Mendelova univerzita v Brně
Provozně ekonomická fakulta

Komponentová architektura cílového software

Disertační práce

Školitel:
doc. Ing. Oldřich Trenz, Ph.D.

Ing. Oldřich Faldík

Brno 2017

Nejprve bych rád poděkoval svému původnímu vedoucímu disertační práce panu profesorovi RNDr. Milanu Mišovičovi, CSc. za nápad na zpracování aktuálního a zajímavého tématu komponentových systémů, za jeho vedení, konzultace, trpělivost, přínosné rady, podporu a čas, který mi věnoval v četných diskuzích. Chtěl bych poděkovat svému současnému vedoucímu disertační práce panu docentovi Ing. Oldřichu Trenzovi, Ph.D. za pomoc a podporu při zpracování této práce. Mé poděkování patří také panu profesorovi Johnu Fitzgeraldovi za umožnění studijního pobytu na Newcastle University ve Velké Británii a podílení se na projektu INTO-CPS. Dále chci poděkovat všem, kteří mi poskytli jakoukoliv pomoc a podporu v souvislosti s mou prací.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Komponentová architektura cílového software** vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 29. května 2017 .....

Abstract

Faldík, O. Component-based software architecture. Dissertation thesis. Brno, 2017.

The object of this dissertation thesis is to contribute to the development of formal analysis of component systems from the perspective of the system theory and computability theory. The thesis deals with various ways in which component systems are described and tasks verified. The first part of the thesis presents an overview of recent publications focusing on a description of component-based systems. This first part describes the basic terminology of component-based systems, such as the component itself, component architecture and component contract. The next section of the thesis comprises a literature review aimed at formalisms which can be used to describe component systems by means of Interface Automata. This is the most important section. The following sections describe the implementation of the principles of the theory of systems using a platform of sets, and specifications of the used terminology (component system, interface, interconnection, orchestration, conformity, configuration, recursivity of components) by means of formal definitions. The next section defines a specific Interface Automaton which allows the conversion of the component system to the Interface Automata and back, and the execution of specific tasks that are difficult to implement in UML. In order to support the conversion from UML to Interface Automata, a formal language is defined, the result of which is a protocol describing the component system in Interface Automata. The resulting language also supports the execution of typical tasks over component systems in the Interface Automata-based language. The thesis further deals with the various Interface Automata applications in the field of Cyber-physical systems (System of Systems). Finally, the improvement to the Contract pattern in use is proposed and the extended Interface Automaton is defined as one of the System of Systems verification options.

Abstrakt

Faldík, O. Komponentová architektura cílového software. Disertační práce. Brno, 2017.

Cílem této práce je přispět k vývoji v oblasti formální analýzy komponentových systémů ve dvou směrech, a to z pohledu teorie systému a teorie vyčíslitelnosti. Práce se zabývá možností deskripce komponentových systémů a verifikačními úlohami nad těmito systémy. V první části disertační práce je uveden přehled současných publikací, které se zaměřují na verbální deskripci komponentových systémů. V této kapitole jsou popsány základní pojmy z oblasti komponentových systémů, jako je například komponenta, komponentová architektura a kontrakt komponenty. V další části disertační práce je uveden přehled publikací, které se zaměřují na formalismy využitelné k deskripci komponentových systémů, zde je nejvýznačnější část, která se zabývá deskripcí komponentových systémů pomocí interface automatů. Dále následují kapitoly, které vedou k naplnění cíle práce. Je to kapitola, která uplatňuje prostřednictvím množinové platformy zákony teorie systémů a formálními definicemi upřesňuje používané pojmy (komponentový systém, rozhraní, propojení, orchestrace, konformita, konfigurace, rekurzivita komponent). Další kapitola se zaměřuje na definici specifického interface automatu, který umožňuje, převod komponentového systému do interface automatů a zpět a vykonání specifických úloh, které lze jen s obtížemi v UML realizovat. Pro podporu převodu z UML do interface automatů je definován formální jazyk, jehož výsledkem je protokol, popisující komponentový systém v interface automatech. Výsledný jazyk také podporuje provádění typických úloh nad komponentovými systémy v jazyce interface automatů. Dále se disertační práce zabývá možnostmi uplatnění interface automatů v oblasti Cyber-physical systems (System of Systems). V této části je navrženo vylepšení používaného Contract pattern, dále je definován rozšířený interface automat jako jedna z možností verifikace System of Systems.

Obsah

1 Úvod **9**

2 Pojetí komponent a komponentových systémů **14**
2.1 Verbální styl deskripce komponentových systémů 14
2.2 Specifikace komponenty 15
2.3 Kontrakty komponent 17
2.4 Komponentová architektura 17
2.5 Komponentový model 18
2.6 Komponentový Framework 18
2.7 Odborné pohledy na komponentu 18
Komponenta a funkcionalita 18
Komponenta – jednotka komponentové architektury 19
Komponenta – jednotka distributivity 19
Komponenta – jednotka pro sestavování aplikací (assembling) 19
Komponenta podléhá pravidlům objektového paradigmatu 21
Komponenta a její komplexita 21
Komponenta a její implementace 21
Komponenta a její opakovatelná využitelnost (reusing) 21
Stav komponenty 22
Specifikace komponenty podrobněji 22
Implementace komponenty versus její specifikace 22
Komponentový systém 23
2.8 Integrace a kompozice komponent 23
2.9 Vývoj a deskripce komponentových systémů 23
2.10 Přehled známých operací nad komponentami a komponentovými systémy 23

3 Formalismy využitelné k deskripci komponentových systémů **25**
3.1 Využití formálních jazyků 25
3.2 Jazyk UML 25
Struktura jazyka UML 26
Předměty 27
Relace 27
Diagramy 28
Nedostatky vytýkané UML 28
3.3 Jazyk OCL 29
Příklady integritních omezení v OCL: 30
3.4 Využití teorie vyčíslitelnosti 31
Publikace (Alfaro, Henzinger, 2001): 31
Publikace (Isazedech, Karimpour, 2008): 34

4 Co bych chtěl v disertační práci rozvinout **37**

5 Cíl disertační práce 39

6 Metodiky pro naplnění cíle disertační práce 40

7 Komponentové systémy a teorie systémů 41
7.1 Systémové úlohy nad komponentovým systémem 44
7.2 Úloha o konstrukci komponentového systému 46
7.3 Popis pracovních postupů metodiky E-P+MM+OM 49
Logická architektura 49
Návrhová architektura 50
Fyzická architektura 52
Architektura nasazení 53
Krok architektonická implementace 54
Krok tvorba Diagramu nasazení a Modelu nasazení 54
7.4 Úlohy verifikace komponentového systému 55
7.5 Analýza interface a propojení dvojic primitivních komponent 55
7.6 Analýza interface a propojení dvojic komponent 57
Úloha o servisní logice (hledání funkcionálních řetězců komponent) . 61
7.7 Hledání funkcionálních řetězců komponent 62
Úloha o orchestraci 63

8 Využití interface automatů v modelování komponentových systémů 65
8.1 Formální definice událostní komponenty 66
8.2 Formální definice interface automatu 68
8.3 Nástroje pro vývoj interface automatů 71
8.4 Design pro D-Framework 73
8.5 Využití formálního jazyka Ψ pro vývoj interface automatů 74
8.6 Implementace D-Frameworku v jazyce Java EE 79

9 Využití interface automata v modelování kontraktů v System of Systems 81
9.1 Contract pattern 82
9.2 OCL jako rozšíření SysML v Contract pattern 83
Contract Definition Viewpoint 83
Contract Protocol Definition Viewpoint 85
9.3 Verifikace kompatibility kontraktů pomocí interface automatů 85
9.4 Překlad The Leader Election Case Study do *rozšířených interface automatů* 87

10 Diskuze 94
10.1 Navržené vylepšení 95
10.2 Ekonomický přínos 95

11 Závěr 99

12 Reference **100**

1 Úvod

Na počátku vzniku výpočetní techniky byly problémy řešeny jednoduchými a nezávislými algoritmy. Později, v důsledku dalšího rozvoje a integrace informačních technologií do lidského života, se řešení problémových domén stává čím dál častěji robustním. Jeden z klíčových mezníků v oblasti informačních technologií je jejich zavádění do podnikové sféry, které je spojováno se vznikem informačních systémů. Tyto systémy optimalizují tok informací a zajišťují, aby se informace dostaly na správné místo ve správný čas.

Vývoj společnosti je zásadně ovlivněn informačními technologiemi. Tato skutečnost má za následek přechod od industriální k informační společnosti. Bity čím dál rychleji nahrazují atomy v tom nejobecnějším slova smyslu. Lidé mají také tendence filtrovat informace, které přijímají, a to v osobní i podnikové sféře. (Negroponte, 2001)

V důsledku takto silné integrace informačních technologií do běžné lidské činnosti roste i variabilita komputerizace jednotlivých problémových domén v závislosti na účelu, nákladech atd.

Jeden z hlavních důsledků vývoje v informačních technologiích byl přechod ze strukturovaného k objektovému paradigmatu v oblasti informačního modelování problémových domén. Strukturované paradigma si s rostoucí komplexností řešeného problému sice umí poradit, ale na druhé straně se mohou výsledky jevit jako nepřehledné, a to i v případě rozdělení jednotlivých programových částí do procedur a funkcí. Nepřehlednost může způsobit špatné uchopení problému, což může implikovat chyby, a to jak implementační, tak i návrhové. (Allen, Garlan, 1996)

Objektové paradigma je založeno na vztazích a principech reálného světa. Vychází z předpokladu, že problém, který řešíme, zahrnuje objekty a vztahy mezi nimi. Následné řešení spočívá v manipulaci (pomocí jejich operací) s těmito objekty (změna jejich stavu).

Objektové paradigma sice zjednodušilo implementační část programování, avšak v oblasti návrhové byly pořád nedostatky. Nejvíce problematické se ukázalo přidávání funkcionalit do již existujícího systému. To v praxi mnohdy znamenalo a znamená provést mnoho nákladných zásahů do celé aplikace, které mnohdy měly za následek nekontrolovanou distribuci chyby. Z tohoto důvodu a dalších, jako je úspora nákladů, optimalizace návrhu aplikace z hlediska rozšiřování funkcionality nebo také z důvodu funkční škálovatelnosti SW produktu – dle potřeb zákazníka – se ukázalo jako vhodné členit SW aplikaci do dílčích celků dle funkcionality, kterou poskytuje. Takové celky se pak nazývají moduly. V průběhu dalšího vývoje termín modul, vzhledem k častému rozdílnému výkladu, přešel do termínu komponenta, která má jasně definované atributy, a tento termín se používá dodnes.

V tvorbě softwarových systémů se v posledních třech desetiletích prosazuje pojetí a uplatňování komponentových architektur. Čím dál tím složitější požadavky jsou kladeny na softwarové komponentové systémy, které se týkají zejména jejich dy-

namických vlastností. Vznik takových požadavků si postupně vynutila praxe vývoje a implementace těchto systémů, zejména pro software informačních systémů.

Software požadovaných kvalit a typu robust nemůže být monolitem. Je všeobecně známo, že vývoj návrhu software informačních systémů dospěl jednoznačně k nutnosti jeho složení ze zcela autonomních, ale spolupracujících architektonických jednotek, které navzájem komunikují pomocí zpráv předepsaných formátů.

Ačkoliv za takové jednotky byly často používány tzv. podsystémy a moduly, viz (Jacobson, Booch, Rumbaugh, 1998) a (Arlow, Neustad, 2007), uzákonila se postupně jejich abstrakce na termínu komponenta. Jinými slovy, podsystémy a moduly jsou specifické typy komponent.

Komponenta především reflektuje adaptabilitu procesů různých domén, vysokou distributivnost, vysoké nároky na integritu domény (procesní, datová a komunikační integrita), škálovatelnost a pružné přizpůsobení procesním změnám, dobrý kontext na externí zařízení a transparentní strukturu na dílčích procesních modulech a architektonických jednotkách.

