. Обяснени са синтаксисът и семантиката му. Освен това са представени няколко примерни архитектурни описания с акцент върху конструкциите, които езикът предлага за динамична реконфигурация.

- В глава 4 е представено разширението на jADL, наречено μσADL (което се фокусира върху описанието на микроуслуги). Чрез илюстративни примери са представени приложенията на езика.

- В глава 5 е представен инструментът, проектиран за jADL. Той е изграден с помощта на рамката Xtext за разработване на домейн специфични езици.

- В глава 6 се обсъждат заключенията и бъдещото развитие.

- В последната глава, Библиография, може да се намери списъкът с референции, използвани за тази дисертация.

Глава 2

# Литературен Обзор

## 2.1 Въведение

В тази глава са представени редица формални езици за описание на архитектури и неформални езици. Има голям брой езици за описание на архитектура (над 120). Редица изследвания са проведени, както може да се види например в (Medvidovic and Taylor, 2000; Malavolta et al., 2012). За тази дисертация извърших анализ, за да получа представително подмножество от езици за сравнение. Това се постигна чрез класификацията и след това извличането на подмножество от езици въз основа на важните (в контекста на тази дисертация) критерии за тяхната поддръжка за динамична реконфигурация и определянето на сложни и дефинирани от потребителя конектори. Други важни фактори, които взех предвид, бяха тяхното използване в практиката и фокусирането им относно описанията (например структурни или време на изпълнение). Освен това бяха избрани езици, които се фокусират върху различни аспекти, като например AADL (анализ) и Xadl (разширяемост). Също така, езици които повлияха на синтаксиса на jADL. jADL е език за описание на архитектури, създаден с акцент върху описанието (на структурата и поведението) на динамични софтуерни системи. В следващите раздели е представено полученото подмножество от езици за описание на архитектурата. В заключението са показани обобщени резултати по отношение на изследваните езици и са посочени основните причини за създаването на допълнителен език.

## 2.2 Darwin

Darwin (Magee et al., 1995) използва компонентно базиран подход за описание на архитектурите акцентирайки върху разпределените приложения. Компонентите се дефинират чрез използването на интерфейси, които представляват услуги, които компонентът предоставя или изисква от средата. Конекторите не се считат за първокласни елементи в Darwin, така че няма такава спецификация при описание на архитектурата. Механизмите за взаимодействие са внедрени вътре в компонентите, като по този начин ги правят по-сложни и по-трудни за повторна употреба.

## 2.3 Wright

Wright (Allen, 1997) е език за описание на архитектури, който следва стила на компонент / конектор / конфигурация за описание на архитектурите. Компонентите в Wright изразяват независими изчисления и се дефинират в два основни раздела. Първо, интерфейсната част, която се състои от портове, дефинира точките на взаимодействие на посочения компонент. Второ, изчислителната част определя поведението на елемента, когато той взаимодейства със средата. Конекторите изразяват комуникацията между компонентите и са дефинирани на две части. Първо, интерфейсната част, състояща се от роли, определя точката на взаимодействие със средата. Второ, спецификацията Glue на конектор определя поведението на конектора. Третирайки конектори и компоненти като първокласни елементи, Wright увеличава независимостта, повторната употреба и облекчава анализа на архитектурните елементи и цялото архитектурно описание.

## 2.4 Rapide

Rapide (Luckham, 1996) е език за описание на архитектури който акцентира върху динамичните архитектури и симулацията им. Rapide дефинира компонентите чрез интерфейси. Те могат да се използват с цел моделиране както на синхронни, така и на асинхронни комуникации и могат също така да включват спецификации относно поведението. Тъй като Rapide използва компонентно базиран подход, не дефинира конектори като първокласни елементи на езика и механизмите на комуникация са интегрирани в компонентите. Това води до затруднение при повторно използване и наличието на по-сложни компоненти. Важен аспект на този език, както е посочено от Ozkaya (2014), е въвеждането на архитектурни ограничения. Те служат като глобални координатори, осигуряващи спазването на ограниченията от участващите елементи.

## 2.5 ACME

ACME (Garlan et al., 1997) започна като рамка за ADLs, предоставяща възможност за използването й като обща платформа за обмен за множество ADLs. Той следва класическата парадигма на компонент / конектор / система и дефинира конектори като първокласни елементи. През годините нуждата от динамична конфигурация нараства и тъй като тя не е „интегрирана“ в ACME, възниква нуждата от допълнителни инструменти/разширения (например ACME / Plastik (Batista et al. 2005)). Въпреки различните създадени разширения, все още има проблеми, когато става въпрос за динамична реконфигурация. ACME Studio (The Acme Studio Homepage 2009) е софтуерен инструмент, изграден като разширение за Eclipse, интегриран в него като плъгин. Той осигурява удобен за потребителя интерфейс за редактиране на архитектурни описания базирани на Acme.

## 2.6 Koala

Koala (van Ommering et al., 2000) е още един компонентно-базиран ориентиран език за описание на архитектури, който се фокусира върху описанието на софтуерни архитектури на продукти свързани с електроника. Компонентите са изчислителните единици и комуникират чрез своите интерфейси. В Koala интерфейсите се считат за първокласни елементи и се използват за моделирането на връзките между компоненти от по-високо ниво. Въпреки наличието на интерфейси, липсата на конектори като първокласни елементи не позволява да се опишат сложни комуникационни механизми. Взаимодействията са описани в раздела за свързване на декларацията на сложен компонент.

## 2.7 XADL

xADL (Dashofy et al., 2001) е разширяващ се и гъвкав език за описание на архитектури, базиран на XML. Използването на XML схеми интегрира в езика висока взаимозаменяемост и модулност. Това предоставя възможност за лесно повторно използване и създаване на свойства, които могат да разширят езика. Той има както текстово, така и графично представяне и предоставя две отделни схеми за спецификации по време на изпълнение и по време на проектиране на системата (Dashofy et al., 2002).

Схемата *Instances* се състои от инстанции на общи архитектурни конструкции като компоненти/интерфейси/конектори/т.н. и схемата *Structure & Types* се състои от типове за тези елементи. Двете схеми могат да бъдат разширени отделно. Друго важно предимство което дава използването на XML стандарти е фактът че има голям брой налични инструменти, които могат да се използват. Освен това са разработени и редица инструменти за поддържане на езика, като ArchEdit (Kotha, 2004). Поради множеството посочени схеми действителното архитектурно описание може да стане доста сложно, така че различна от XML нотация също може да се използва.

## 2.8 AADL

Architecture Analysis & Design Language (Feiler et al., 2006) (AADL) е проектиран с акцент върху спецификацията и анализа на критични за изпълнение в реално време разпределени компютърни системи (Architecture Analysis and Design Language 2015). Предоставя текстово и графично репрезентиране. Съществена разлика с обсъжданите досега езици е че има фиксиран набор от категории за компоненти, от които може да се избира, когато се дефинира архитектурата. Има три категории (Feiler et al., 2006): 1) приложен софтуер, 2) платформа за изпълнение и 3) сложен, която се състои от системни типове за спецификация на композитни типове. AADL не дефинира конектори като първокласни елементи и определя интерфейси, чрез които се осъществява комуникацията между компонентите.

## 2.9 π-ADL

π-ADL (Oquendo, 2004) е формален език за описание на архитектури, проектиран с акцент върху перспектива на една система. Той дефинира както компонентите, така и конекторите като първокласни елементи. Всеки от тези архитектурни елементи е дефиниран в две части. Първо се декларират връзките. След това се описва поведението на всеки елемент чрез прости оператори. И накрая, архитектурата се създава декларирайки инстанциите и техните взаимовръзки.

Освен това е разработен софтуерен инструмент (Cavalcante et al., 2015) за генерирането на програмен код на езика за програмиране GO (Donovan and Kernighan, 2016). π-ADL предоставя конструкциите, необходими за успешното изразяване на динамични и мобилни архитектури и, както се вижда от (Minora et al., 2012), може да поддържа (макар че може да се изисква използването на други езици) динамични реконфигурации.

## 2.10 PADL

PADL (Bonta, 2008) е език за описание на архитектури с висока експресивност и анализируемост. Описанията на архитектурата се изразяват чрез архитектурни типове (по отношение на компоненти и конектори). Един архитектурен тип се определя от неговото поведение и неговите взаимодействия. Последната стъпка при дефинирането на архитектурен тип е декларирането на архитектурната топология чрез изразяване на инстанциите на по-ранно декларираните архитектурни типове и техните взаимовръзки.

Езикът също е интегриран в TwoTowers (TwoTowers 5.1 2009) - софтуер с свободен код за верификация, анализ на сигурността и оценка на производителността. PADL2Java (Bonta и Bernardo, 2009) е софтуерен инструмент, създаден за генерирането на шаблони на програмен код на Java от PADL модели.

## 2.11 Неформални езици

Съществува голям брой езици за моделиране, които предлагат неформални начини за описание на софтуерните архитектури. Тук се пропускат формалности, срещани в предишните езици. В следващите раздели е представена част от тях, състояща се от езици, които са станали популярни през годините сред общността на софтуерното инженерство.

### 2.11.1 UML

Unified Modeling Language (UML) (Seidl et al., 2015) е език за моделиране с общо предназначение, който набира популярност през последните десетилетия и се превърна в един от най-широко използваните езици в общността на софтуерното инженерство. Той

дефинира два изгледа за моделиране на различните аспекти на системата. Статичният изглед се използва за представяне на статичната структура на системата, а динамичният изглед се използва за представяне на поведението на системата по време на изпълнение. Компонентите могат да бъдат дефинирани в компонентна диаграма в UML. Графика на взаимосвързани компоненти представлява архитектурата на системата. Интерфейсите, използвани за комуникацията, са разделени на два типа; предоставящи (*provided*) и изискващи (*required*).

Конекторите не са дефинирани като първокласни елементи в UML и взаимодействията между компонентите се моделират като прости канали за комуникация между техните портове (Ozkaya, 2014).

### 2.11.2 ComponentJ

ComponentJ (Seco и Caires, 2002) е, подобаващ на Java, език ориентиран към компонентно базирано програмиране и акцентиращ върху динамичната реконфигурация и еволюцията на софтуерните компоненти. Той не дефинира конектори като първокласни елементи на езика и дефинира три типа първокласни елементи (Seco et al., 2008): обекти, компоненти и конфигуратори. Предимство на ComponentJ е възможността за динамична конструкция и модификация по време на изпълнение на структурата и поведението на архитектурните елементи. Това води до добра поддръжка за описанието на реконфигурациите, които се появяват по време на изпълнение.