Existují dva typy softwaru informačních systémů, viz (Král, Žemlička, 2000) a (Král, Žemlička, 2003). První je SWC (Software Components), složený z permanentně dostupných komponent, které jsou považovány za služby – konfederační software. Druhým typem jsou SWA (Software Alliance), tzv. semi-konfederační, které se formují v průběhu softwarového systému a označují se jako softwarové aliance.

V obou zmíněných publikacích je podána hluboká filosofie relevantních problémů týkajících se SWC/SWA, jako je vytváření kopií komponent (klonování), zřizování a ničení komponent v době běhu software (dynamická rekonfigurace), spolupráce zcela autonomních komponent, správa programovatelných rozhraní jednotlivých komponent v závislosti na interní funkcionalitě komponent a požadavcích zákazníka (funkcionalita, bezpečnost, verzování).

V dnešní době se již setkáme s mnoha případy existence SWC/SWA s velmi rozvinutou architekturou, která vyhovuje převážné většině zmíněných požadavků. Na druhé straně, v praxi vývoje komponentových systémů s dynamickou architekturou (tj. architekturou s dynamickou rekonfigurací) a dle (Rychlý, Weiss, 2008) s mobilní architekturou (tj. dynamickou architekturou s mobilitou komponent – component mobility) se potvrzuje nedostatečnost návrhových metod obsažených v UML 2.0.

Softwarové inženýrství v současné době disponuje se dvěma odlišnými přístupy k systémům SWC/SWA. První přístup je označován jako komponentově orientovaný vývoj softwaru CBD (Component-based Development). Podle (Szyperski, Gruntz, Murer, 2002) je CBD kolekcí metodik, které jsou silně orientovány na možnosti sestavení a znovupoužitelnosti částí softwaru v rámci jeho architektury. Ačkoliv se CBD neprokazuje vysokým teoretickým přístupem, je zařazen pod obecným vývojem softwaru SDP (Software Development Process), viz (Sommerville, 2010) jako jeden ze dvou dominantních směrů. Klasické procesní, objektové metodiky jsou výrazně zaměřeny na informační modelování problémových domén, a výsledkem jejich fáze *Rozpracování* je komponentový systém domény jako předobraz pro jeho napro-

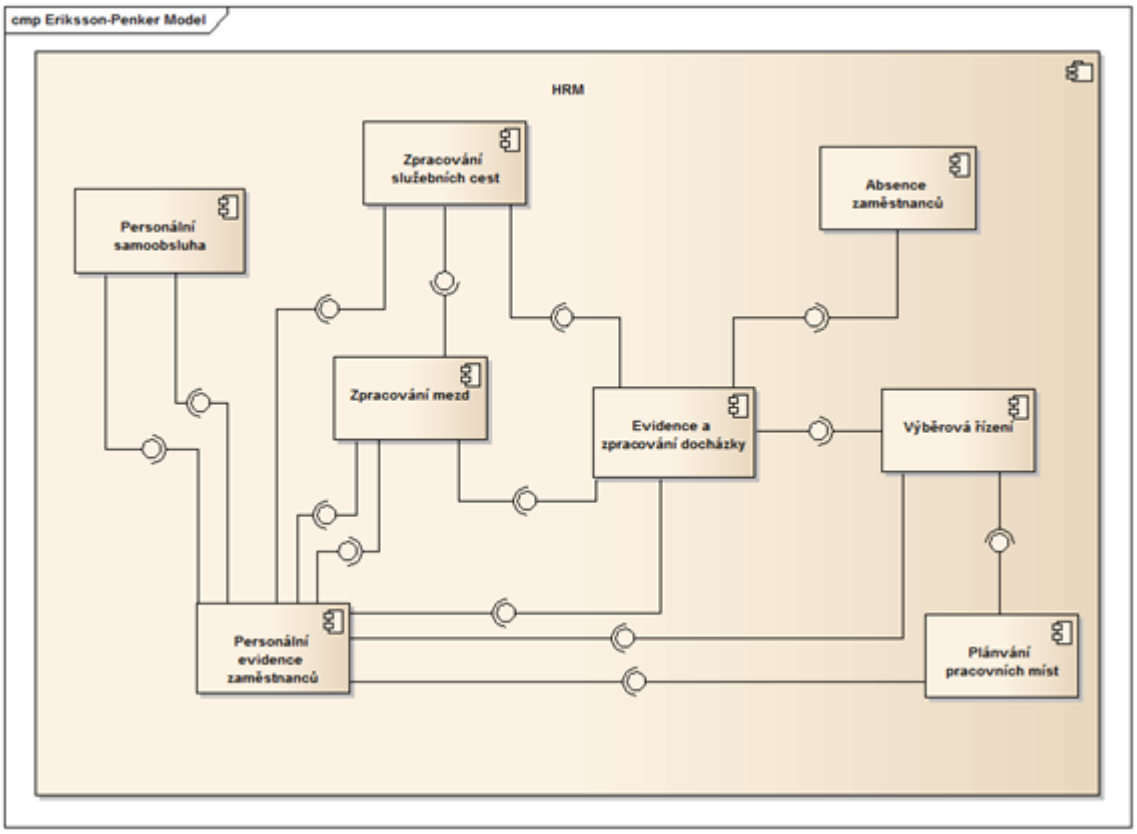
gramování do softwarové podoby. Ačkoliv je řízení problematiky realizace (naprogramování) součástí objektové metodiky, metodika nemá nástroje k jeho provedení.

Ze strukturálního hlediska je softwarový systém složen ze soběstačných, vzájemně spolupracujících architektonických jednotek – komponent na základě dobře definovaných rozhraní. Klasické procesní, objektové metodiky výrazně nevyužívají tzv. komponentových metamodelů, na základě kterých se potom tvoří výsledné komponentové systémy. Na druhé straně, tzv. komponentové modely popisují syntax, sémantiku komponent, jsou systémem pravidel pro komponenty, konektory a konfiguraci. Komponentové modely pro dynamickou a mobilní architekturu rovněž popisují koncepci pravidel pro změny konfigurace (rules for reconfiguration). Za velmi známé metamodely se dnes považují modely Wright pro statickou architekturu, SOFA a Darwin pro dynamickou architekturu a SOFA 2.0 pro mobilní architekturu, viz (Rychlý, Weiss, 2008).

V CBD přístupu jsou verbálně definovány základní termíny jako komponenta (primitivní/složená), rozhraní, komponentový systém, konfigurace, rekonfigurace, logický (structural) pohled, procesní (behavioural) pohled, statická komponentová architektura, dynamická architektura, mobilní architektura (fully dynamic architecture), viz (IEEE, 2000) a (Crnkovic, 2006).

CBD přístup rovněž předkládá několik jazyků ADL (Architecture Description Languages), kterými je možné popsat architekturu softwaru. Známé jsou tyto jazyky: integrační ACME a účelový UML (Unified Modeling Language), viz (Garlan, 2000) a (ISO/IEC 19501, 2005). Druhý přístup k systémům SWC/SWA je formován na základě architektury SOA (Service-oriented Architecture).

K zobrazení komponentových systémů jsou v této práci použity notace zavedené v UML 2.0. Obrázek (viz obr. 1) tyto notace ilustruje.



Obrázek 1: Komponentový diagram pro HRM (Human Resource Management) malého podniku (1. vrstva procesů) (Faldík, 2014)

Komponentovému přístupu k tvorbě softwaru předcházelo několik užitečných technik a metod. Všechny byly orientovány na zvýšení přehlednosti, čitelnosti a možnosti modifikace kódu. Zejména komputerizace podnikové procesní sféry požadovala, aby zmíněné vlastnosti byly dosaženy a umožnily podniku změněným softwarem podepřít nové výrobní a obchodní programy. Bylo již zřejmé, že robustní monolity zmíněné požadavky podnikům nezaručí.

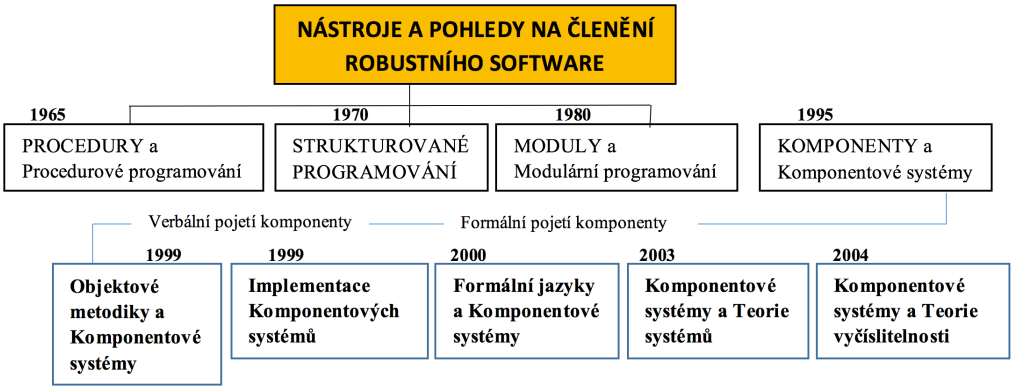
První známou technikou bylo tzv. Strukturované programování. Cílem této techniky bylo zachovat přehlednou strukturu výsledného programu zejména pomocí blokace takových příkazů, jako je GOto a IF. Robustnost výsledného softwaru však byla velkou překážkou praktické využitelnosti této techniky.

Významnou metodou konce 70. let bylo tzv. Modulární programování. Metoda spočívala v členění výsledného softwaru na dílčí moduly. Moduly vystupovaly samostatně s pevně navrženou strukturou a vzájemnou spoluprací v celém systému. V teoretických poznatcích byla metoda podepřena teoretickým modulárním systémem, který byl postaven na několika binárních a n-árních relacích nad moduly systému.

Nejobtížnější skutečností ovšem byla kódová konstrukce a implementace tzv. *Framework*, tj. podpory modulárního programování buď v programovacím jazyku nebo ve vznikajících operačních systémech počítačů. Ani v průběhu 80. let nedosáhla implementace modulárního programování výrazných úspěchů.

Objektové paradigma, nastupující postupně koncem 80. let, muselo začít chápat členění softwaru v souladu se svými základními pilíři a materiálem, kterým byly objektové třídy. Výsledný software je členěn na komponenty, které jsou objektovými třídami se speciální vzájemnou komunikací.

Na vývoj nástrojů a pohledů pro členění robustního software se můžeme dívat prostřednictvím následujícího schématu (viz obr. 2).



Obrázek 2: Ilustrace nástrojů a pohledů v implementaci problematiky členění robustního software

Předložená práce se zabývá specifickou aktivitou v CBD, tj. souladem pojetí vývoje komponentového systému výsledného softwaru a pokročilé procesní, objektové metodiky (Arlow, Neustad, 2007), (Kanisová, Muller, 2007), (Kruchten, 2003) u problémových domén s vícevrstvou procesní logikou. Je ukázána metoda sjednocení, založená na navrženém metamodelu Komponentový vývoj softwaru, vedoucí k tvorbě komponentového systému složeného z více navzájem souvisejících komponentových diagramů a pracovních postupů objektové metodiky UP. Tento metamodel byl široce diskutován v publikacích (Mišovič, Faldík, 2013) a (Mišovič, Faldík, 2013a). Zmíněný metamodel je členěn na pracovní postupy (Logická architektura, Návrhová architektura, Fyzická architektura a Implementační architektura), pro které jsou dodávány vstupní artefakty různých pracovních postupů z fází klasické, procesně založené objektové metodiky. Je brán důsledný ohled na konzistenci vstupních a výstupních artefaktů v pracovních postupech metamodelu a zmíněné objektové metodiky.

Vedle toho předložená práce chápe komponentový systém jako specifický systém, na který je možné pro notace jeho systémových vlastností a základních termínů použít množinovou, grafovou a systémovou algebru. Základem tohoto přístupu jsou různé formalizace vlastností komponentových systémů, které budou rozebírány později.

2 Pojetí komponent a komponentových systémů

Rozbor stávající situace s komponentovými systémy poslouží ke stanovení cílů disertační práce. *Každý komponentový diagram nutno považovat za systém*, protože je složen z komponent, které navzájem interagují. Je tedy zřejmé, že u komponentových diagramů se uplatní všechny systémové atributy. Pochopitelně, jestliže je pro modelovanou doménu vypracováno několik vzájemně souvisejících komponentových diagramů, často hierarchické podoby, vytvořil se tak výsledný komponentový systém domény.

Obecné důvody zavádění vývoje software (Development Process) ve formě komponentových systémů (Component-based Systems) s hierarchickou povahou jsou dány praxí uplatnění software a jejími požadavky. Je mnoho problémových domén, které jsou charakterizovány jak vícevrstvými procesními diagramy, tak i vícevrstvými komponentovými systémy, které tyto domény komputerizují.

S uplatněním směru formálních jazyků prostřednictvím teorie vyčíslitelnosti jsem se setkal v publikacích (Alfaro, Henzinger, 2001), (Alfaro, Henzinger, 2004), (Brim, Černá, Vařeková, Zimmerová, 2006) a (Isazedech, Karimpour, 2008).

2.1 Verbální styl deskripce komponentových systémů

Dle (Brown, 2000) je komponenta užitečný *fragment softwarového systému, který může být propojený s ostatními fragmenty softwarového systému.* Množiny těchto fragmentů formují větší fragmenty nebo přímo kompletní softwarové řešení. (Crnkovic, Larson, 2002) definují komponentu jako *znovupoužitelnou jednotku nasazení a kompozice, jejíž funkcionalita je přístupná prostřednictvím rozhraní.*

(D'Souza, Wills, 1999) definují komponentu jako *znovu spustitelnou část softwaru, která je nezávisle vyvinuta a může být kombinována s jinými komponentami, aby tak vytvořily větší jednotky.*

Další z mnoha definic (Szyperski, Gruntz, Murer, 2002) definuje komponentu jako *jednotku kompozice se smluvně dohodnutým rozhraním a explicitním kontextem závislosti a softwarová komponenta je umístěna nezávisle a je subjektem pro kompozici dalších komponent.*

Komponenty mohou pocházet ze tří zdrojů (Crnkovic, Larson, 2002):

- vlastní vývoj - pocházejí z vlastních zdrojů
- již existující komponenty - například použité v jiném systému
- komerční - zakoupené od specializované SW firmy

Na komponenty lze nahlížet ze tří pohledů:

- z pohledu balíčků, kde je komponenta jednotkou určenou k distribuci a nasazení
- z pohledu, že komponenta poskytuje službu.
- Poslední pohled definuje komponentu jako datovou integritu nebo zapouzdřující hranici.

UML OMG specifikace (ISO/IEC 19501, 2005) definuje komponentu jako *„fyzickou a zaměnitelnou část systému, takže implementuje a poskytuje množinu rozhraní.“* Tato definice se řadí do pohledu, který vnímá komponentu jako balíček. UML dále rozlišuje určité druhy komponent, mezi které patří komponenty jako spustitelné jednotky, dokumenty, soubory, knihovny a tabulky. Z pohledu komponenty jako služby, která definuje komponentu jako softwarovou jednotku, která poskytuje službu, je kladen důraz na kontrakty mezi poskytovatelem a spotřebitelem množiny služeb. Služby mohou být shlukovány do koherentních, kontraktních jednotek, tzv. rozhraní. Komponenta je tedy v tomto pojetí jako softwarový balíček, který poskytuje služby prostřednictvím rozhraní. (Brown, 2000)

Servisní pohled na komponentu je logickou notací komponenty. Servisní pohled na komponentu dál rozlišuje specifikaci komponenty a implementaci komponenty. Specifikaci komponenty lze určit podle toho, že definuje, co komponenta dělá. Implementace definuje, jak to komponenta dělá. Tento rozdíl mezi specifikací a implementací je důležitý, protože spotřebitel by měl být závislý jen na specifikaci dané komponenty. Jakákoliv závislost na implementaci může způsobit nekonzistentnost systému při náhradě nebo úpravě komponenty. (Brown, 2000)

Z hlediska pohledu integrity je důležité, aby byla stanovena hranice záměny jedné komponenty za jinou. Komponenta tedy představuje nezávislou, nahraditelnou jednotku chování. Z pohledu integrity je komponenta definována jako implementační zapouzdřená hranice. (Brown, 2000)

CS/3.0 standard firmy Sterling Software definuje komponentu jako *nezávislý, poskytnutelný balíček softwarových operací, který může být použit k vytváření aplikací nebo velkých komponent.* Důraz na nezávislost je důležitý vzhledem k tomu, že servisní pohled na komponentu nevyžaduje implementační nezávislost.

Na základě předchozích definic komponenty můžeme odvodit společné vlastnosti verbálních definic:

- Komponenta je jednotkou kompozice v komponentovém celku. Integrace je provedena na základě rozhraní.
- Komponenta je integrována, udržována a měněna.

2.2 Specifikace komponenty

Specifikace komponenty popisuje chování dané komponenty. Většinou se k popisu chování komponent se používá seznam názvů operací s jejich vstupními a výstup-

ními parametry, textový popis funkcionality komponenty, případného užití a historii vývoje. Jako další z atributů specifikace komponenty se používá popis prostředí, ve kterém bude komponenta pracovat, například operační systém. Poslední část specifikace komponenty je výkonnost a dostupnost při typickém provádění komponenty (Brown, 2000).

(Brown, 2000) dále definuje: *„Specifikace komponenty většinou zahrnuje množinu služeb komponentou nabízených, které jsou sjednoceny do významových klastrů. Tyto klastry služeb jsou nazývány rozhraní.“* Rozhraní deklaruje množinu operací, které mohou být po komponentě požadovány. Je důležité poznamenat, že rozhraní komponenty nenabízí žádnou implementaci jakékoliv ze svých operací. Rozhraní specifikuje jediný přístup ke komponentě.

Dle (Crnkovic, Larson, 2002) komponenta může být specifikována pomocí rozhraní, které musí být přesně vytvořeno podle operací komponenty. Rozhraní musí být pro celek standardizováno, aby mohlo být provedeno znovupoužití komponent. Klient spravuje komponenty pomocí rozhraní, protože rozhraní je jediná viditelná část komponenty.

Podle (Crnkovic, Larson, 2002) by specifikace komponenty měla obsahovat:

- specifikaci nefunkcionálních vlastností, které jsou buď poskytovány nebo požadovány
- test kódu, který potvrzuje navrženou konexi k dalším komponentám
- dodatečnou informaci, která obsahuje dokumenty spojené s plněním specifikace požadavků.

Velmi užitečná implementační definice komponenty je uvedena v UML 2.0.

Komponenta představuje modulární část systému, jež zapouzdřuje svůj obsah, a v rámci daného prostředí jsou její projevy nahraditelné. Komponenta definuje své chování v podobě poskytovaných a požadovaných rozhraní. Jako taková slouží komponenta jako typ, jehož konformita je definována právě těmito poskytovanými a vyžadovanými rozhraními. Jedna komponenta tedy může být nahrazena druhou komponentou pouze tehdy, pokud obě jsou typově konformní.

Existují ovšem i jiné definice komponenty. Např. (Brown, 2000) považuje za významné následující užitečné pohledy:

1. *komponenta je část software se stanovenou funkcionalitou, která je opakovaně využitelná a poskytuje uživateli stanovené služby (role – nese funkcionalitu jako služby)*
2. *komponenta je jednotkou pro distributivitu (role – komponenta pohybující se mezi poskytovatelem a konzumentem - distributivita).*

V literatuře, (Brown, 2000), (Crnkovic, Larson, 2002) se jako relevantní uvádějí ještě tyto pohledy:

1. *komponenta je rovněž základním stavebním prvkem aplikací sestavených z použitých komponent, je rovněž základním prvkem softwarové architektury (role – stavební prvek aplikací)*
2. *komponenta podléhá zákonům objektového paradigmatu (role – hraje specifickou objektovou třídu)*

2.3 Kontrakty komponent

Kontrakty jsou upřesňující specifikace chování komponent. Kontrakt obsahuje obecná omezení, která budou u komponent zachována, a to které vyžaduje klient, a které vyžaduje komponenta. Dále kontrakt může být také využit ke specifikaci interakce mezi různými skupinami komponent. Kontrakty specifikují interakce mezi komponentami pomocí:

- množina zúčastněných komponent
- role každé komponenty pomocí kontraktových povinností
- neměnnost, která musí být zachována u komponent
- specifikace metod, které dokládají kontrakt

Celkové chování komponenty může být docela komplexní, protože se může podílet na mnoha kontraktech.

2.4 Komponentová architektura

Komponentová architektura je výsledným produktem komponentově založeného vývoje software (CBD). Komponentová architektura podle (Crnkovic, Larson, 2002) je definována jako *Softwarová architektura programů, nebo výpočetních systémů je strukturou, která zahrnuje softwarové komponenty, viditelné vlastnosti těchto komponent a vzájemné vztahy mezi nimi.*

Funkcionalita a chování komponentového systému je definována na základě vlastností funkcionalit jednotlivých komponent.

Existují tři základní typická užití softwarové architektury:

- posuzování a hodnocení
- řízení konfigurace
- dynamická softwarová architektura

2.5 Komponentový model

Podle (Crnkovic, Larson, 2002) je *komponentový model množinou všech typů komponent, interface, různých rozhraní a specifikací povolených vzorců interakcí mezi komponentovými typy.* Dle (Brown, 2000) je komponentový model definován jako *množina standardů pro návrh, instalaci a nasazení aplikací po částech.* Tato definice zahrnuje, jaké jsou po jednotlivých komponentách vyžadovány služby, jak komponenta informuje další komponenty o své existenci a jak budou komponenty mezi sebou komunikovat. Mezi hlavní komponentové modely patří COM+, COBRA a Java/EJB.

2.6 Komponentový Framework

Poskytuje běhové služby pro komponenty. Je to běhový rámec komponentového systému. Slouží k redukci komplexnosti výsledné aplikace a umožňuje vyhnout se psaní stejného kódu. Na základě definice (Crnkovic, Larson, 2002) se mohou frameworky dělit na white-box a black-box. White-box se vyznačuje tím, že vývojář musí provést rozšíření třídy a implementaci rozhraní, zatímco black-box se skládá z komponent a tříd, od kterých může vývojář odvodit objekt a provést jeho konfiguraci.

Dle (Crnkovic, Larson, 2002) se komponentový Framework skládá z těchto částí:

- Návrhové dokumenty – zahrnují diagram tříd, verbální popis nebo pouze myšlenku vývojáře.
- Úloha rozhraní – specifikuje externí chování tříd.
- Abstraktní třídy – jde o částečnou implementaci jednoho nebo více rozhraní.
- Komponenty – v implementaci je takový rozdíl mezi komponentou a třídou, že komponenta je třída, která je dodána se svým API. V tomto případě API představuje rozhraní.
- Třídy – jsou na nejnižší úrovni abstrakce ve frameworku.

2.7 Odborné pohledy na komponentu

Dostupná odborná literatura nabízí mnoho užitečných pohledů na komponenty. V dalším textu jsou uvedeny některé z nich.

Komponenta a funkcionalita

Toto je kardinální pohled. Z něj plyne, že komponenta komputerizuje část funkcionality podniku (což by mělo být prosto redundance), že musí být opakovatelně

využitelná (nelze připustit pro každé jednotlivé využití jinou implementaci). Výsledek informačního modelování stanovuje, jak rozsáhlá je služba, kterou komponenta poskytuje zákazníkovi (např. v procesech).

Komponenta – jednotka komponentové architektury

Základním zdrojem pro formování komponent jsou podnikové procesy. Často je podnik členěn na dílčí subjekty – procesní podsystémy a každý z nich má svou procesní množinu. Potom je procesní logika podniku dána konečným počtem procesních fragmentů a komunikacemi mezi nimi. Předpokládejme, že máme komputerizovat jeden, vybraný subjekt S_i. Jinými slovy, procesy a zpracování dat v tomto subjektu máme realizovat pomocí softwaru, který může být libovolné architektury. Pro tento vývojový proces musíme použít informační modelování pomocí pokročilé objektové vývojové metodiky s dominancí procesů a komponentové architektury. Je rovněž jiná možnost, a to pomocí metamodelu komponentového vývoje, který využívá výsledky pracovních postupů především, klasické objektové metodiky.