### 2.11.3 ArchJava

ArchJava (Aldrich, Chambers и Notkin, 2002a) е изграден като разширение и интегриран в езика за програмиране Java. Тъй като често имплементацията е различна от дефинираната архитектурата, ArchJava се опитва да разреши този проблем, като предоставя средства за описание на архитектурни характеристики „вътре“ в реализацията. Компонентите са специален вид обекти в ArchJava и комуникацията им се осигурява чрез определянето на портове. Те могат да декларират три групи от методи (Aldrich et al., 2002b): *requires, provides* и *broadcasts*. В ArchJava, първоначално, конекторите не са първокласни елементи, но вместо това се използва примитива *connect* между два или повече порта. Създадено е разширение (Aldrich et al., 2003), което осигурява моделирането на конектори.

### 2.11.4 SysML

SysML (Friedenthal et al., 2014) е език за моделиране с общо предназначение за приложения за инженерни системи (SysML 2018). Той е създаден като разширение на UML и въвежда нови свойства. Компонентите се дефинират чрез блокове, свързани помежду си с портове. Поддържа спецификация на поведението, но конекторите не са първокласни елементи.

### 2.11.5 SoaML

SoaML (SoaML 2019) е още едно разширение на UML и се фокусира върху архитектури ориентирани към услуги (SOA) (Erl, 2016). Той предоставя необходимите елементи за моделиране на услугите в SOA архитектура. Компонентите могат да бъдат представени като участници, които си взаимодействат помежду си чрез използване на услуги. Също така е разработен и софтуерен инструмент.

## 2.12 Заключение

В таблицата по-долу са представени обобщени данни. DNA означава "не се прилага". Използва се за неформалните езици в колоните Поколение (класификацията включва само формални езици) и Динамична реконфигурация (извън обхвата на дисертацията).

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

| Language | Generation | High-level Components | Connectors as first-class entities | Formal behavior specification | Dynamic reconfiguration |
|---|---|---|---|---|---|
| **ComponentJ** | *d.n.a.* | ✓ | **X** | **X** | *d.n.a.* |
| **ArchJava** | *d.n.a.* | ✓ | ✓ | **X** | *d.n.a.* |
| **SysML** | *d.n.a.* | ✓ | **X** | *state machine diagrams* | *d.n.a.* |
| **SoaML** | *d.n.a.* | ✓ | **X** | *state machine diagrams* | *d.n.a.* |

Както е представено в главата, всеки от тези езици се фокусира върху различни аспекти, що се отнася до описанието на архитектурата на софтуерните системи - напр. Wright в механизмите за комуникация, Rapide в симулация и т.н. Въпреки че има множество езици, все още има проблеми относно способността на език за описание на архитектури да изрази динамиката в настоящите софтуерни системи, тяхната висока степен (в повечето случаи) на формалност и липсата на инструменти. Тези три въпроса представляват основните причини, които ме доведоха до решението за създаване на нов език:

- динамизма и необходимостта за динамична реконфигурация в софтуерните системи, която се увеличи през последното десетилетие (напр. IoT, микроуслуги).

- липсата на ADL, който може да изрази тези нужди, като предостави синтаксис и езикови конструкции, които ще бъдат познати и сравнително лесни за изучаване и използване в практиката.

- да се предостави набор от инструменти (например редактор, преводач и т.н.) за архитекти и заинтересовани лица, за да се улесни използването на езика.

# Глава 3

# jADL

## 3.1 Въведение

В предишните глави бяха представени редица съществуващи езици за описание на архитектури, както и техните предимства и недостатъци. Въпреки големия им брой, изследвания, като тези проведени например от (Ozkaya, 2014; Malavolta et al., 2012; Ozkaya and Kloukinas, 2013), сочат че все още има проблеми относно използването на такива езици. Една от основните идентифицирани причини е необходимостта от адекватно изразяване на динамиката в софтуерните системи. Друга важна причина е високата степен на формалност, срещана в повечето архитектурни езици, която влияе негативно относно използването им. Освен това, вторичен проблем може да бъде, че тези езици, често не са придружени от инструменти, които биха улеснили използването им.

jADL е формален език за описание на архитектури, създаден в тази дисертация, за описание (както на структурата, така и на поведението) на статични, динамични и мобилни софтуерни архитектури. Той осигурява гъвкавостта и изразителността, необходими за изразяване на динамичните реконфигурации на интензивни софтуерни системи. jADL се базира на версия на π-calculus за асинхронни процеси на Milner (Milner, 1999), наречен приложен π-calculus, за изучаване на паралелността и взаимодействието на процесите. Той дефинира сравнително прост синтаксис и езикови конструкции, които могат да бъдат познати и лесни за научаване на разработчиците на софтуер, тъй като наподобяват на широко използвани езици за програмиране. Това е с цел да се повиши степента на използване на езиците за описание на архитектури в индустриалните процеси за имплементиране на софтуер, при който все още е много ограничена. Това може да доведе до подобрени и автоматизирани начини за създаване на реализации, които са в съответствие с първоначалната проектирана архитектура. Освен това езикът е придружен от инструменти (например редактор) за улесняване на неговото използване, които са представени в глава 5. Синтаксисът на jADL е повлиян от добри практики на други архитектурни езици, като ACME и π-ADL, представени в предишната глава.

В останалата част на този раздел е представен синтаксисът на jADL. Практическото използване на езика и различните му конструкции са илюстрирани чрез описанието на архитектурният модел Message Bus, чрез който са показани възможностите на езика за динамична реконфигурация на архитектура.

## 3.2 Синтаксис на jADL

В jADL основните елементи и първокласни архитектурни единици са компоненти и конектори. Освен това интерфейсите и communication traits също се считат за първокласни единици. jADL позволява създаването както на примитивни, така и на композитни компоненти и конектори. Архитектурните елементи са представени в следващите секции.

### 3.2.1 Компоненти

Следвайки парадигмата компонент-и-конектор, компонентите в jADL представляват изчислителните елементи и елементите за съхранение на данни. За да комуникират със средата си, те декларират редица портове, които представляват тяхната единствена точка на взаимодействие. Комуникацията между два компонента е строго чрез използване на конектори. Поведението на всеки компонент се дефинира чрез конфигурацията на техните *provides* портове (3.2.3) и определянето на вътрешни методи.

### 3.2.2 Конектори

Конекторите в jADL моделират комуникацията между различните компоненти и тяхната среда и само чрез тях два компонента могат да комуникират. Те декларират редица роли, които са прикачени към портове, така че комуникацията да бъде осигурена. Поведението на всеки конектор се дефинира чрез конфигурацията на техните *provides* роли и техните вътрешни методи. Компонентите и конекторите, участващи в комуникацията, могат да бъдат част от един и същи процес (или нишка), както и части от различни процеси и нишки и могат да бъдат групирани заедно, за да се получи композитен елемент. За да се осъществи комуникация между компонент и конектор, трябва да се установи връзка между тях. В jADL това се постига чрез прикачване на роля към порт, използвайки прост оператор, както е обяснено в следващите раздели.

### 3.2.3 Портове & Роли

Портовете са единствената точка на взаимодействие за компоненти (съответно роли за конектори). И портовете и ролите в jADL се разглеждат като първокласни архитектурни елементи. Те се използват за осигуряване на контрол и поток от данни, който се установява с прикачването на роля към порт. Те се характеризират със своите *интерфейси*, *вид* и *множественост* и *синхронност* на връзката си. Тези също са факторите, които определят дали прикачването ще бъде успешно или не; двата интерфейса трябва да са *съвместими*, техните видове трябва да са *противоположни* и с *еднакъв* синхрон. Когато порт и роля са свързани, техните интерфейси трябва да са съвместими. Това се постига чрез унифициране между интерфейса на порта и интерфейса на ролята.

#### 3.2.3.1 Вид

При декларация на портове и роли ключовите думи *provides* и *requires* се използват за деклариране на техния *вид*. Всеки порт или роля трябва да има вид. Видът *provides* се използва за деклариране на порт или роля, която изпраща данни чрез връзка. Информацията, обработена в прилаганите методи на даден компонент, например, е достъпна на неговия порт и ще бъде предоставена на всяка успешно прикачена роля към него, която ще я поиска. От друга страна, видът *requires* се използва за порт или роля, която очаква данни чрез връзки. При създаването на връзка видовете на участниците се сравняват и ако те не са противоположни, връзката е неуспешна.

#### 3.2.3.2 Многочисленост (Multiplicity)

Най-простият тип връзка е когато една роля е прикачена към един порт (1-1 комуникация). В допълнение към това, jADL поддържа и по-сложни връзки от типа 1-N комуникация. Фигура 2 показва случаи, в които при връзките участват повече от два архитектурни елемента.

Докато прикачването в 2.a е успешно, това в 2.b не е и трябва да се трансформира, както е показано на фигурата. Това се дължи на факта, че в jADL има ограничение по отношение на портовете и ролите на вида *requires*. *Само деклариран като provides порт (или роля) може да бъде прикачен към множество requires роли (или портове)*. Когато повече от един *provides* портове или роли са прикачени към една *requires* роля или порт, тогава се появяват проблеми с недетерминизъм.

#### 3.2.3.3 Синхронност

В jADL при деклариране на порт или роля може по избор да се използва ключовата дума (*synchronized*), която определя синхронността на комуникацията; когато се използва комуникацията е синхронна, а при пропускане комуникацията е асинхронна.