Komponenta – jednotka distributivity

S příchodem Webu 2.0 přišly i nové možnosti distributivity. Software je možné dodávat více způsoby. Můžeme se také setkat s poskytnutím služby - pronájmem komponenty. V tomto případě služba běží na straně poskytovatele a zákazník je ušetřen nákladů na provoz. To se v případě software komponenty realizuje pomocí tzv. webových služeb.

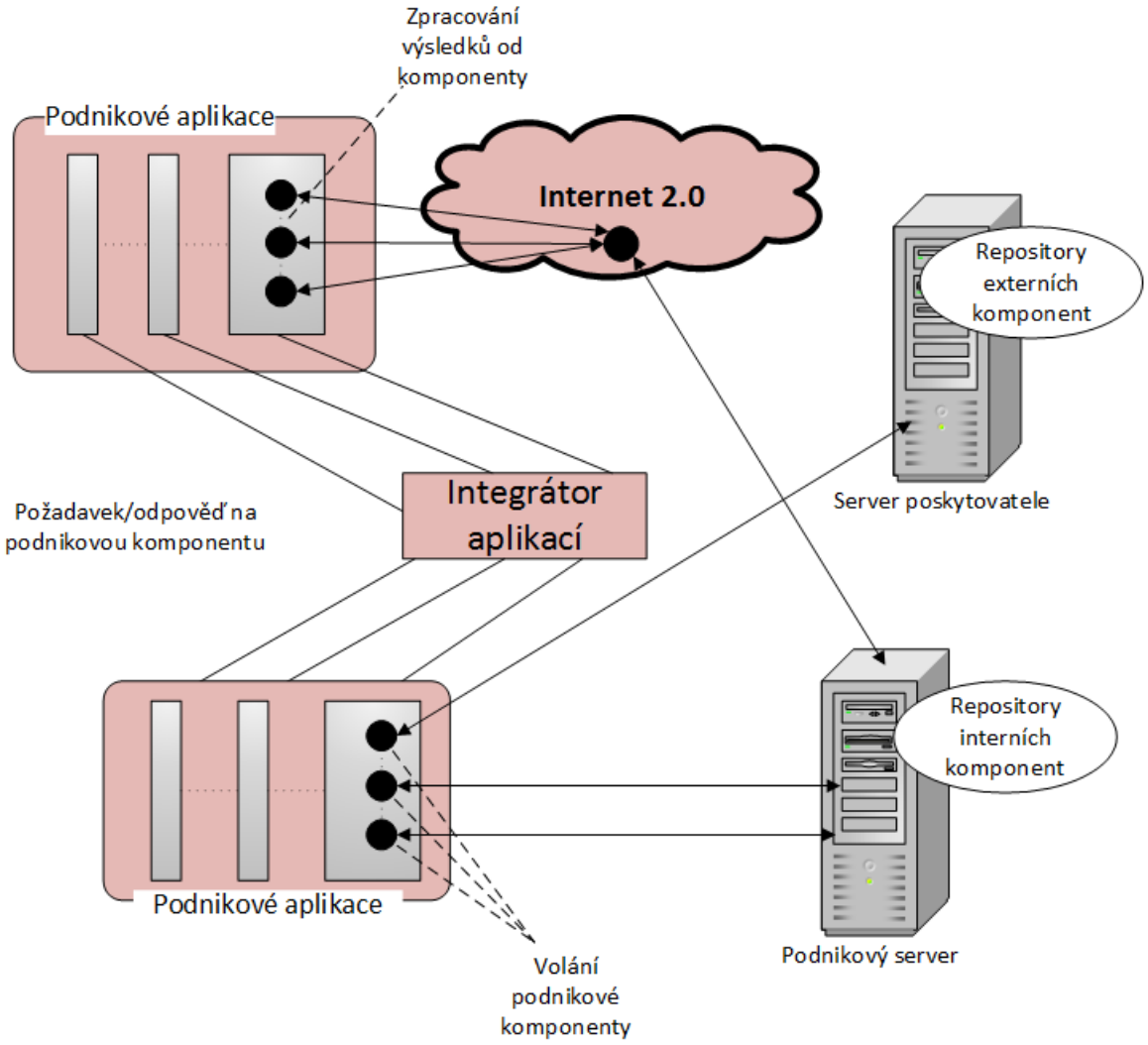
Komponenta – jednotka pro sestavování aplikací (assembling)

Je-li zákazníkovi poskytnuta komponenta ve formě instance, mohou ji programátoři podniku použít v tzv. sestavování aplikací (Assembling of applications). Tak může být postupně sestavena skupina aplikací pro komputerizaci několika podnikových podsystémů (viz obr. 3), např. Finančního podsystému z ERP, Personálního podsystému z HRM a Skladového podsystému z SCM. Komponenta se tak stává základním stavebním kamenem vzniklých aplikací. Pochopitelně, komponenta se chová v souladu s komponentovým modelem, podle kterého byla vytvořena a může mít vysokou komplexitu.

V případě, že je komponenta poskytnuta zákazníkovi ve formě instance, je možné, aby byla použita k sestavování aplikací (Assembling of applications). Takto je možné sestavit robustní informačí systém, který se skládá z několika podsystémů. Komponenta je pak základní stavební jednotkou tohoto software. Např. Personálního podsystému z HRM a Skladového podsystému z SCM.

Komponenta se tak stává základním stavebním kamenem vzniklých aplikací. Pochopitelně, komponenta se chová v souladu s komponentovým modelem, podle kterého byla vytvořena a může mít vysokou komplexitu.

Takto vytvořená aplikace nemá čistě dynamickou komponentovou architekturu, ale tzv. semi-dynamickou, protože komponenty jsou pospojované pomocí kódu vybraného programovacího jazyka.



Obrázek 3: Assembling komponent do podnikové aplikace

Takto sestavený software pro komputerizaci jisté domény rozhodně nepostrádá možnost flexibilních změn, protože veškerá funkcionalita je obsažena v komponentách, a ty je možno rychle modifikovat, nebo zaměňovat.

Pokud by bylo nutné zřizovat komunikaci mezi aplikacemi, je možné vytvořit tzv. integrátor a použít některou z metod komunikace mezi aplikacemi.

Zmíněná možnost využití komponent není vše, co je v rámci sestavování aplikací (assembling) možné. Na základě jednodušších komponent můžeme vytvářet silně kompozitní komponenty, což zvýrazní inženýrský přístup k tvorbě softwaru podniku.

Komponenta podléhá pravidlům objektového paradigmatu

V tomto přístupu se rozlišuje pojetí třídy COMPS, která je ve frameworku komponentové architektury. Tato třída je schopna generovat kolekci dále využitelných objektů. Jedná se o komponenty. Druhé pojetí chápe komponentu jako generované instance. Generovaná komponenta přebírá celou specifikaci třídy COMPS. Komponenta je zapouzdřena a přístup k atributům a operacím je možný jen přes tzv. rozhraní (interface) prostřednictvím zasílání zpráv. Struktura COMPS jako třídy je komplikovaná a většinou je ilustrována grafem. Obálka grafu je tvořena jen třídami, schopnými produkovat reálné komponenty. Třída COMPS může být také kompozitní (složená z dceřiných komponent), toto pak vyžaduje sestrojení Sekvenčního diagramu, který bude jasně definovat její chování (behaviour).

Jinak vypadá tento Sekvenční diagram pro komponentu, která není kompozitní a je složena pouze z tzv. dílčích návrhových tříd. Pro kompozitní třídu COMPS je nutné vytvořit Sekvenční diagram na úrovni jednotlivých podtříd komponent a potom můžeme kreslit detailní sekvenční diagramy každé z dceřiných komponent.

Třída COMPS může být charakterizována jen svým rozhraním (interface), tj. rovněž chování je vyjádřeno pomocí rozhraní. Problematika rozhraní tříd je uvedena v textu dále. K problematice se vrátíme při analýze tzv. specifikace komponenty. V následujícím textu bude použit jen termín komponenta, v souladu s výše nastavenou sémantikou. Bude se předpokládat existence třídy COMPS.

Komponenta a její komplexita

Zde je nutné rozlišit tzv. primitivní a kompozitní komponenty. Rozhodnutí, o který typ komponenty se bude jednat, je na managementu podniku a odvíjí se od rozhodnutí, ze kterých procesů se bude komponenta skládat. V tomto případě je také důležitá volba postupu informačního modelování. Je doporučeno vytvořit sekvenční diagram (a to i pro dceřiné komponenty).

Komponenta a její implementace

Nasazení komponenty v informační infrastruktuře zákazníka je svázáno s komponentou v implementační rovině. Do implementace komponenty patří zejména kód, který realizuje chování komponenty a umístění komponenty. Vývoj implementace je dán tzv. *Životním cyklem komponenty*, který je nejvýrazněji patrný až v komponentových modelech.

Komponenta a její opakovatelná využitelnost (reusing)

Hlavní výhodou komponenty je její znovupoužitelnost, která je docílena tím, že komponenta je samostatná jednotka. Kód komponenty je zapouzdřen a přístup k funkcionalitě komponenty zajišťuje rozhraní, které může být reprezentováno jako výčet metod.

Stav komponenty

Komponenta je chápána jako objektová třída, proto se její stav nikterak neprojevuje vně samotné komponenty. Tedy stav komponenty by neměl být z okolí komponenty nijak pozorovatelný.

Specifikace komponenty podrobněji

Specifikace komponenty pouze říká, co komponenta dělá a jak se chová, když jsou použity její služby. (Brown, 2000) Služby mohou být ve specifikaci podrobněji popsány a uvedeny operace, pomocí kterých se projevují. Rovněž mohou být uvedeny zvláštní stavy služeb a jejich omezení. Dále je často velmi podrobně popsáno interface, což je pro zákazníka velmi užitečné. Studium specifikace dané komponenty by mělo zákazníkovi dát veškeré informace o chování a vyvolání komponenty pomocí interface. V žádném případě se do specifikace nezařazuje skutečný kód, ve kterém je funkcionalita komponenty zapsána. Kód je součástí implementace komponenty a konzument k němu nemá přístup.

O specifikaci již bylo mnohé řečeno, z čehož je nutno specifikaci komponenty chápat jako spojení deskripce jejího chování (operace – sémantika) a rozhraní. Specifikace je tedy hlavním materiálem pro zákazníka, který chce komponentu využívat. Jazyk, ve kterém jsou deskripce chování a případně rozhraní komponenty popsány, často nemusí být běžným programovacím jazykem. Pochopitelně, specifikaci vytváří poskytovatel-tvůrce komponenty. Závěrem uvádím seznam součástí specifikace:

- Deskripce sady rozhraní komponenty
- Seznam operací a jejich notací, které uvádí popis parametrů pro vstup a výstup
- Neformální textový popis funkcionality, scénář využití komponenty
- Neformální popis operačního kontextu komponenty (hardware, operační systém, očekávané verze instalovaného software)
- Zpracování a dostupnost dat pro typické provedení komponenty

Často bývá součástí specifikace rovněž deskripce použití sady notací o propojení s dalšími komponentami, která poučuje uživatele, jak s komponentou manipulovat, dokonce to může být uvedeno kódově.

Na druhé straně je možnost, že rozhraní je specifikováno mimo komponentu samotnou. V každém případě musí být popis interface takový, aby bylo jasné, co je nutné pro provedení komponenty, jaká jsou vstupní a jaká výstupní data.

Implementace komponenty versus její specifikace

V CBD je jedna z dominantních zásad oddělení specifikace od implementace. Implementace komponenty je nezávislá a je složena z kódu komponenty, tj. z interpretace chování a z interpretace všech rozhraní komponenty. Použitý kód je mnohdy

odlišný od kódu aplikace, který se nachází na straně klienta. Význam specifikace a implementace realizuje, v souladu s pojetím komponenty jako třídy, zapouzdření komponenty.

Komponentový systém

Na základě pracovních postupů moderní objektové metodiky, orientované na vývoj cílového software ve formě komponent a vazeb mezi nimi (na základě tzv. rozhraní - interface) získáme tzv. komponentový systém. Vzniká tak tzv. komponentová architektura cílového software, založená na architektonické jednotce – komponentě.

2.8 Integrace a kompozice komponent

Vzhledem ke složitosti komponentového systému není integrace komponent zaručenou cestou k vytvoření optimálního komponentového systému. Dle (Crnkovic, Larson, 2002) je integrace komponent „mechanismus propojení komponent mezi sebou, takže jsou propojeny potřeby a služby jedné s potřebami a službami jiných komponent“. Integrace je založena na syntaktické informaci, tedy dle (Crnkovic, Larson, 2002) „Integrace samotná však není dostatečná k zajištění kvality mnoha detailů běhových interakcí v systému složeném z komponent“. Oproti tomu je kompozice zaměřena na chování komponent vzhledem k sytému, ve kterém se nacházejí.

2.9 Vývoj a deskripce komponentových systémů

Od roku 2000 začaly být komponenty všeobecně uznávanými architektonickými jednotkami pro robustní software. Stávající objektové metodiky (UP, RUP a další) byly rychle vybaveny pracovními postupy, které se problematikou komponent a komponentových systémů zabývají. Zejména je to viditelné v publikaci (Arlow, Neustad, 2007), která předkládá upravenou objektovou metodiku UP. Tento informační vývoj komponentových systémů je zaznamenán v jazyku UML (Unified Modeling Language).

2.10 Přehled známých operací nad komponentami a komponentovými systémy

Jsou zde uvedeny především operace, které se dají využít v rámci informačního modelování komponent a komponentových systémů, resp. po ukončení modelování. Patří sem následující operace:

- Zavedení komponenty
- Zavedení nebo odvolání portu komponenty
- Naplnění hodnoty portu p_{in}/p_{out} na základě události „naplnění“

- Změna hodnoty portu na základě události „naplnění“
- Odstranění hodnoty daného portu
- Odstranění rekurzivity portů u téže komponenty
- Propojení dvou komponent C_1, C_2 na základě hodnot v jejich portech
- Verifikace propojení komponent (konfigurace)
- Vyřazení komponenty z komponentového systému
- Přidání komponenty do komponentového systému
- Nalezení funkcionálních řetězců komponent v komponentovém systému
- Nalezení vytížení komponenty

Seznam uvedených operací jistě není konečný, ale je neustále obohacován o další, které vzniknou na základě nových pohledů na komponenty a komponentové systémy. V disertační práci budou analyzovány ještě dva pohledy:

1. Komponentové systémy z pohledu Teorie systémů (viz kap. 7)
2. Komponentové systémy z pohledu Teorie vyčíslitelnosti (viz kap. 8)

3 Formalismy využitelné k deskripci komponentových systémů

Teoretický přístup ke komponentovým systémům je většinou rozčleněn na směr využití formálních jazyků a směr využívající teorie vyčíslitelnosti. Rapidní nástup obou směrů začal v roce 2001 a pokračuje dodnes. Snahou obou směrů je verifikovat závažné vlastnosti komponentových systémů. Každý z uvedených směrů má své výhody a nevýhody. Např. ve směru využití formálních jazyků dominuje syntax a deskripce hierarchičnosti, kdežto směr uplatnění teorie vyčíslitelnosti, přesněji konečných automatů, se snaží zavést rovnováhu v dominanci syntaxe a sémantiky komponentových systémů.

3.1 Využití formálních jazyků

Komponentové systémy jsou vývojovým produktem použití objektové metodiky (s dominancí komponentové architektury) k informačnímu modelování problémových domén. Jinými slovy řečeno, výsledný software je členěn na samostatné, navzájem spolupracující komponenty, což je právě zobrazeno v komponentovém systému.

V některých publikacích jsou návrhy vlastních formálních jazyků, v jiných jsou navrhovány jen rozšíření dvou mezinárodně zavedených jazyků UML (Unified Modeling Language) a OCL (Object Constraint Language) viz (OCL, 2014). Těmto dvěma jazykům se budu věnovat podrobněji.

3.2 Jazyk UML

Jazyk UML vytvořili Grady Booch, James Rumbaugh a Ivar Jacobson jako závěr slučování jejich grafických způsobů deskripce výsledků pracovních postupů svých jednotlivých objektových metodik.

UML není a ani nenabízí nějaký typ objektové metodiky, je pouhým grafickým jazykem využitelným v deskripci výsledků analytických postupů různých objektových metodik. UML není vázán na jednu specifickou metodiku, je společnou vizuální platformou různě se lišících objektových metodik. Např. rozdíly mezi metodikami Select Perspective, RUP (Rational Unified Process/UP) a OPEN (Object-oriented Process, Environment and Notation) jsou v jejich uplatnění a pracovních postupech a ne v implementovaném UML. Na druhé straně není v metodikách nikterak zakázáno přidávat k platformě UML další specifické diagramy, které zvýší transparentnost analytických prací a jejich grafických notací.

Další vývoj jazyka může být založen na metodice MDA (Model Driven Architecture), která chápe tvorbu-generování software na základě několika modelů: CIM (Computer-independent Model), PIM (Platform-independent Model) a PSM (Platform-specific Model). Teprve z modelu PSM má být pomocí vhodného generátoru získán výsledný kód.

Unifikace UML je uznávána na základě několika jeho vlastností:

1. Slouží jako vizuální deskripce pro všechny fáze vývojového cyklu software.
2. Má schopnosti k notacím výsledků modelování různých problémových domén (podniky, školy, armáda, systémy řízené počítačem, …), lišících se charakterem procesů.
3. Není závislý na žádném programovacím jazyku. Výsledky zapsané v UML se dobře programují jak v „čistých“ objektových jazycích (Smalltalk, Java, C#, Ruby, …), tak rovněž v tzv. hybridních objektových jazycích (C++, Visual Basic, …) a tzv. jazycích založených na objektech. Mnozí programátoři programují modely UML rovněž v neobjektovém jazyku C.

UML plně respektuje práci s objekty tak, jak je to v objektovém programování (viz Collaboration diagrams - diagramy spolupráce - komunikace objektů). Jedná se o programování spolupráce objektů na základě zasílaných zpráv.

UML dokáže modelovat statiku problémové domény (třídy a vztahy mezi nimi, stavy tříd = stavy jejich objektů).

Vedle toho je UML plně dostatečný rovněž pro modelování dynamiky problémové domény, tedy chování objektů jednotlivých tříd a spolupráci mezi nimi.

Struktura jazyka UML

Standard UML ve verzi 2.0 se skládá ze čtyř částí:

1. **SuperStructure** – popis UML z hlediska uživatele (analytik/programátor). Tato část popisuje jednotlivé diagramy.
2. **Infrastructure** – metamodel stojící v pozadí za UML, specifikovaný pomocí Meta-Object Facility (MOF).
3. **Object Constraint Language (OCL)** – jazyk pro specifikaci vstupních a výstupních podmínek, invariantů v jednotlivých diagramech.
4. **Diagram Interchange** – popis XML struktur pro výměnu konkrétních modelů mezi jednotlivými modelovacími nástroji.

Struktura UML je postavena na následujících prvcích:

- **Stavební bloky** - jsou základní prvky modelu (třídy), relace a diagramy.
- **Společné mechanismy** - v UML jimi dosahujeme specifických cílů.
- **Architektura** – pohled v UML na architekturu navrhovaného software.

Skutečností je to, že sám UML je navržen v UML a tento návrh je tzv. metamodelem UML. Většinou si budeme všímat zejména stavebních bloků. Obecně má UML jen tři stavební bloky:

- **Předměty** (things).

- **Vztahy** (relationships).
- **Diagramy** (diagrams).

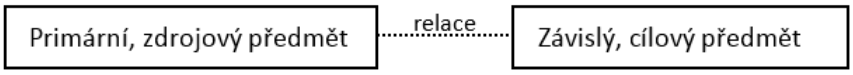
Předměty

Mezi předměty jazyka UML řadíme:

- **Strukturální abstrakce** (structural things), což jsou podstatná jména jako třídy, rozhraní, spolupráce, případ užití, aktivní třída, abstraktní třída, komponenta, uzel, …
- **Chování** (behavioural things) např. chování, stav, …
- **Seskupení** (grouping things) balíčky pro seskupení významově souvisejících prvků modelů do soudržných jednotek.
- **Poznámky** (annotation things) anotace přidávané k modelu.

Relace

Relace umožňují zachytit významový (sémantický) vztah mezi předměty a je možné je chápat na množinovém základě. Často se používá jednoduché pojmenování členů určité relace (viz obr. 4). Relace se chápe od primárního k závislému předmětu.



Obrázek 4: Ukázka relace

Následující tabulka (viz obr. 5) uvádí typy relací a syntax pro jejich notaci v UML.

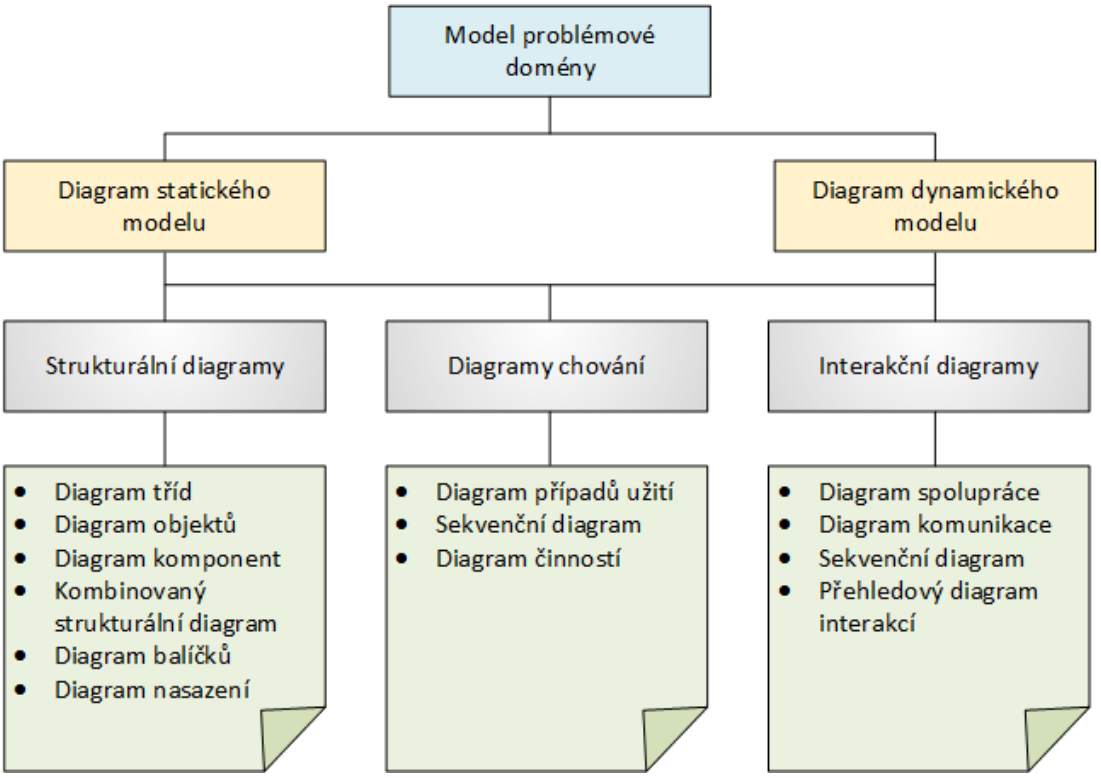
Relace	Syntax v UML	Deskripce relace
Závislost (dependency)		Změna v jistém primárním předmětu ovlivňuje význam závislého předmětu.
Datová asociace (association)		Popis datového spojení mezi třídami a pochopitelně, potom mezi jejich objekty.
Agregace (aggregation)		Cílový závislý předmět je součástí zdrojového primárního prvku.
Kompozice (composition)		Silnější forma agregace, která má více omezení než agregace běžná.
Ochranná nádoba (containment)		Zdrojový primární předmět obsahuje cílový závislý předmět.
Dědičnost – zobecnění generalizace (generalization)		Jeden předmět je specializací jiného předmětu a lze jej nahradit obecnějším předmětem (univerzálnějším).
Realizace (realization)		Asociace mezi klasifikátory, kde jeden klasifikátor určuje dohodu, jejíž uskutečnění zaručuje druhý klasifikátor.