При 1-N комуникация могат да се появят допълнителни проблеми от споменатите в предишния подраздел, когато всеки от участващите архитектурни елементи е част от различна нишка. На фигура 3 е илюстриран този случай; двата конектора, всеки изпълнен в различна нишка, могат да се опитат да получат достъп до един и същ ресурс, така че ще възникнат проблеми свързани с конкурентността. Например, нека приемем, че в компонент *C* има опашка (*q1*), дефинирана в други елементи, и двата конектори *Con1* и *Con2* се нуждаят от размера на тази опашка, за да продължат със своите изчисления. Тогава в описанието на компонента *C* ще има част където този размер ще бъде достъпен за други елементи чрез неговия порт *p* и конфигурацията ще бъде:

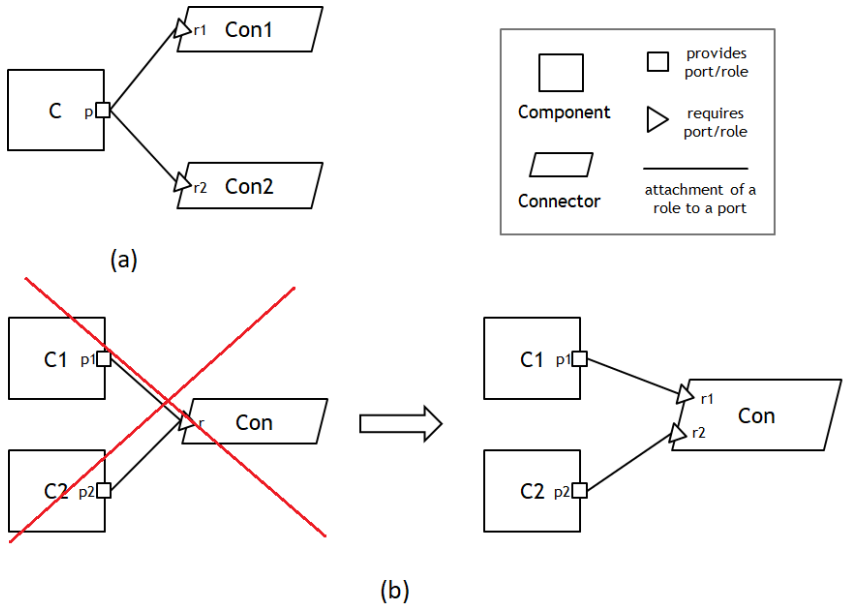
```
provides port IQueue p;
config p as {
```

```
int getSize() {
        return q1.size();
}       }
```

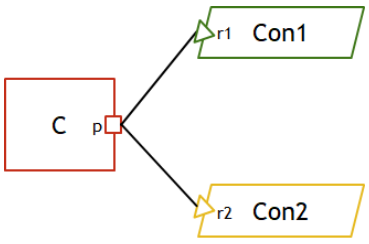
В описанието на конектора *Con1* (и съответно на *Con2*) ще има част, където този размер ще бъде поискан и ще изглежда като:

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

Този код би бил правилен, ако имахме само една нишка на изпълнение. Но тъй като имаме три различни нишки, кодът трябва да бъде променен; ключовата дума *synchronized* трябва да бъде добавена при декларациите както на порта *p*, така и на ролите *r1*, *r2*.




Фиг. 2. *Различни случаи на 1-N комуникации в jADL.*




Фиг. 3. *1-N комуникация на три различни нишки в jADL.*

### 3.2.4 Интерфейси

Интерфейсите се използват за определяне на формата на комуникацията и поведението на порт или роля. Те представляват описания на протоколи, които определят комуникацията между архитектурните елементи. Основно предимство на интерфейсите е възможността за групиране заедно на различни канали за връзка.

Интерфейсът на порт или роля определя формата на комуникация и не бива да бъде разбиран погрешно като извикване на метод поради тяхната синтактична прилика. Например, *void f (Integer a, Double b, String c)* като част от интерфейс представлява канал според π-calculus (Milner, 1999), където *f* е името на канала. Ако интерфейсът се използва за деклариране на *requires* порт, тогава се очаква през канал *f* да бъдат получени стойностите *tuple (a, b, c)*. Типовете им са (*Integer, Double, String*).

### 3.2.5 Описание на поведение

Поведението на портове и роли се дефинира чрез използването на израза *config*, между скоби *{ }*. Тази дефиниция се състои от услуги. Те са същите като тези, дефинирани в интерфейсите на портове/роли, но „съдържат“ поведението, което се дефинира под формата на прости изрази. Всички портове и роли, декларирани като *provides*, трябва да бъдат конфигурирани с помощта на този оператор.

Операторът *config* може да се използва и по време на изпълнение, за динамично дефиниране на поведение. Това означава, че можем да реконфигурираме поведението на порт или роля и това е един от механизмите на jADL за поддържане на реконфигурируемостта на архитектурните елементи по време на изпълнение. Пример за използване по време на изпълнение е, когато порт или роля са (ре)конфигурирани в декларация за агрегиране на *communication trait* (в следващия подраздел).

### 3.2.6 Communication Traits

Communication trait е сложна комуникационна структура в jADL, която може да групира портове и роли и се счита за първокласен елемент. Използването на тази конструкция се състои от две части; първо декларирането на communication trait, което може да бъде направено както вътре, така и извън друг архитектурен елемент (компонент и конектор в този случай). Второ, агрегирането на този communication trait, което трябва да се извърши вътре в архитектурния елемент, който ще го използва.

Във всеки communication trait може да бъде деклариран различен брой портове или роли, стига типът на архитектурните елементи да остане един и същ - т.е. *всеки communication trait може да съдържа само портове или само роли*. Чрез капсулиране на портовете и ролите в отделна структура и чрез използването на втората форма на оператора *attach* (както е описано в подраздел 3.2.7.1) предоставяме възможност за динамичното инстанциране на

портове и роли по време на изпълнение. Без тази структура реконфигурирането на архитектурен елемент би изисквало поредица от декларации за отделяне, създаване на нов елемент и серия от прикачващи изявления. Communication traits позволяват да се извършват такива операции с използването на просто изявление и ползите им могат да се видят (особено) в примера на описанието на архитектурния модел Message Bus, показан в последния раздел на тази глава.

Друга полезна характеристика на тази конструкция е, че връзките в jADL, когато са използвани два communication traits, се правят на фона и няма нужда от изричното деклариране на имена. Въвеждането на тази сложна структура повишава гъвкавостта и изразителността на jADL, особено когато става въпрос за описание на динамични архитектури и изразяването на предвидени (които са известни по време на проектиране) и непредвидени (които не са известни по време на проектиране) реконфигурации.

### 3.2.7 Изрази в jADL

jADL дефинира редица изявления, за да предостави средствата на софтуерните архитекти да опишат формално дадена архитектура. В следващите подраздели тези изявления са представени. Те се отнасят до два аспекта на дефиницията на архитектурата в jADL; връзките и дефиницията на поведението.

#### 3.2.7.1 Attach / Detach

Операторът *attach* се използва за обединяването на портове и роли и създаването на комуникационен канал, така че да се осигури контрола и потока от данни между архитектурните елементи. Той може да приеме (<*role*>, <*port*>) двойка или (<*trait*>, <*trait*>) двойка като аргументи. В първия случай, както е описано в раздел 3.2.3, проверките за успешното обединяване са: съвместимостта на интерфейсите, противоположните видове на порта и ролята и същата синхронност. Във втория случай компилаторът проверява освен това портовете и ролите на двата *traits*, предоставени като аргументи и или се установява комуникационен канал между двата елемента, или се връща грешка. Първо, се проверява дали единият *trait* се състои от портове, а другият от роли. Втората проверка е дали и двата се състоят от един и същ брой роли и портове на противоположни видове и дали всяка двойка противоположни видове има еднакви интерфейси. Ако и двете проверки са успешни, тогава прикачването е установено и портовете и ролите са унифицирани.

Операторът *detach* е обратният на *attach* и се използва за унищожаване на комуникационния канал, установен между два архитектурни елемента. Подобно на *attach*, той може да приеме едни и същи двойки аргументи. Тези два оператора могат да се използват както при дефиниране на архитектура, така и по време на изпълнение за динамичното реконфигуриране на архитектурата.

#### 3.2.7.2 Delay

Операторът *delay ce* използва за блокиране на изпълнението на операциите в рамките на система за даден период от време. Той има две форми за определянето на този период, под който може да бъде деклариран; чрез използването на цяло число (милисекунди) или израз.

#### 3.2.7.3 Select

Операторът *select* се използва при дефинирането на поведението на елемент в операторът *config*. Поредицата от изрази, които трябва да бъдат изпълнени, се избират от блока където *when* изразът е верен.

#### 3.2.7.4 Process

Този оператор се дефинира с ключовата дума *process*. Използва се в архитектурно описание за да изрази, че даденият архитектурен елемент продължава да работи „*както е*".

#### 3.2.7.5 Bind

Този оператор се отнася до специален случай относно връзките в jADL. Описва връзката между външен порт или роля на сложен компонент или конектор с вътрешен порт или роля на един от неговите вътрешни архитектурни елементи. Единственото ограничение относно *bind* е че *аргументите трябва да бъдат от същия вид*.

### 3.2.8 "Прости" Изрази

Освен специалните оператори представени досега, в jADL стандартни изрази, които съществуват в повечето езици за програмиране, могат да се използват. Подобаващи на Java, за фамилиарност относно програмисти, те включват операторите: *if, for* и *while.*

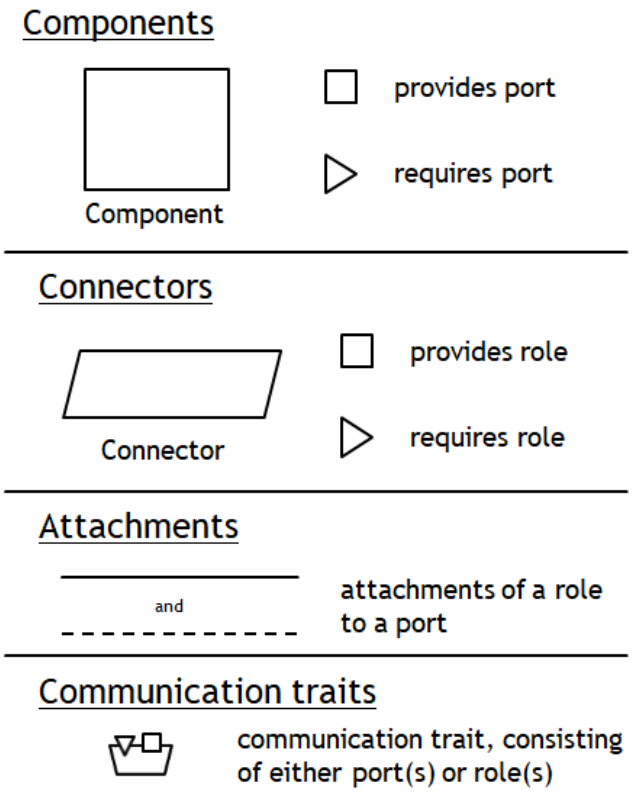
### 3.2.9 Променливи и структури от данни

jADL дефинира атрибути и променливи като други езици за описание на архитектури, които се използват за описанието на ограничения над архитектурата. Той дефинира примитивни типове данни като Integer, String и др. и параметризирани по тип структури от данни като List, Hashmap и т.н. За jADL вторите представляват компоненти с два порта (един за получаване и един за предоставяне на информация), които имат зададени услуги.

## 3.3 Графично представяне на jADL

jADL като обикновен архитектурен език има две части: текстово представяне на архитектурен скрипт, както и графична част - графично представяне на архитектурата.

Графичните изображения могат да улеснят комуникацията между различните заинтересовани лица и предоставят удобен начин за представяне на архитектурата. На фигура 4 може да се види графичната нотация на всеки архитектурен елемент в jADL.