Obrázek 5: Typy relací a syntax pro jejich notaci v UML

Diagramy

UML diagramy, tak jak jsou předepsané, nejsou modely. Považujeme je za jisté pohledy na modely. Nakreslené předměty a relace můžeme z diagramu odstranit, ale v repositáři podpůrného CASE stále zůstávají. Z modelu je musíme zvlášť vymazat. Co je ovšem důležité, je to, že CASE nástroj provádí neustále kontrolu požadovaných konzistencí v rámci všech modelů, které má ve svém repozitáři uložené.

Diagramy je možné chápat ze dvou pohledů, a to, zda modelují statickou nebo dynamickou stránku problémových domén. Statický model problémové domény zachycuje především předměty, a strukturální asociace mezi předměty. Na rozdíl od toho, dynamický model zachycuje způsob, kterým předměty na sebe navzájem působí tak, aby bylo namodelováno chování problémové domény a potom rovněž tomu odpovídající chování výsledného softwarového systému. Zařazení jednotlivých diagramů ukazuje následující obrázek (viz obr. 6).



Obrázek 6: Složení modelu problémové domény z diagramů UML

Nedostatky vytýkané UML

V publikaci (Merunka, 2006) je vytýkáno UML následující:

- Totožný obdélník pro objektovou třídu a její instanci.

- UML pracuje s typovanými datovými modely, které čistě dynamickým objektovým jazykům nevyhovují.
- Nedostatečná podpora procesního modelování.
- Více variant pro zobrazení některých detailů v modelech (stavové diagramy – mixáž automatů typu Mealyho a Moora).
- UML obsahuje příliš mnoho termínů a na různých stupních abstrakce (např. vazby mezi případy užití).

3.3 Jazyk OCL

Jazyk OCL (Object Constraint Language) definuje podmínky, které musí být dodrženy na modelovaném systému. Tento jazyk tvoří, stejně jako UML, spojku mezi formálním způsobem popisu systému a verbálním jazykem. OCL je silně typový jazyk. (OCL, 2014) Každý výraz v OCL má definovaný typ. Syntaxe OCL dodržuje následující tvar:

context <jméno> [inv|pre|post]: <výraz>

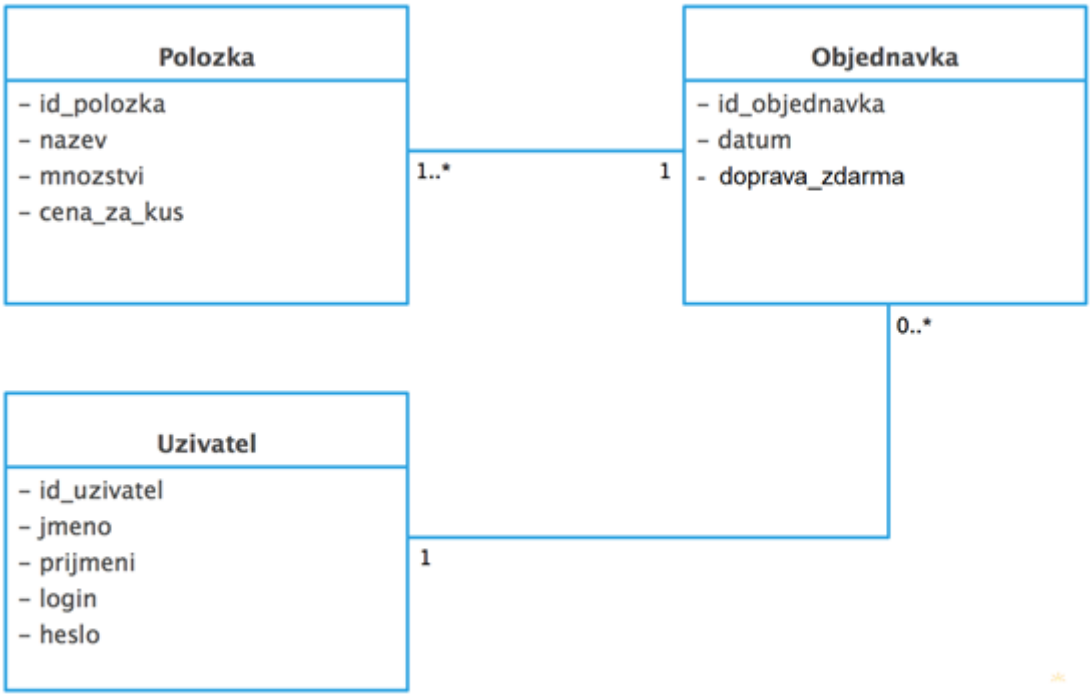
OCL je možno použít pro doplnění diagramů UML. Blok package určuje problémovou doménu nebo její část, do které daná problematika popisovaná v jazyce OCL spadá. Klíčové slovo context označuje třídu, u které se bude definovat omezení. (OCL, 2014) Tento jazyk umožňuje pracovat s osmi typy výrazů, a to:

- Inv – značí podmínku, která musí být vždy splněna pro všechny instance třídy, typy nebo interface. Invarianta musí být platná po celou dobu.
- Pre a post – vstupní nebo výstupní podmínka, která musí být splněna před nebo po akci.
- Body – specifikuje tělo operace nebo proměnné.
- Init – nastavení výchozí hodnoty atributů.
- Def – slouží k nastavení proměnné na určitou hodnotu.
- Let – definice lokální proměnné.
- Derive – slouží k odvození nových elementů v modelu.
- Priorit – určuje prioritu znaků.

V jazyce OCL se dělí datové typy na primitivní (Boolean, Integer, Real, String), strukturované (Tuple) a vestavěné (OclAny, OclType, OclState, OclVoid, OclMessage). (OCL, 2014) Nad těmito typy lze provádět operace. Jedná se o typické operace známé z programovacích jazyků (např. porovnání). Některé operace zajišťují vestavěné funkce (např. absolutní hodnota – abs(), spojování řetězců – concat()).

Příklady integritních omezení v OCL:

Následující obrázek (viz obr. 7) ilustruje zjednodušený diagram tříd aplikace na objednání zboží:



Obrázek 7: Diagram tříd pro systém Objednání zboží

Na základě výše uvedeného diagramu tříd mohou být analytikem doplněna omezení, která nemohou být diagramem tříd postihnutelná. Následuje několik příkladů obecných tvrzení.

„Uživatelské heslo musí být větší než pět znaků.“
context Uzivatel inv: self.heslo->size() >= 5

„Celkový počet kusů u jedné objednávky nemá být větší než sto.“
context Objednavka inv: self.polozky->mnozstvi->sum() <= 100

„Datum u nové objednávky nesmí být starší než aktuální datum.“
context Objednavka inv: self.datum >= self.aktualni_datum()

V rámci jazyka OCL je také možné popisovat i metody tříd. Definujme u třídy Uzivatel metodu osolHeslo. Tato metoda bude zabezpečovat hash hesla proti slovníkovému útoku. Metoda bude přebírat heslo a uživatelské jméno jako parametr. Vstupní heslo musí být delší než pět znaků.

context Uzivatel::osolHeslo(heslo:String, uzivatel:String):String pre: heslo.size() > 6 post: result=hash(heslo+uzivatel)

Dále následuje ještě několik příkladů na metody tříd:

„Na dopravu zdarma má nárok jen ten, jehož objednávka přesáhne 2000 Kč.“
context Objednavka::vyhodnotDopravuZdarma():boolean pre: self.polozky->cena_za_kus*mnozstvi->sum() > 2000 def: self.doprava_zdarma=true

„Pokud je u objednávky více než sto položek, je nárok na 10% slevu.“
context Objednavka::vyhodnotSlevu():bool pre: self.polozky->sum() > 100 post: result=true

3.4 Využití teorie vyčíslitelnosti

Společnou myšlenkou směru použití teorie vyčíslitelnosti je uplatnit konečné automaty nejen k deskripci komponent, ale rovněž k realizaci operací nad nimi, resp. k verifikaci jejich požadovaných vlastností. Významné použití této myšlenky je zejména v publikacích (Alfaro, Henzinger, 2001),(Alfaro, Henzinger, 2004), (Brim, Černá, Vařeková, Zimmerová, 2006) a (Isazedech, Karimpour, 2008). Např. ve čtvrté a páté publikaci se začíná uplatněním formalismu na samotné komponenty, potom se pro ně definují účelové konečné interface automaty, které většinou prezentují vlastnosti interakce komponent v komponentovém systému. Všechny uvedené publikace mají odlišné formalizmy základních pojmů. Rozhodl jsem se při rekapitulaci a hodnocení výsledků původní formalizmy zachovávat.

V prvotní publikaci (Alfaro, Henzinger, 2001) jsou reprezentační automaty nazývány interface automaty (interface automata), protože se problematika dotýká nejvíce rozhraní (interface) komponent a jejich sladění. V publikaci (Isazedech, Karimpour, 2008) autoři navržené pojmenování automatů zachovávají. Do češtiny budu termín „interface automata“ překládat jako „interface automaty“, ačkoliv termín interface není český.

Publikace (Alfaro, Henzinger, 2001):

Pojetí interface automatu je použito pro každou komponentu komponentového sytému, ale předpokládá se, že budou existovat komponenty, které nemohou spolupracovat se žádnou komponentou v komponentovém systému. Interface automaty a zavedené vlastnosti pro ně (kompatibilita, kompozice, …) by měly řešit všechny situace týkající se interaktivity komponent.

Interface automat pro komponentu P je definován následovně:

$\mathbf{P} = (\mathbf{V_P}, \mathbf{V_P^{init}}, \mathbf{A_P^I}, \mathbf{A_p^0}, \mathbf{A_P^H}, \mathbf{T_P})$, kde

V	$\mathbf{V_P}$ je množina stavů, $\mathbf{V_P^{init}}$ je množina počátečních stavů
A	$\mathbf{A_P^I}, \mathbf{A_P^O}, \mathbf{A_P^H}$ jsou navzájem disjunktní množiny input, output akcí a skrytých akcí, jejich sjednocení je množina A_P všech akcí
T	$\mathbf{T_P} \subseteq \mathbf{V_P} \times \mathbf{A_P} \times \mathbf{V_P}$

Autoři navrhují pro interface automat grafické zobrazení - schránku (box), kde porty korespondují s input a output akcemi. Pro rozlišení charakteru akcí se používají znaky „?“ – akce input, znak „!“ – akce output a znak „ ; “ – akce interní. Šipky jsou použity pro stavy.

Dále se definují některé drobné vlastnosti jednotlivých množin a **uzavřenost/otevřenost** automatu. První z potřebných operací je definováno sestavení - **složení** (composition) dvou interface automatů P, Q a potom jejich **kompatibilita** (compatibility).

Dva automaty se považují za sestavitelné (composable, Definition 3) jen v tom případě, že jejich akce jsou disjunktní (disjoin) a input akce jednoho jsou koincidenční (coincident - shodné) s output akcemi druhého, tedy:

$$\mathbf{A_P^I} \cap \mathbf{A_Q^I} = \emptyset$$
$$\mathbf{A_P^O} \cap \mathbf{A_Q^O} = \emptyset$$
$$\mathbf{A_P^H} \cap \mathbf{A_Q} = \emptyset$$
$$\mathbf{A_Q^H} \cap \mathbf{A_P} = \emptyset$$

Jestliže jsou dva automaty P, Q sestavitelné (Definition 4), potom $(\mathbf{P}, \mathbf{Q}) = (\mathbf{A_P^I} \cap \mathbf{A_Q^O}) \cup (\mathbf{A_P^O} \cap \mathbf{A_Q^I}) \neq \emptyset$. *Produkt, který vznikne sestavením dvou interface automatů P, Q, je označen notací* $P \otimes Q$.