Фиг. 4. *Графични нотации в jADL.*

## 3.4 Message Bus Architectural Pattern

Една от най-приетите дефиниции на Enterprise Service Bus (ESB) (Keen et al., 2005) е „*стил на интеграционна архитектура, който позволява комуникация чрез обща комуникационна шина, която се състои от различни връзки от точка до точка между доставчици и потребители на услуги* ”(Enterprise Service Bus 2013). ESB представя архитектурен модел, който очертава основния набор от правила за интегриране на различен брой хетерогенни приложения заедно.

Днес е широко използвана концепция, тъй като бързото разпространение на интернет и непрекъснато нарастващият брой услуги като IoT, и др. изискват различни приложения да комуникират и/или да обменят информация бързо, сигурно и надеждно. Концепцията за ESB определя модел, който позволява на различни системи да комуникират, без да имат зависимости помежду си. Той даде адекватен отговор на необходимостта от различен

подход от интеграцията от точка до точка, която често е трудно (или невъзможно в случаи) да се управлява или развива във времето (силно взаимозависими модули).

Основното предимство на ESB архитектурата е фактът, че се увеличава разединяването между компонентите които комуникират. Те са свързани към шината, а не към реалния доставчик на услугата, като по този начин елиминират всякакви зависимости между тях и облекчават процеса на добавяне/премахване на компоненти. Той представлява предпочитана среда за налагане на сигурност, тъй като следи всички взаимодействия между компонентите. Допълнителните предимства и функции, предоставени от ESB, могат да бъдат поддръжка при отказ, балансиране на товара за подобряване на работата и т.н.

Тук е представено архитектурно описание на вариант на Message Bus Architectural Pattern (MBAP). Архитектурата на MBAP се състои от конектор, който играе ролята на Message Bus, и различен брой (динамично променящи се) компоненти, които играят ролята на податели и приемници. Архитектурата на MBAP може да се види на фигура 5.

**jADL Client Component Description**

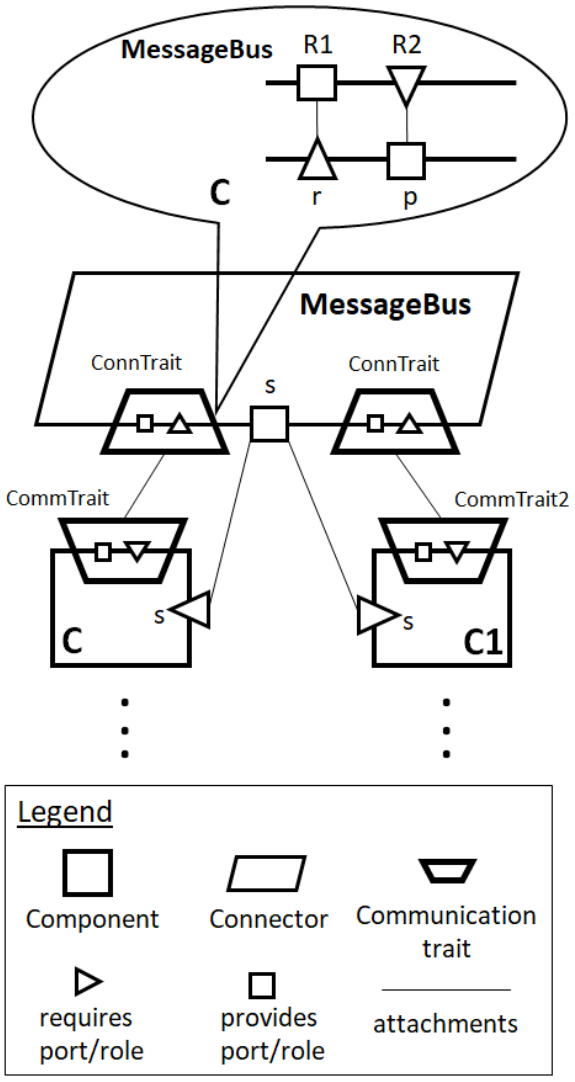
```
1.   type Message;
2.   interface ISendMsg {
3.    service void sendMsg (Message msg, CommTrait comT);
4.   }
5.   interface IReceiveMsg {
6.    service void getMsg (Message msg);
7.  }
8.  interface ISubscribe {
9.   service void subscribeTo (CommTrait comT);
10.   service void unSubscribeFrom (CommTrait comT);
11.  }
12.  trait CommTrait {
13.   provides port IReceiveMsg p;
14.   requires port ISendMsg r;    }

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

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

Фраг. от код 1. *Описание на клиента в jADL.*

Ред 1 в фрагмент на код 1 определя, че архитектурата е параметризирана по отношение на данните (съобщенията), обменяни между компонентите (например XML документи). *Type*, е абстракция за различните типове данни, поддържани в jADL.




Фиг. 5. *Архитектура на Message Bus Architectural Pattern.*

Компонентът *C* може да бъде всеки компонент на дадено приложение, което трябва да комуникира със средата си чрез обмен на съобщения. В декларацията си само един порт е деклариран статично - *requires* порта *s*. От този порт той може да изпрати заявката си, когато трябва да се абонира или да се отпише. За да се абонира успешно за конектора *MessageBus*, той трябва да има подходящи интерфейси на своите портове. Това се постига чрез използването на *CommTrait*, които е динамично инстанциирян по време на изпълнение. *CommTrait* се състои от необходимите портове (и интерфейси), които един компонент трябва да има, за да се абонира за *MessageBus*. След като прикачването е успешно,

компонентът може да започне да изпраща съобщенията си през неговия порт *com1.r* и да получава съобщения от *MessageBus* през неговия порт *com1.p*. Вътре в тялото на компонента, портът *com1.p* е конфигуриран с помощта на оператора *config* и се назначава поведение по отношение на обработката на получено съобщение.

При инициализацията на конектора има само една роля, статично декларирана - ролята, наречена *s*. Тя е декларирана като *provides*, така че към нея да могат да бъдат прикачени множество компоненти, без да се срещат проблеми с недетерминизъм, тъй като само този вид портове/роли позволява множество връзки в jADL. Към ролята е назначен интерфейса *ISubscribe* и е конфигурирана както е показано в фрагмент на код 2.

**jADL Connector Description (MBAP)**

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

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

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

11.  trait Comm1 aggreates ConnTrait {
12.    config R1 as {
13.     void sendMsg(Message msg, CommTrait comT){
14.       messages.put(comT, msgs.add(msg));
15.     }
16.    }
17.  }
18.
19.  config s as {
20.    service void subscribeTo (CommTrait comT) {
21.     if (subscribers.size() < maxRoles ){
22.       Comm1 com1 = new Comm1();
23.       attach(com1, comT);
24.       subscribers.put(comT, com1);
25.     }
26.    }
27.    service void unSubscribeFrom (CommTrait comT) {
28.     detach(subscribers.get(comT), comT);
29.     subscribers.remove(comT);
30.     messages.remove(comT);
31.    }
32.   }

33.   while(true){
34.    for(messages msgKey : msgVal) {
```

```
35.     for(subscribers subsKey : subsVal) {
36.      if (subsKey != msgKey) {
37.       subsVal.R2.getMsg(msgVal.get(0));
38.       msgVal.remove(0);
39.      }    }   }
40.    delay 20;
41.   }    }
```

Фраг. от код 2. *Описание на конектора (MBAP) в jADL*.

Има само два типа заявки, които се изпращат към конектора чрез тази роля - първият е от компоненти, които искат да бъдат прикачени към *MessageBus*, а другият от компоненти, които искат да бъдат отделени от него. Ако пристигне нова заявка от компонент, който трябва да бъде абониран, нова инстанция от *trait* на конектора се създава и се прикрепя към *trait* на компонента, който се предоставя като аргумент от услугата *subscribeTo*. Описаното семантично представено е близко до действителните реализации, използвани днес.

Двете структури от данни (*hashmaps*) са дефинирани с цел управление на абонатите и техните входящи съобщения за разпространение. Първият (*subscribers*) се състои от препратки към *traits* на компонентите като ключове и препратките към прикрепените им *traits* като стойности и се използва за управление на абонатите. Конекторът, използвайки тази структура, може да определи кой компонент е прикачен към всяка от дефинираните роли. По този начин той може да определи изпращача на всяко ново получено съобщение, броя на абонатите във всеки даден момент и т.н. Втората структура на данни (*messages*) се използва за обработка на съобщенията, които конекторът получава. Състои се от препратки към *traits* на всеки прикачен компонент като ключ и Списък на съобщенията от всеки компонент съответно като стойности.

Както беше обяснено в предишните раздели, тези структури от данни се разглеждат като компоненти с налични услуги на портовете. Следователно вътрешните елементи, които съставят MBAP, могат да се видят на фигура 6.

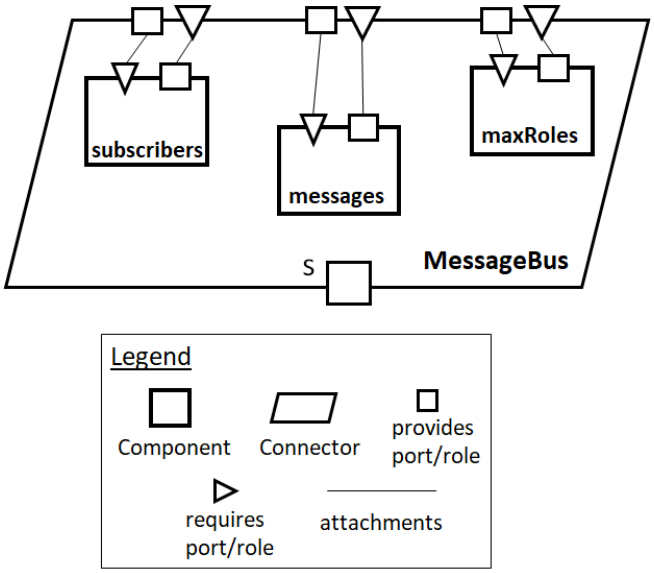
**jADL MBAP Description**

```
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);  }
```

Фраг. от код 3. *MBAP описание в jADL.*




Фиг. 6. *Вътрешни компоненти на MessageBus*.

В фрагмент на код 3 се показва скрипт, който трябва да бъде изпълнен след дефиницията на елементите. Първо те се инстанциират и след това се дефинира топологията на системата. Определят се само първоначалните връзки между порта и ролята *s*. След като компонент изпрати заявката си да се абонира за *MessageBus* (напр. ред 8), конекторът обработва динамично останалите взаимодействия и осигурява получаването и изпращането на съобщения от и към компонента.

## 3.5 Заключение

В тази глава беше представен архитектурният език за описание jADL, създаден в тази дисертация. Анализирани бяха архитектурните елементи и останалите конструкти на езика. Синтаксисът е сравнително прост и добре познат на разработчици на софтуер, като например оператора *new*, който се използва за създаване на нови архитектурни елементи. Въпреки че е формален архитектурен език, той определя лесен и елегантен синтаксис, който

позволява добра изразителност и гъвкавост, както е показано в предишните раздели. Тези са и основните характеристики, които могат да помогнат за популяризирането на ADL при по-нататъшно използване в практиката.

Освен това в последния раздел е представен пример където се описва архитектурата на Message Bus, широко използван архитектурен модел за интегриране на различен брой хетерогенни приложения. Езиковите конструкции, предоставени от jADL (особено използването на communication traits), се оказаха адекватни за описание на системата и показаха гъвкавостта на езика, когато става дума за изразяване на динамични реконфигурации.

# Глава 4

# μσADL

## 4.1 Въведение

В тази глава е представено разширението, създадено за jADL, наречено μσADL. Целта му е да предостави средствата за описание на архитектури на микроуслуги. Създаден е за да позволява дефиницията на архитектурни описания, използвайки прости структури, които крият формалностите, срещани в архитектурните езици, които обезкуражават архитектите да ги използват. Добавяйки един допълнителен слой на абстракция „крие“ ненужно стриктни дефиниции, предоставяйки практичен начин за да се опишат адекватно софтуерни системи, които следват този архитектурен стил.

## 4.2 Архитектури на Микроуслугите

Архитектури на Микроуслугите (микроуслуги) (Amundsen et al., 2016; Newman, 2015) е нов архитектурен стил, който се появи през последното десетилетие, ставайки все по-популярен. Няколко софтуерни компании са преминали към микроуслуги с обещаващи до сега резултати. Има нарастващ брой изследвания, които се занимават с различни аспекти на микроуслугите, като например (Mayer and Weinreich, 2017). Въпреки, че архитектурен стил на микроуслугите не е конкретно описан, една широко използвана дефиниция е тази, дадена от Lewis и Fowler (Microservices 2014). Те определят микроуслугите като "*подход за разработване на едно приложение като набор от малки услуги, всяка от които работи в свой собствен процес и общува с леки механизми, често API чрез HTTP. Тези услуги са изградени около бизнес възможности и независимо са разпределени чрез напълно автоматизирани машини за внедряване*".

Както показва (Francesco, 2017) има липса на език за описание на архитектури за описанието на микроуслугите и архитектите обикновено използват езици които описват архитектури ориентирани към услуги, като SoaML.

## 4.3 μσADL Конструкции

μσADL като типичен архитектурен език се състои от две части: текстова презентация и графична такава. Една микроуслуга в μσADL се състои от портове, множество от атрибути,

(по избор) лично съхраняване на данни и поведението му. Портовете са дефинирани и конфигурирани по същия начин, показан в предишната глава.

Всяка микроуслуга отговаря за изчислителната част. Така тя представлява компонент, когато е трансформирана от μσADL на jADL и всичките оператори/изрази/т.н. на jADL могат да се използват.

### 4.3.1 Комуникация на микроуслугите в μσADL

Микроуслугите имат за цел приложения, където свързването (*coupling*) е възможно най-разхлабено и кохезията (*cohesion*) е възможно най-силна. Това се описва като „*умни точки и прости тръби*“; микроуслугите получават заявка и връщат отговор. Има два начина, използвани главно за комуникация при изграждане на приложение с микроуслуги; директна комуникация чрез леки протоколи (REST) или съобщения през Message Bus (Microservices 2014).

В μσADL това се моделира по следния начин. В първия случай имаме типичен архитектурен модел клиент-сървър, при който една микроуслуга действа като клиент и изпраща заявка до втора микроуслуга (действащ като сървър), от която очаква отговор. Вторият начин на комуникация е чрез Event/Message Bus (MB). Всяка микроуслуга, която е абонирана за MB, произвежда съобщения/събития, които праща към MB и консумира съобщения/събития от там. Заедно с използването на предварително дефинираните и интегрирани в jADL communication traits, представени в предишната глава, могат да бъдат описани различни видове MB. Те могат да бъдат извикани директно, инстанциирани и използвани в архитектурното описание.

### 4.3.2 Съхранение на данни в μσADL

В архитектурата на микроуслуги, когато става въпрос за съхранение на данни, се предпочита децентрализиран подход за управлението на данни. Всяка микроуслуга управлява собствена си база данни.

Следвайки този принцип, в μσADL позволяваме на всяка микроуслуга да дефинира своя собствена инстанция на база данни. Използвайки ключовата дума *database* и в *{ }* архитекта може да определи необходимите атрибути за създаване на конектора, който той/тя желае за дадена микроуслуга и база данни. Използвайки това просто описание, след това можем автоматично да генерираме подходящия конектор в jADL. Например, ако приемем, че имаме микроуслуга, разположена на същото място с нейната база данни (*localhost*) и се нуждаем от стандартен конектор JDBC. Описанието в μσADL ще бъде:

```
database {
  location: localhost;
  connector: JDBC;
```

***schema: invSchema;***
***username: user1;***
***password: mypass;  }***

Описаното по-горе би довело до създаване в jADL на нов компонент на базата данни и неговия подходящ конектор, така че да може да бъде прикачен към микроуслугата:

**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. }
```

Фраг. от код 4. *Преведените, в jADL, компонент и конектор*.

## 4.4 Проектиране на  микроуслуги с μσADL и BPMN

Възприемаме към стила на микроуслугите подобен подход на (Oquendo, 2008), който се отнася до описанието на ориентирани към услуги архитектури. Предлагаме процес като първа стъпка към методология за описанието на софтуерни системи, които са изградени въз основа на този архитектурен стил, използвайки μσADL и jADL, състоящ се от 3 части:

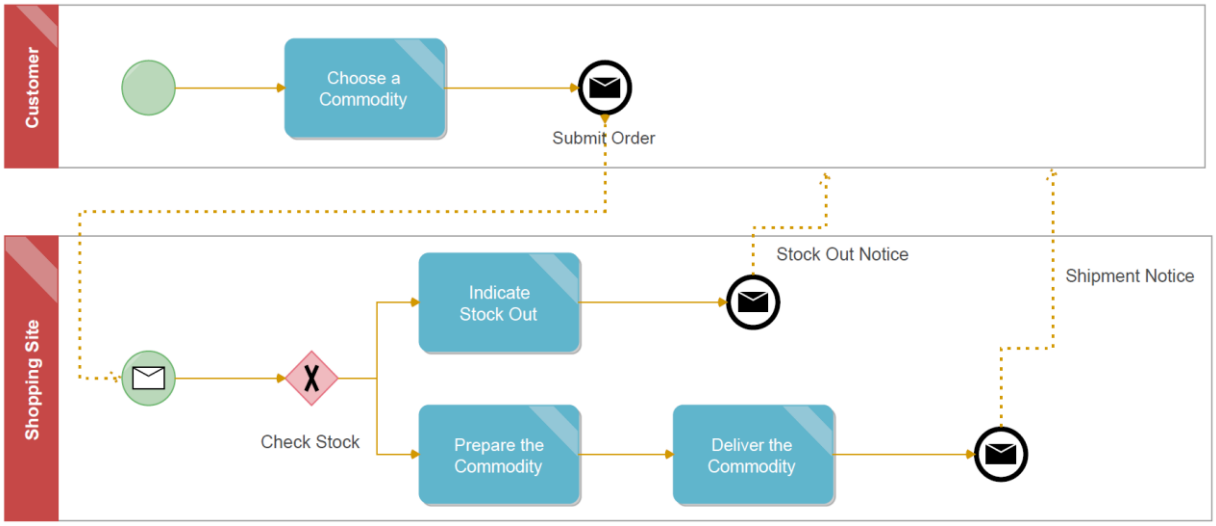
i) Извличане на първоначална архитектурна скица от BPMN представяне, при което всеки процес може да бъде моделиран като микроуслуга.

ii) Спецификацията на архитектурата с μσADL. Езикът предоставя необходимите конструкции, за да дефинира както всяка от микроуслугите, така и цялостната архитектура на системата (техните механизми за комуникация и т.н.).

iii) Автоматичен превод на μσADL описанието в jADL описание.

### 4.4.1 Изучаване на случаите на Проста Система за Онлайн Пазаруване

Business Process Modelling Notation (BPMN) е стандартизирана визуална нотация за моделиране на бизнес процеси. На фигура 7 се вижда прост процес на онлайн пазаруване.




Фиг. 7. *Процеси на Онлайн пазаруване в BPMN* – от (Online Shopping Process 2019).

Клиентът избира стока и изпраща заявката си на сайта. Инвентаризацията се проверява и се изпраща или известие за наличност, или известие относно данните за пратката. Архитектурата на тази система е динамична: стоките могат да се добавят или премахват и начинът на доставка на стоките може да варира в зависимост от клиента. От гледна точка на софтуерната архитектура това е типичен комуникационен модел клиент-сървър. Клиентът изпраща заявката си до сървъра (магазин за пазаруване) и след обработката на заявката съответно се изпраща отговор.

Сега се фокусираме върху сървъра и как е организиран. Използвайки архитектурния стил на микроуслугите, сървърът може да бъде разделен, както следва. Всеки от процесите може да бъде моделиран като отделна микроуслуга - получаване на поръчка, проверка на инвентара и информация за пратката.

Първата стъпка от процеса е извличане на първоначална архитектурна скица на архитектурата на софтуерната система от BPMN модел. На този етап гранулирането на микроуслугите в дадена архитектура зависи от архитекта. Дефиницията на подходящата

подробност (*granularity*) все още е област, в която има много текущи изследвания. Тук използваме три микроуслуги.

Втората стъпка от предложения процес се отнася до описанието на архитектурата в μσADL. Описанието на микроуслугата *shipping* е представено в кодов фрагмент 5. Важна разлика между *order* и другите две микроуслуги е, че *order* няма собствена база данни, за разлика от другите две. Просто изпраща съобщение, когато поръчката е приета и връща отговор на клиента, когато обработката приключи.

Езиковите конструкции, предоставени от μσADL, се оказаха адекватни за описанието на всяка микроуслуга и техните механизми за комуникация. Строга и твърде формална семантика са „скрити“ в μσADL и архитектът може да определи архитектурата по прост и елегантен начин.

По време на третия етап от процеса се извършва автоматичен превод на описанието от μσADL в jADL. Това се прави, за да се използва редакторът, изграден за jADL, за валидиране на дефинираната архитектура. Генерираното текстово архитектурно описание в jADL на микроуслугата *Inventory* може (частично) да се види във фрагмент на код 6.

**μσADL description**

```
1. microservice Shipping {
2.
3.   requires port ISubscribe r;
4.
5.   trait ShipTrait aggregate CommTrait {
6.     config p as {
7.       service void getMsg (Message msg) {
8.         reply(msg);
9.       }
10.      }
11.   }
12.    instance com1 = new CTrait();
13.
14.    database {
15.     location: "localhost";
16.     connector: "MySQL";
17.     schema: "shipSchema";
18.     username: "user1";
19.     password: "mypass1";
20.    }
21.
22.    config pDB as {
23.      service void getQueryRes (type data) {
24.         com1.r.sendMsg(com1, data);
25.      }
26.    }
27. }
```

Фраг. от код 5. *Описание на микроуслуги в μσADL.*

**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);   }
```

Фраг. от код 6. *jADL описание на микроуслугата Inventory*.