Autoři dále deklarují, jaký automat $P \otimes Q$ vznikne po operaci sestavení. Především, že je to opět interakční automat definovaný takto (Definition 4):

$$V_{P\otimes Q} = V_P \times V_Q$$
$$V_{P\otimes Q}^{init} = V_P^{init} \times V_Q^{init}$$
$$A_{P\otimes Q}^I = A_P^I \cup A_Q^I / \text{sdílející (P, Q)}$$
$$A_P^O \otimes Q = A_P^O \cup A_Q^O / \text{sdílející (P, Q)}$$
$$A_{P\otimes Q}^H = A_P^H \cup A_Q^H \cup \text{sdílející (P, Q)}$$

Sestavení výsledných položek automatu $P \otimes Q$ je dáno kartézskými součiny a sjednocením položek obou automatů. Vznikají tak uspořádané dvojice stavů a akcí. Množina $T_{P\otimes Q}$ je snadno vyvoditelná. Sestavení dvou interface automatů, které vede rovněž na interface automat, je předobrazem sestavení dvou kom-

ponent do nového celku na základě jejich interface. Je to dáno podmínkou, že $(\mathbf{A_P^I} \cap \mathbf{A_Q^O}) \cup (\mathbf{A_P^O} \cap \mathbf{A_Q^I}) \neq \emptyset$.

Autoři uvádějí další významné vlastnosti individuálních nebo dvou sestavitelných interface automatů, které mají interpretaci v komponentovém systému. Jde o:

- nepovolené stavy (Definition 5),
- okolí interface automatu (Definition 6),
- povolené (legal) okolí interface automatu (Definition 7),
- kompatibilita dvou interface automatů (Definition 8),
- kompatibilita na základě stavů (Definition 9),
- kompozice dvou sestavitelných interface automatů (Definition 10).

Významné jsou zejména vlastnosti **povoleného okolí, kompatibilita, kompatibilita na základě stavů** a **kompozice dvou sestavitelných interface automatů**.

Nechť jsou dány dva sestavitelné interface automaty P, Q (Definition7). Povolené okolí dvojice (P, Q) je okolí pro $\mathbf{P} \otimes \mathbf{Q}$ *takové, že žádný stav v* $\mathbf{Illegal(P, Q)xV_E}$ *není dosažitelný v* $(\mathbf{P}, \mathbf{Q}) \otimes \mathbf{E}$.

Na základě Definice 7 je potom zaveden pojem kompatibility dvou sestavitelných interface automatů.

Dva interface automaty P, Q jsou kompatibilní (Definition 8), jestliže jsou ***neprázdné***, ***sestavitelné*** *a existuje legální okolí pro (P, Q).*

Uvažujme dva sestavitelné interface automaty P, Q. Jejich kompozice (composition, Definition 10) $(\mathbf{P}||\mathbf{Q})$ je interface automatem se stejnou množinou stavů jako $(P \otimes Q)$. Stavy kompozice jsou:
$V_{P||Q} = Cmp(P, Q)$, počáteční stavy jsou: $V^{init}_{(P||Q)} = V^{init}_{(P \otimes Q)} \cap Cmp(P, Q)$
a kroky jsou: $T_{P||Q} = T_{P \otimes Q} \cap (Cmp(P, Q) \times A_{P||Q} \times Cmp(P, Q))$.

Vedle definice uvádějí autoři na konci příspěvku dvě velmi zajímavá tvrzení, která nedokazují.

Tvrzení 1: Dva interface automaty P, Q jsou kompatibilní, jestliže:

1. jsou sestavitelné a
2. jejich kompozice je neprázdná.

Tvrzení 2: Pro všechny interface automaty P, Q a R platí:

1. buď $(P||Q||R)$ a $(P||R||Q)$ jsou nedefinovatelné, protože některé z interface automatů nejsou sestavitelné, anebo
2. $(P||Q||R) = (P||R||Q)$.

První tvrzení naznačuje, že kompatibilitu dvou interface automatů P, Q zaručuje jejich sestavitelnost a jejich neprázdná kompozice. Druhé tvrzení diskutuje situaci v kompozici více interface automatů.

Významné myšlenky publikace (Alfaro, Henzing, 2001):

1. Interface automat komponenty C je definován přímo, bez ohledu na pojetí komponenty.
2. Základem interface automatu jsou množiny input, output a internal akcí.
3. V interface automatu není předepsáno, jakými hodnotami se naplňují porty, porty nejsou vůbec uvažovány.
4. Význačnými operacemi jsou sestavitelnost (composition) dvou interface automatů a kompatibilita dvou interface automatů.
5. Na komponentový systém v podobě interface automatů se nevztahuje systémový pohled a řešení základních systémových úloh.

Publikace (Isazedech, Karimpour, 2008):

Autoři publikace (Isazedech, Karimpour, 2008) Ayaz Isazadech a Jaber Karimpour z University of Tabriz (Iran) používají nejdříve velmi složitý formalismus pro deskripci diskrétní událostní komponenty v nehierarchickém komponentovém systému. Popis komponenty je natolik široký, že přebírá i některé vlastnosti konečných automatů.

Komponenta **C** je definována jako šestice $\mathbf{C} = (\mathbf{P}, \mu, \mathbf{s^0}, \mathbf{S}, \Sigma, \mathbf{T})$, kde

P	je konečná množina portů složená z input portů P^I a output portů P^0 .
μ	je přiřazení hodnoty portům z univerza V, tedy $\mu : P \leftarrow 2^V$.
S	je množina stavů a $\mathbf{s^0}$ je počáteční stav.
$\Sigma =$	$\Sigma^I \cup \Sigma^O \cup \Sigma^H$ je množina událostí, Σ^I je množina input událostí, Σ^O je množina output událostí a Σ^H je množina skrytých událostí. Události Σ^I značí naplnění input portů, kdežto události Σ^O značí naplnění output portů.
T	je množina kroků, $T \subseteq S \times \Sigma \times S$ (kartézské součiny). Kroky se provedou na základě stávajícího stavu a události.

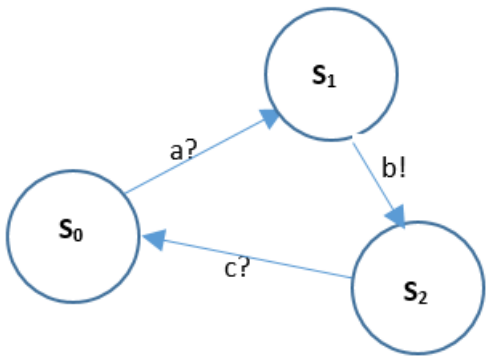
Dále autoři zavádějí interface automat IA pro každou komponentu. Tento automat reprezentuje deskripci interakce komponenty na vlastnostech konečného automatu. Automat je překvapivě chudý na složky, protože některé jeho činnosti (např. přijetí události a naplnění portu a vybrání dalšího stavu) byly přisouzeny diskrétní událostní komponentě.

Interface automat IA je definován jako uspořádaná čtveřice $IA = (s^0, S, \Sigma, T)$, kde jednotlivé složky odpovídají stejně pojmenovaným složkám diskrétní událostní komponenty.

Všechny interface automaty komponentového systému určují tzv. I-protokol a zaznamenávají průběh interakcí v komponentovém systému. Důležité je, že interface automat přesně zaznamenává události své komponenty a přidělování hodnot jednotlivým portům. Autoři publikace úspěšně vyřešili několik problémů, např. relace mezi jednotlivými interface automaty a přechod na diskrétní hierarchickou událostní komponentu.

Autoři proto zavedli relaci priority $< \Sigma$ přímo do popisu komponenty a tím umožnili zpracovávat hierarchické komponentové systémy a pomocí nich vysvětlili, kdy se jedná o vyšší nebo nižší prioritu mezi dvěma komponentami. Snadným způsobem je autory nad interface automaty zavedena relace souvislosti portů a porovnání jejich interakcí.

Názorná je rovněž grafická ilustrace (viz obr. 8) interakčního diagramu zavedené komponenty C. Např. komponenta má množinu input portů $P^I = \{a, b\}$ a množinu output portů $P^O = \{c\}$. Potom je grafická ilustrace jejího interakčního automatu ve tvaru:



Obrázek 8: Grafická ilustrace interakčního automatu komponenty C

Vzhledem k možnosti značného množství komponent, a tedy rovněž interface automatů, pokusili se autoři zavést pojmy uzel (node) a regulátor (controller), které by situaci zjednodušily. Uzel může fungovat množinově, tj. jako jeden nebo více uzlů a může být ve spojení se svými, řízenými uzly. Pomocí uzlů bylo hned upraveno pojetí jejich hierarchičnosti. Vedle toho bylo na základě událostí upraveno pojetí

kroků (transitions) a verifikace základních vlastností komponentového systému byla přenesena na analýzu vlastností uzlů. Takovými vlastnostmi jsou zejména události, konfigurace, počáteční konfigurace uzlu a kroky (transactions) automatů. Vedle toho jsou přesně definovány input a output kroky interface automatů. V závěru publikace se autoři orientují na pojetí verifikace komponentového systému prostřednictvím základních vlastností uzlů. Verifikují se kroky, dosažitelnost stavů, naplnění stavů a vlastnosti propojení automatů.

Významné myšlenky publikace (Isazedech, Karimpour, 2008):

1. V diskutované publikaci autoři nahradili model komponentového systému pro diskrétní a událostní komponenty matematickým modelem interface automatů, které registrují život komponent.
2. Do systému interface automatů byla zavedena relace hierarchičnosti, což umožnilo analyzovat hierarchické komponentové systémy.
3. Do systému interface automatů byly zavedeny pojmy uzel a regulátor a všechny vlastnosti interakčního modelu se analyzovaly na jeho rozšíření.
4. Do oblasti verifikovaných vlastností byly zařazeny události, konfigurace, input a output kroky, dosažitelnost, naplnění stavů a vlastnosti propojení automatů.
5. Na komponentový systém se ale nehledí jako na systém různých systémových vlastností komponent, bez převodu komponentového systému do kompozice interface automatů.
6. Formalizmy v systému automatů jsou obtížně interpretovatelné do komponentového systému zobrazeného v jazyku UML.

4 Co bych chtěl v disertační práci rozvinout

V literatuře, kterou jsem použil za základ své rešerše, jsem se především setkal s následujícími přístupy ke komponentám a komponentovým systémům:

1. s verbálními definicemi základních pojmů v oblasti komponent a komponentových systémů (komponenta, rozhraní, komponentový systém, konfigurace, rekonfigurace, orchestrace, propojení komponent, syntax a sémantika, klonování komponent, primitivnost a složitost komponent, …),
2. s formalizací pojetí a s vlastnostmi dynamických komponentových systémů,
3. s formální definicí událostní komponenty, s transformací komponenty do konečného interface automatu (interface automaton) a reprezentací všech vlastností komponenty tímto automatem,
4. s transformací komponentových systémů do systémů interface automatů a s reprezentací vlastností komponentových systémů vlastnostmi těchto automatů.

Přístup 1. se vyskytuje v základní knižní literatuře o komponentách a komponentových systémech. Přístup 2. se vyskytuje v pojetí moderních objektových metodik, jejichž výsledkem jsou komponentové systémy jako finální verze informačního modelování komputerizace problémových domén (viz metodiky UP, RUP a další).

Na druhé straně přístupy 3. a 4. se vyskytují výlučně v odborných příspěvcích na vědeckých konferencích. Jestliže text publikací v přístupech 1. a 2. byl velmi transparentní, tak přístupy 3. a 4. bylo někdy obtížné analyzovat a převádět četné výsledky do problematiky komponent a komponentových systémů.

Společným rysem všech uvedených přístupů bylo opomíjení pohledu na komponentové systémy z hlediska obecné teorie systémů s implementací úloh, které se v této teorii na konkrétní systémy implementují.

Všechny uvedené přístupy se liší ve svých cílech, formalismech a získaných výsledcích. Podstatu, tj. terminologii a vývoj komponent a komponentových systémů objasňují zejména přístupy 1. a 2., ačkoliv převážně verbálně. Pohledy 3., 4. a 5 aplikují na komponentové systémy poznatky z Teorie vyčíslitelnosti a Teorie systémů.