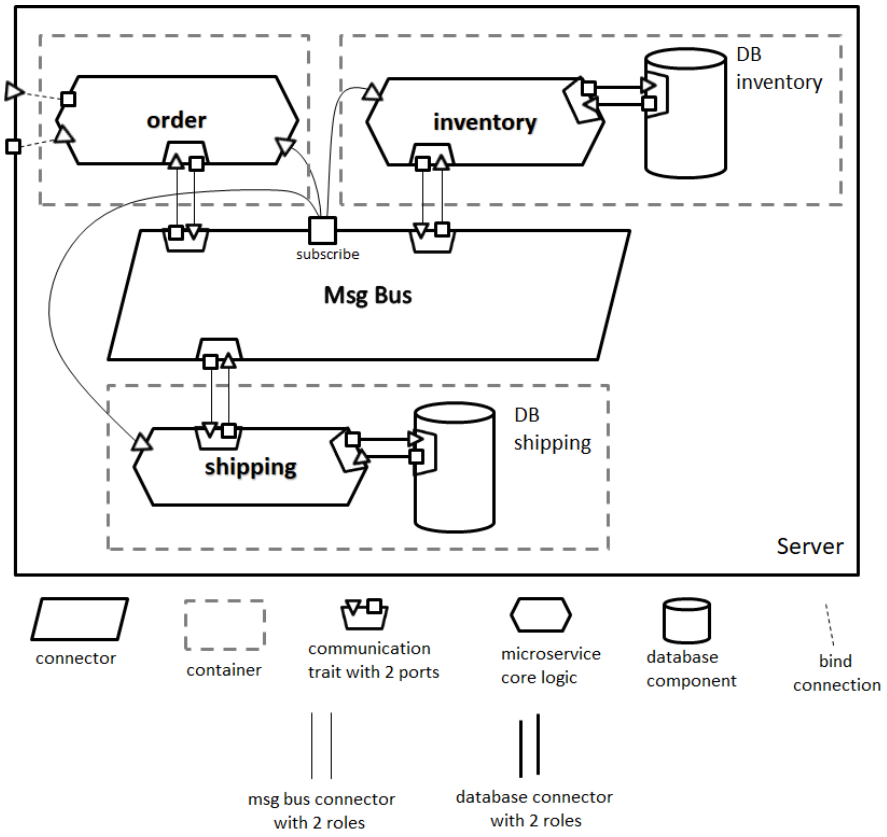
За да могат микроуслугите да комуникират, архитектурният модел на шината за съобщения (MBAP), описан в предишната глава на тази дисертация, може да се използва. Необходимият *communication trait* е:

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

Когато микроуслуга (или компонент в jADL) използва този *trait*, той трябва да определи поведението на *provides* порта *p*, както е показано в кодов фрагмент 5.

След инстанциране на сървърния компонент можем да определим архитектурата на системата за пазаруване онлайн, която се състои от описания по-рано сървър, клиент и конектор.

Използвайки прост общ модел BPMN който описва бизнес процеси свързани с онлайн магазин за пазаруване и μσADL, стигнахме до формалното описание на архитектурата в jADL.




Фиг. 8. *Графично представяне на сървърния компонент в jADL.*

### 4.4.2 Динамична реконфигурация

Обща характеристика на микроуслугите е необходимостта от динамична реконфигурация - т.е. промяна (предвидена или непредвидена) на топологията на софтуерна система по време на изпълнение.

Продължавайки с предишния пример, процесът на инвентаризация може да се промени в бъдеще, следователно нова микроуслуга трябва да замени старата. Езиковите конструкции *attach* и *detach* позволяват лесно да се опише такава промяна на ниво инстанция на дадената архитектура.

Мащабируемостта (*scalability*) е друг важен качествен атрибут, що се отнася до микроуслугите. Въпреки че може да бъде предизвикателство, тъй като може да изисква работа с различни компоненти, в μσADL един от начините да се реши този проблем е да се използва архитектурният модел на динамичния load-balancer, реализиран в jADL.

По подобен начин, по който MBAP се използва в предишната глава, архитектът може да го използва със същото или различно поведение. Вместо сървърите, представени там, инстанциите на всяка микроуслуга могат да бъдат управлявани от такъв load balancer, след като се конфигурира поведението на всяка микроуслуга. Прилагайки това към този пример, кодът за сървърният компонент става:

**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();     }
```

Фраг. от код 7. *Описание на сървъра в μσADL*.

## 4.5 Заключение

Разширението на jADL, наречено μσADL, беше представено в тази глава. Разширение, създадено с цел да се улесни описанието на софтуерните системи, които следват архитектурния стил на микроуслугите.

Една от основните цели на μσADL е да добави допълнителен слой на абстракция, в сравнение с jADL, където строгите и твърде формални изисквания на jADL могат да бъдат „скрити“. Когато става въпрос за постоянното съхранение на данни за микроуслуга, това се постига с използването на израза *database*. Както е показано в предишните раздели на тази глава, се използва проста декларация, състояща се от *име:стойност* двойки. Що се отнася до комуникацията между микроуслугите, *communication traits* и MBAP, обсъдени в предишната глава, могат да се използват. Това допълнително автоматизира и облекчава описанието на софтуерни системи, изградени с помощта на микроуслуги.

Освен това беше представен процес относно практическото приложение на μσADL. Той се отнася до предложен начин за достигане до формално архитектурно описание на софтуерна система, като се започне от BPMN диаграма(и). Както е показано в примера, представен в

тази глава, следвайки трите стъпки на предложеният процес това може да бъде постигнато. С използването на серия от прости и елегантни изрази в μσADL може да се получи подробно формално описание на архитектурата в jADL. Формалните дефиниции относно прости архитектурни елементи могат да бъдат пропуснати или значително намалени и опростени, като по този начин се предоставя по-практичен и удобен за потребителя начин за описание на софтуерни архитектури. Това може да помогне за използването на езиците за описание на архитектури в индустриални процеси на разработване на софтуер, тъй като BPMN диаграмите са широко използвани в практиката.

Въпреки, че могат да бъдат решени прости случаи, архитектурният стил на микроуслугите се появи през последните няколко години и все още има много текущи изследвания. И така, бъдещата работа по отношение на μσADL се отнася до подобряването и/или въвеждането на допълнителни процеси/конструкции/декларации/т.н. за по-нататъшно поддържане на различните им аспекти като пр. мащабируемоста (*scalability*).

Глава 5

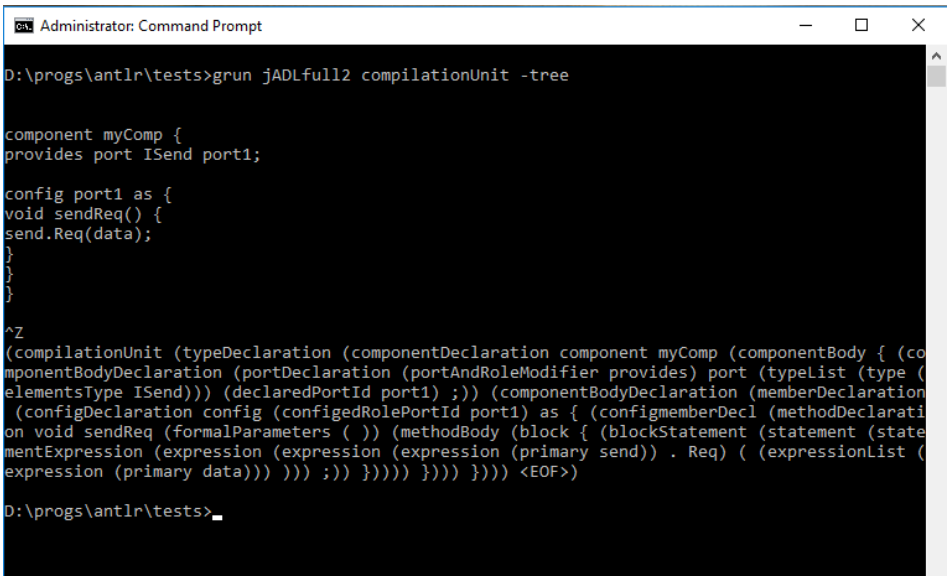
# Инструменти / Валидация

## 5.1 Въведение

В предишните глави jADL и неговото разширение за микроуслуги, μσADL, са представени. За да се улесни ползването им, е разработен инструмент. Инструментът има за цел да улесни използването им чрез осигуряване на средства за проверка на грешки в описанията, автоматични преобразувания и др. По време на това изследване бяха използвани две различни рамки (frameworks); първо ANTLR и след това Xtext. И двете са представени заедно с пример за валидация на езика.

## 5.2 Начален инструмент - ANTLR

Първият анализатор (parser) създаден по време на това изследване, е изграден с помощта на ANTLR (ANTLR 2014). Като вход изисква дефиницията на граматика, използвайки EBNF. Определената по това време граматика, не съответства напълно на окончателната версия на граматиката, представена в предишните раздели, тъй като ANTLR е използван в началото на това изследване, докато все още експериментирахме с граматиката.




Фиг. 9. *Абстрактно синтактично дърво от описанието, представено в текстов вид.*

При дефиниция на граматиката, ANTLR може автоматично да генерира анализатор, който може да изгражда и обхожда дървета. ANTLR предлага различни опции за анализ на абстрактното синтактично дърво. В нашия подход използвахме модела за дизайн Visitor Design Pattern (Gamma et al., 1994). След компилирането на създадените файлове, използвайки интерфейса от команден ред (CLI), можем да въведем архитектурното описание на jADL и да видим генерираното абстрактно синтактично дърво. Въпреки предимствата, които ANTLR предоставя за изграждането на анализатор (и други опции, които не са представени тук, като например възможността за интеграция в Java програма), решихме да променим рамката на Xtext.

## 5.3 Инструменти - Xtext

Продължавайки това изследване, решихме да променим използваната рамка на Xtext (Efftinge и Spoenemann, 2018). Xtext е разработен като plugin за Eclipse и предоставя ценни инструменти за проектиране на домейн специфични езици. Предлага възможност за автоматично получаване на анализатор и редактор за Eclipse, след описването на граматиката на езика. Също така, позволява писането на програми на Xtend (Bettini, 2013), подобен на Java език, който може да се използва за добавяне на допълнителни функционалности към правилата на граматиката.

### 5.3.1 Редактор

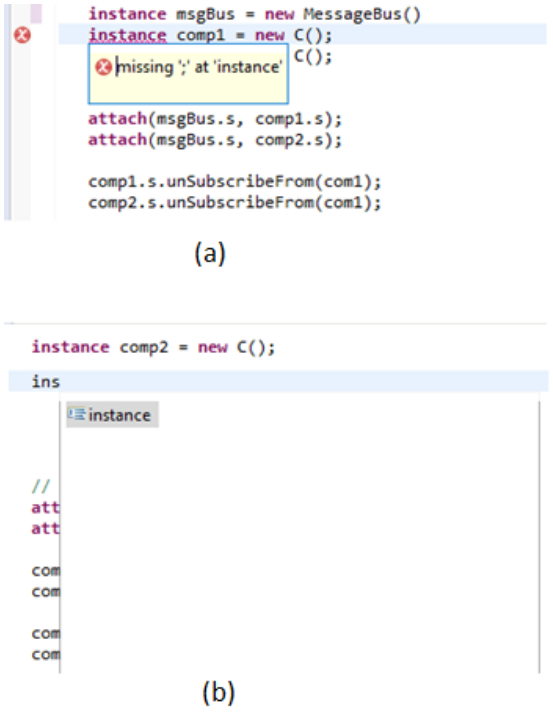
След дефиниране на граматиката на jADL се генерира редакторът за езика на Eclipse. Той автоматично поддържа типични функционалности за редактори (както например auto-completion), както е показано на фигура 10.

### 5.3.2 Преобразовател към π-ADL

За експериментиране, по време на това изследване, и като първа стъпка към генерирането на софтуерни артефакти, предложихме начин за достигане до програмен код на езика GO от jADL описание, използвайки π-ADL като междинен ADL. Както е показано в (Cavalcante et al., 2014), има генератор за GO код от спецификация на π-ADL. Следователно, ние създадохме преобразовател, за да автоматизираме процеса на преобразуване от jADL в π-ADL описание. На фигура 11 е представена разширена версия на таблицата от (Cavalcante et al., 2014). Добавена е информация относно архитектурните елементи в jADL, съответно към π-ADL и езика за програмиране GO. Процесът на трансформация за всеки един от тях е обяснен в този раздел.

*Компоненти и конектори*. И двата езика разглеждат компоненти и конектори като първокласни елементи (first-class entities) и следват класическата парадигма компонент/конектор/система. И в двата случая те се дефинират с помощта на ключовите

думи *component* и *connector*, последвани от идентификатор. Вътре в тези декларации и двата езика определят как елементът ще комуникира със средата си (портове / роли / интерфейси в jADL, връзки в π-ADL) и какво ще бъде неговото поведение (най-вече чрез *behavior* в π-ADL и *config* в jADL). И двете отговарят на *Functions* (*goroutine*) на езика за програмиране GO.




Фиг. 10. *(а) откриване на грешки, (б) auto-completion.*

*Поведение*. Точно едно поведение трябва задължително да бъде декларирано, за да се определи поведението на всеки архитектурен елемент в π-ADL. Използвайки ключовата дума *behavior*, поведението се дефинира в един блок от поредица от инструкции/декларации (напр. декларации за тип/променлива, функционални повиквания и т.н.) (Cavalcante et al., 2014). В jADL поведението се дефинира по различен начин; с използването на конструкцията *config*, поведението на всеки *provides* порт/роля се определя като набор от инструкции. Всяко допълнително поведение може да се определи вътре в тялото на архитектурния елемент. Така че, за да трансформираме поведението, ние събираме *config* декларациите (т.е. услугите, които те предоставят) и всякакви допълнителни изявления за поведение. След това, използвайки оператора choice, ги обединяваме в *behavior блок* на π-ADL код.

*Връзки*. В π-ADL се определят *връзки*, както за компоненти така и за конектори. Тези връзки са типизирани и ограничени в обхвата на архитектурния елемент. Те имат идентификатор, посока на връзката (in/out) и съществуващ тип. В jADL, от друга страна, разграничаваме връзките, отнасящи се до компоненти (портове) и тези, отнасящи си за конектори (роли).

Фактът, че можем да дефинираме N услуги в интерфейса на всеки порт/роля, добавя ненужна сложност, когато става въпрос за преобразуването им в връзки в π-ADL. За това, на този етап, когато става въпрос за преобразуване на описанието от jADL в π-ADL, допускаме точно една услуга на интерфейс. Това улеснява процеса на извличане на типа за всяка връзка. Останалите две свойства на всяка връзка се анализират от описанието на jADL; идентификатора и посоката (provides/requires – in/out) от декларацията на порта или ролята. Тези спецификации съответстват на *Канали* (*Channels*) на езика GO.

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

Фиг. 11. *Препечатано и разширено от (Cavalcante et al. 2014).*

*Архитектура*. Както jADL, така и π-ADL, след като дефинират всеки конкретен елемент, дефинират в отделна архитектурна декларация топологията на това архитектурно описание. И двете определят съответните инстанции и как те са свързани. В jADL това се дефинира чрез използване на ключовите думи *instance* и *new*, докато в π-ADL ключовата дума е *is* (Cavalcante et al. 2014). За връзките между тях, в jADL използваме оператора *attach* и в π-ADL ключовата дума *unifies* се използва между двете връзки. Важно е да се отбележи, че обединенията в π-ADL трябва да бъдат написани по специфичен начин (от изходна връзка на елемент към входна връзка на друг). По този начин, когато трансформираме команда за свързване в обединителен оператор в π-ADL, е важно да извлечем посоката на порта/ролята от оператора *attach*, така че да може да бъде поставен от правилната страна на оператора *unifies*. Декларацията на *Архитектурата* съответства на основната функция (*Main Function*) на езика GO.

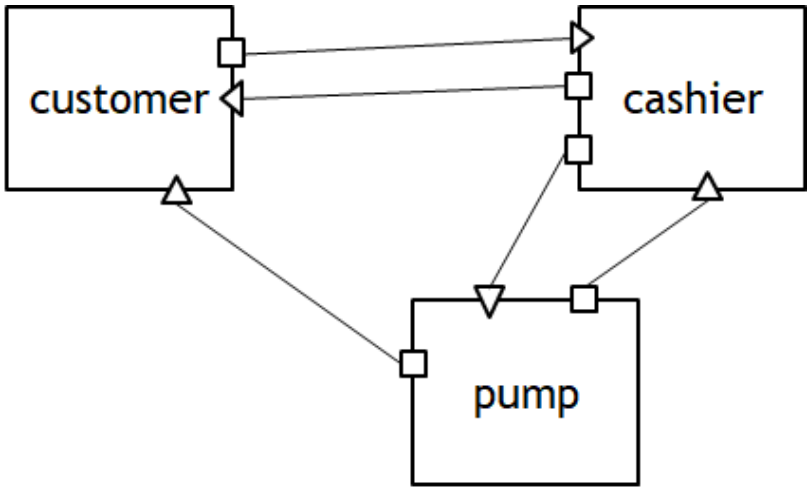
*Декларация и унифициране на връзките*. Декларациите и обединенията на връзките са дефинирани в *Архитектурата* за jADL и π-ADL, както е обяснено по-горе. Що се отнася до езика за програмиране GO, декларациите на такива връзки съответстват на Карти на Канали (*Maps of Channels*) и техните обединения с *Канали като Параметри* на *goroutines*.

Следвайки примера в (Cavalcante et al., 2014), ние дефинираме подобно просто jADL архитектурно описание. За да може генераторът да работи, синтаксисът на полученото π-

ADL описание трябва да няма синтактични грешки. Трябва да отбележим една съществена разлика между генерирания код и този в (Cavalcante et al., 2014) - липсата на *протоколи*. Протоколите в π-ADL се използват за налагане на типовите стойности, които трябва да бъдат предадени, и реда, в който операциите за изпращане/получаване трябва да бъдат извършени. По време на имплементирането на този трансформатор избрахме да ги пропуснем поради две причини; първо, те са незадължителни и второ, типът се декларира по време на декларацията за връзка и редът на операциите за изпращане/получаване е дефиниран по адекватен начин в поведението на всеки архитектурен елемент. Тъй като архитектурното описание на π-ADL е семантично и синтактично правилно, генерирането на GO програмен код беше успешно.

## 5.4 Пример за валидация на jADL

За валидацията на езика е представен пример (*case study*). Той се отнася до система за газ и се състои от 3 компонента; *client*, *cashier* и *pump* компонент, както е показано на фигура 12. Описаната архитектура е адаптирана към тази, представена в (Naumovich et al., 1997), заедно с модификацията от (Ozkaya, 2016). Модификацията се отнася до портовете и връзките между компонентите *client* и *pump*. В първия от тях имаше порт за всеки клиент в компонента *pump*, докато във втория и в този, представен тук, има един порт за свързване на множество клиенти.




Фиг. 12. *Графично представяне на системата за газ.*

Първо, използваните интерфейси се декларират в кодов фрагмент 8. Първият, *ICustomer*, се използва за комуникация между компонентите *client* и *pump* и *cashier*. Вторият, *IGas*, се използва за комуникация между компонентите *cashier* и *pump*. Дефинираме 2 интерфейса, тъй като различаваме първия тип комуникация (външен клиент) и втория, където *cashier* комуникира с *pump* (вътрешна комуникация). Услугите *payment* и *getGas* се използват от клиента, за да извърши плащане към касата и да направи заявка за газ към помпата.

**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. }
```

Фраг. от код 8. *Интерфейси за системата на газ.*

Компонента *client*, показан в кодов фрагмент 9, се състои от три порта: *rCash*, *pCash* и *rPump*. Чрез своя *rCash* порт, той изисква *pumpID*, след успешното приключване на плащането. След като плащането бъде обработено (в компонента *cashier*), то изпраща заявка до компонента *pump* (ред 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. }
```

Фраг. от код 9. *Описание на компонента client.*

Продължаваме с компонента *Cashier* (фрагмент на код 10), който се състои от четири порта: *rCust*, *pCust*, *rPump* и *pPump*. Чрез своя *pCust provides* порт, той приема заявка от клиента относно плащане. След като провери сумата, той изисква информация за следващата налична помпа от компонента *pump* и я изпраща на клиента. Също така, *pPump* порта е конфигуриран за предоставяне на отговор на компонента *pump* относно състоянието на плащането (успешно или не) на клиент.

**jADL Cashier Description**

```
1. component Cashier {
2.     requires port ICustomer rCust;
3.     provides port ICustomer pCust;
```

```
4.     requires port ICustomer rPump;
5.     provides port ICustomer pPump;

6.     config pCust as {
7.            service int payment(float amnt) {
8.                   if (check(amnt))
9.                          return rPump.getPump();
10.           }
11.    }
12.    config pPump as {
13.           service boolean checkOrder(int custid) {
14.                  if (check(custid))
15.                         return true;
16.                  else
17.                         return false;
18.           }
19.    }   }
```

Фраг. от код 10. *Описание на компонента cashier*.

Компонентът *pump* е показан след това, в кодов фрагмент 11. Чрез своя порт *pCash* той изпраща следващата налична помпа при всяка заявка от компонента *cashier*. В конфигурацията на порта *pCust* се обработват заявките от клиент (*custID*) за освобождаване на помпа (*pumpID*). Ако проверката за плащането на клиента е успешна (ред 12), помпата се освобождава за клиента.

**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.    }   }
```

Фраг. от код 11. *Описание на компонента pump*.

На края, архитектурното инстанциране се представя в фрагмент на код 12. Конекторите *SimpleConn* и *SimpleConn2* не са описани по-рано в раздела. Това е така, защото в този случай ние ги считахме за прости конектори за обмен на данни между компонентите. Всеки от елементите е инстанциран (редове 2-7) и накрая връзките са дефинирани (редове 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. }
```

Фраг. от код 12. *Архитектура на системата за газ.*

От по-горните дефиниции на трите компонента и цялостната архитектура на системата за газ е показано че jADL предоставя езиковите конструкции за адекватно изразяване на поведението на всеки от компонентите. Това се постига с използването на прости оператори, като например редове 7,8 в описанието на компонент *client*. Освен това, използването на добре познати от програмисти конструкции, като оператора *new*, и конструкции, чиито семантични значения са доста само-обясняващи се (като *attach*, използван за деклариране на връзка между порт и роля), могат допълнително да улеснят приемането и използване на езика в практиката.

## 5.5 Заключение

В тази глава беше представен инструмента, създаден за езика jADL и пример за неговата валидация. Първо беше представен първоначалният инструмент изграден с използването на ANTLR и бяха показани някои от неговите характеристики, като напр. извличане на визуално представяне на абстрактното синтактично дърво от текстово описание на jADL. На следващо място, рамката беше променена, тъй като преминахме към Eclipse Xtext, главно поради автоматизираните възможности, предлагани от рамката и нейната интеграция с Eclipse. Редактор беше получен след дефиницията на граматиката, който, има интегрирани стандартни функции за редактори (например откриване на грешки в синтаксиса) и също така може да бъде допълнително разширен с различни плъгини,

предвидени за Eclipse. Освен това е представен транслатора към π-ADL, който е използван за експериментиране на генерирането на код. π-ADL има инструмент за генериране на код на програмният език GO от своите архитектурни описания и затова е използван като междинен език. Във втората част на тази глава беше представен пример за оценка на jADL. Избран е примерът (*case study*), представен в (Naumovich et al., 1997), след като е допълнително модифициран от (Ozkaya, 2016). Той се отнася до система за газ и се състои от 3 компонента; *client, cashier* и *pump* компонент. Описани и обяснени са в jADL всеки компонент и цялостната архитектура на системата за газ. Езикът оказа адекватна подкрепа за описанието на архитектурата на системата.

# Глава 6

# Заключение

## 6.1 Обобщение на изследването

Това изследване започна от момента, в който се опитахме да определим, и след това да разгледаме проблема относно езиците за описание на архитектура (ADLs) и тяхното използване. ADLs са домейн специфични езици, използвани в областта на софтуерните архитектури и софтуерното инженерство. Те описват софтуерни архитектури от по-високо ниво и игнорират детайли относно по-ниското ниво на имплементация. Могат да осигурят средства за валидиране и верификация на дадена архитектура. Изследвания като (Ozkaya, 2016; Malavolta et al., 2012; Minora et al., 2012) относно архитектурните езици посочват два важни проблема: (1) високата степен на формалност, срещана в тези езици и (2) тяхното ниво относно поддръжката на динамични реконфигурации. Други проблеми могат да бъдат липсата на такъв език, който да описва конкретни архитектурни стилове, като микроуслуги (Francesco, 2017), или липсата на инструменти.

Извършен е обширен анализ на литературата, както е описано в глава 2. Поради големия брой съществуващи езици за описание на архитектури, за този анализ трябваше да бъде избрано подмножество от тях. Въз основа на споменатите резултати от изследванията, един от критериите за класифицирането и избирането на подмножеството беше тяхната подкрепа за динамична реконфигурация. Вторият, беше тяхната поддръжка за дефинирани от потребителя конектори. Смятаме, че последният е важен аспект на тези езици, тъй като би позволил разделянето между изчислителната и комуникационната част. След това е избрано подмножеството и резултатите са представени в глава 2.

Резултатите от този обзор помогнаха в дефинирането и поставянето на целите, описани в глава 1:

- *създаването на нов език за описание на архитектури, наречен jADL, който може формално да описва динамични архитектури, и същевременно използвайки сравнително прост синтаксис.*
- *възможността на езика да описва иновативни архитектурни стилове, като микроуслуги.*
- *разработване на инструмент за улесняването на ползването на jADL.*

В глава 3 jADL е представен и обяснен подробно. Обсъждат се неговите езикови конструкции и целият синтаксис. Дефинираният синтаксис наподобява широко използвани

езици за програмиране (пр. оператора *new* за създаването на архитектурни елементи) и се представя чрез EBNF. Въпреки приликата в някои техни конструкции, той е формален език за описание на архитектури, като в същото време се опитва да отговори на проблема посочен от обзора, че разработчиците на софтуер смятат архитектурните езици за твърде формални за да бъдат използвани в практиката. Той предоставя както текстов, така и графичен начин за описание на архитектури. Допълнително, в края на главата, чрез описанието в jADL на Message Bus Architectural Pattern, е представен пример за практическото използване на езика и различните му конструкции с които могат да се опишат динамични реконфигурации в архитектурата. Една от езиковите конструкции, която jADL въвежда, communication traits, се оказа вероятно най-полезна за описанието на динамични реконфигурации. Представлява сложна комуникационна структура, която позволява групирането на портове и роли, и може да се използва както по време на проектиране, така и по време на изпълнение на системата.

В глава 4, е разгледана втората дефинирана цел. Представен е μσADL, разширение на jADL специално за проектиране на софтуерни архитектури, които следват архитектурния стил на микроуслугите. Според твърдението на (Francesco, 2017), липсата на език за описание на архитектури за формалното описание на микроуслугите води до това архитектите да използват езици за моделиране на SOA, като SoaML. μσADL предоставя прости езикови конструкции, които могат адекватно да опишат архитектури на микроуслуги. Чрез допълнителен слой на абстракция, строги и твърде формални дефиниции се пропускат или са скрити зад прости изявления. Една проста поредица от присвояване на стойности води до създаването на формални архитектурни елементи, както е показано в глава 4. Освен това, е представен процес за практическото приложение на μσADL. Започвайки с една диаграма BPMN и следвайки три прости стъпки, може да се получи формално архитектурно описание. Представен е илюстративен пример относно описанието на система за пазаруване онлайн, чрез които са показани средствата които μσADL предоставя за просто, но също така и формално описание на архитектури на микроуслуги. Въпреки това, все още има въпроси които трябва да бъдат разгледани, както е описано в следващия раздел, касаещи, например, т.нар. *мащабируемост* (scalability) или определянето на *подробността* (granularity) на всяка микроуслуга.

В глава 5, е представен инструмент за jADL както и пример за валидирането му. Първо е представен инструментът създаден с помощта на Xtext framework. Възползвайки се от функционалности, предлагани от Xtext, беше създаден редактор, където може да се дефинират архитектурни описания на jADL. Редакторът е генериран с интегрирани типични функционалности като напр. auto-completion. Допълнително, е създаден трансформатор от jADL към π-ADL (за който е изграден генератор на GO програмен код), за да се използва за експериментиране с генерирането на софтуерни артефакти. Примерът, представен в края на главата за валидирането на jADL, показа, че езикът предоставя необходимите конструкции за описанието на изисканата архитектура на системата.

## 6.2 Приноси

Основните приноси на тази дисертация може да се класифицират като приложни и научно-приложни и са както следва:

- *Обзор и анализ на литературата.* Чрез обзор на литературата са показани основните проблеми, относно езиците за описание на архитектури (Architecture Description Languages-ADLs) и тяхното използване. В анализа извършен и представен в глава 2, се обсъждат предимствата и недостатъците на голяма част от съществуващите ADLs.

- Създаването на нов ADL, наречен jADL, който:

  - *може да описва динамични софтуерни архитектури*, предоставяйки средства за описание на динамични реконфигурации на дадена архитектура.

  - *осигурява лесен за възприемане и използване синтаксис*. Високата степен на формалност представлява един от основните проблеми, свързани с използването на такива езици, а jADL предоставя прост и добре познат за разработчиците на софтуер синтаксис.

  - *може да описва съвременни архитектурни стилове*, като архитектурния стил на микроуслугите, както е показано в глава 4.

- *Дизайн и разработка на инструмент за улесняване на използването на jADL.* В глава 5 е показан редактор, създаден за архитектурните описания в jADL, заедно с реализирания транслатор към π-ADL.

- *Описанието на широко използвани архитектурни модели.* В глава 3 са описани модели на популярните архитектурни стилове *self-adapting load balancer* и *message bus*.

- *Илюстративен пример за оценка на създадения език.* Общ пример, използван в областта на софтуерните архитектури, е представен в глава 5.

- *Процес за преобразуване на BPMN модели в jADL модели.* В глава 4 чрез илюстративен пример е показано как можем да стигнем до спецификация на jADL, като се започне от BPMN модел. Това ще помогне за повишаването на степента на използване на езиците за описание на архитектури, тъй като BPMN е широко използван в практиката.

## 6.3 Насоки за бъдещи изследвания

Както беше обсъдено в предишния раздел, основните цели поставени в началото, бяха постигнати успешно и като бъдеща работа могат да се изтъкнат следните насоки за бъдещи изследвания:

- разработването на графичен потребителски интерфейс за визуално дефиниране на архитектури, въз основа на графичното представяне на jADL, показано в глава 3.

- разработването на компилатор/генератор за jADL, така че да не е необходим междинен език.

- задълбочено валидиране на езика чрез провеждане на допълнителни експерименти (*case studies*).

- предоставяне на възможност за валидирането на различни аспекти на микроуслугите, като например т.нар. *подробност* (*granularity*) и *мащабируемост* (*scalability*).

# Библиография

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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