Ve své disertační práci bych chtěl v souladu s cílem disertační práce aplikovat především pohled Teorie systémů a implementovat vybrané systémové úlohy na komponentové systémy. Chtěl bych začít formální definicí komponenty a komponentového systému, poté pokračovat na implementaci vybraných systémových úloh. Tím bych chtěl ukázat, že ačkoliv tento přístup není v odborné literatuře běžný, přesto přináší v implementaci systémových úloh pozoruhodné výsledky.

Dále bych se chtěl zabývat pohledem na reprezentaci komponenty interface automatem (interface automaton) a využitím interface automatů k reprezentaci komponentových systémů popsaných na bázi jazyka UML. Současně bych chtěl vyvinout pro práci s interface automaty podpůrný Framework, zaměřený na verifikační oblast vlastností komponentových systémů.

Tento pohled je odlišný od pohledů v publikacích (Alfaro, Henzinger, 2001), (Alfaro, Henzinger, 2004), (Brim, Černá, Vařeková, Zimmerová, 2006) a (Isazedech, Karimpour, 2008). V tomto pohledu chci ukázat, že interface automaty jsou hodnotným nástrojem pro informační modelování komponentových systémů.

5 Cíl disertační práce

Vzhledem k závěrům charakteristiky současného odborného stavu problematiky komponentových systémů jsem si pro disertační práci stanovil následující cíl a kroky k jeho naplnění.

Cílem této práce je přispět k vývoji v oblasti formální analýzy komponentových systémů ve dvou směrech, a to z pohledu teorie systému a teorie vyčíslitelnosti.

Kroky vedoucí ke splnění cíle jsou následující:

1. Stanovit základní vlastnosti komponentových systémů a specifikaci kladů a záporů těchto vlastností. Chápat komponentové systémy jako specifické systémy prvků, vazeb mezi nimi a operacemi nad prvky i úlohy nad celými systémy. Uplatnit prostřednictvím množinové platformy zákony teorie systémů a formálními definicemi upřesnit používané pojmy (komponentový systém, rozhraní, propojení, orchestrace, konformita, konfigurace, rekurzivita komponent a funkcionální řetězce). Implementovat specifické systémové úlohy pro komponentové systémy.
2. Reprezentovat vlastnosti komponenty specifickým interface automatem (interface automaton) a převést komponentový diagram na systém tzv. interface automatů. Realizovat operace nad interface automaty a převést je zpětně do komponentového systému. Vytvořit v abstraktní rovině formální aparát pro deskripci komponent (tj. komponentovou algebru a její operace nad komponentami), deskripci komponentových systémů, požadovaných a nabízených interface, provádění propojení a provádění četných statických úloh nad komponentovými systémy. Implementovat specifické systémové úlohy pro komponentové systémy.
3. Reprezentovat vlastnosti komponenty specifickým interface automatem a převést komponentový diagram na systém tzv. interface automatů. Realizovat operace nad interface automaty a převést je zpětně do komponentového systému.
4. Navrhnout způsob uplatnění vybraného formálního přístupu z oblasti komponentových systémů v oblasti Cyber-physical systems (System of Systems), v nichž komponenta představuje úzké propojení mezi software a hardware. Tyto komponenty po propojení mezi sebou tvoří tzv. Smart system, který na základě propojených komponent poskytuje nově vzniklou funkcionalitu.
5. Vytvořit programovou realizaci – tzv. framework pro oboustranný převod mezi komponentovým systémem a systémem interface automatů a rovněž pro vybranou množinu úloh nad systémem interface automatů. Jako demonstrační doménu použiji Správu lidských zdrojů (HRM - Human Resource Management).

6 Metodiky pro naplnění cíle disertační práce

Základem použitých metod řešení problematiky disertační práce jsou systémová analýza a systémová syntéza. Systémová analýza bude využita v pohledu na možné deskripce komponentových systémů pomocí verbálních a formálních prostředků.

V disertační práci metodicky využiji specifické pohledy na komponentové systémy, a to pohled z teorie systémů a teorie vyčíslitelnosti. V těchto dvou pohledech využiji především konečné automaty a systémovou algebru postavenou na teorii množin.

Dále využiji metodiku tvorby a využití tzv. frameworku systémů, tj. softwarové podpory obecných systémů. Pomocí této metodiky vytvořím specifický framework pro podporu vztahu komponentových systémů v jazyku UML a systémů interface automatů.

7 Komponentové systémy a teorie systémů

Začátek této kapitoly zaměřím na možnost formalizace základních pojmů z problematiky komponent a komponentových systémů, zbytek kapitoly věnuji vlivu Teorie systémů (operace nad komponentovými systémy) na tak specifické systémy, jimiž komponentové systémy jsou.

K formalizaci základních pojmů z problematiky komponent a komponentových systémů je možno zvolit více přístupů, např. nástroje z Teorie množin (kartézský součin, množinové operace), nástroje z Teorie vyčíslitelnosti (specifické interface automaty). Vzhledem k tomu, že interface automaty budou podrobně analyzovány v roli prezentace komponent a komponentových systémů v 8. kapitole „Využití interface automatů v modelování komponentových systémů“ (kde bude zřejmé, že základní pojmy z problematiky komponent a komponentových systémů jsou jimi snadno prezentovatelné), zvolil jsem přístup využití nástrojů Teorie množin.

Při formalizaci pojmů z problematiky komponent a komponentových systémů se vychází z jejich uznávaných verbálních definic uvedených v kapitole 2 „Pojetí komponent a komponentových systémů“.

Následující definice jsou pro teoretický přístup k uvedeným pojmům užitečné již v tom, že jsou vedeny snahou uplatnit podstatu obecných systémů. Na druhé straně, formalizované pojmy jsou přesnější než jejich verbální definice a zavádí se tak jednotný výklad jejich pojetí. Jsou uvedeny formální definice k těmto pojmům:

- komponenta
- propojení komponent
- typová konformita
- komponentový systém
- konfigurace komponentového systému
- primitivní rekurzivita
- orchestrace.

Definice 1. *Komponentou* $\mathbf{C}$ *nazýváme uspořádanou trojici* $\mathbf{F}$, $\mathbf{I_{in}}$, $\mathbf{I_{out}}$, *kde* $\mathbf{F}$ *je procesní základ komponenty (procesní chování),* $\mathbf{I_{in}}$ *je konečná množina všech požadovaných rozhraní a* $\mathbf{I_{out}}$ *je konečná množina všech nabízených rozhraní. Komponentu* $\mathbf{C}$ *zapisujeme ve tvaru* $\mathbf{C} = (\mathbf{F}, \mathbf{I_{in}}, \mathbf{I_{out}})$.

V souladu s nahoře uvedenou myšlenkou, definuji v dalším textu tyto základní pojmy, propojení, konformita (slabá/silná) a komponentový systém.

Definice 2. *Buďte* $\Im^{\mathbf{1}}_{\mathbf{out}}$, $\Im^{\mathbf{1}}_{\mathbf{in}}$ *nabízené a požadované rozhraní komponenty* $\mathbf{C_1}$. *Buďte* $\Im^{\mathbf{2}}_{\mathbf{out}}$, $\Im^{\mathbf{2}}_{\mathbf{in}}$ *nabízené a požadované rozhraní komponenty* $\mathbf{C_2}$. *Propojení dvou kompo-*

nent $\mathbf{C_1}$ *a* $\mathbf{C_2}$ *v daném pořadí je prvkem vazební relace* $\Re$*, zapsané v grafickém tvaru* $\Im^{\mathbf{1}}_{\mathbf{out}}$ —◎— $\Im^{\mathbf{2}}_{\mathbf{in}}$ *nebo ve tvaru* $\Im^{\mathbf{2}}_{\mathbf{out}}$ —◎— $\Im^{\mathbf{1}}_{\mathbf{in}}$*. Lze použít vysoce abstraktní notace ve tvaru* $(\Im^{\mathbf{1}}_{\mathbf{out}}\ \Re\ \Im^{\mathbf{2}}_{\mathbf{in}})$*,* $(\Im^{\mathbf{2}}_{\mathbf{out}}\ \Re\ \Im^{\mathbf{1}}_{\mathbf{in}})$*, nebo* $(\Im^{\mathbf{1}}_{\mathbf{out}}\ ,\ \Im^{\mathbf{2}}_{\mathbf{in}})$*,* $(\Im^{\mathbf{2}}_{\mathbf{out}}\ ,\ \Im^{\mathbf{1}}_{\mathbf{in}})$*.*

Vlastnosti vazební relace $\Re$ závisí na její implementační definici. Někdy jsou ceněny zejména její možné vlastnosti reflexivity, symetrie a tranzitivity, které se dají v implementační praxi vhodně využít.

Velmi často se v komponentovém systému uvažuje o záměně jedné komponenty jinou. Možnost záměny je řízena podle tzv. typové konformity obou komponent.

Definice 3. *Dvě komponenty* $\mathbf{C_1}$, $\mathbf{C_2}$ *jsou slabě typově konformní, podle vazební relace* $\Re$*:* $\mathbf{C_1}$ —○— $\mathbf{C_2}$*, jestliže existuje alespoň jedna rovnost* $\Im^{\mathbf{1}}_{\mathbf{in}} = \Im^{\mathbf{2}}_{\mathbf{out}}$ *nebo,* $\Im^{\mathbf{1}}_{\mathbf{out}} = \Im^{\mathbf{2}}_{\mathbf{in}}$ *a platí* $\mathbf{I^1_{out}} \neq \mathbf{I^2_{in}}$, $\mathbf{I^1_{in}} \neq \mathbf{I^2_{out}}$.

To značí, případy, např. kdy komponenta $\mathbf{C_1}$ poskytuje méně svých nabízených rozhraní, než komponenta $\mathbf{C_2}$ potřebuje. Konzistence procesního chování $\mathbf{F_1} \approx \mathbf{F_2}$ se nevylučuje. Jinými slovy, $\mathbf{C_1}$ a $\mathbf{C_2}$ jsou slabě typově konformní, jestliže mají stejné alespoň jedno požadované a nabízené rozhraní, ale ne všechna.

Slabá typová konformnost nestačí na výměnu komponent se zachováním života komponentového systému.

Definice 4. *Dvě komponenty* $\mathbf{C_1}$, $\mathbf{C_2}$ *jsou silně typově konformní, jestliže platí rovnost mezi všemi jejich nabízenými rozhraními a všemi požadovanými rozhraními a mají navíc konzistentní procesní chování.*

To značí velmi silný požadavek na jejich množiny nabízených a požadovaných rozhraní, tj. $\mathbf{I^1_{out}} = \mathbf{I^2_{in}}$ a $\mathbf{I^1_{in}} = \mathbf{I^2_{out}}$. Typově úplná konformnost zaručuje výměnu komponent bez narušení života komponentového systému.

Jelikož jsme již definovali základní pojmy, týkající se komponent, můžeme přikročit k definici Komponentového systému.

Definice 5. *Komponentovým systémem* **CS** *nazýváme uspořádanou trojici* $\mathbf{K}$, $\mathbf{L}$, $\mathbf{I}$*, kde* $\mathbf{K}$ *je množina všech komponent systému,* $\mathbf{L}$ *je množina všech propojení mezi komponentami systému a* $\mathbf{I}$ *je množina všech rozhraní komponentového systému. Komponentový systém zapisujeme ve tvaru* $\mathbf{CS} = (\mathbf{K}, \mathbf{L}, \mathbf{I})$*. Množina* $\mathbf{I}$ *je definována jako* $\mathbf{I} = (\mathbf{U^m_{i=1}I^i_{out}}, \mathbf{U^n_{j=1}I^j_{in}})$*, kde m, n jsou přirozená čísla.*

Na druhé straně se často množina L zapisuje notací kartézského součinu $\mathbf{L} \subseteq (\mathbf{U^m_{i=1}I^i_{out}xU^n_{j=1}I^j_{in}})$.

Množiny K a L tvoří základ *konfigurace* komponentového systému.

Definice 6. *Množina* $\mathbf{K}$ *všech komponent a množina* $\mathbf{L}$*, daná notací kartézského součinu* $\mathbf{L} \subseteq (\mathbf{U^m_{i=1}I^i_{out}xU^n_{j=1}I^j_{in}})$ *tvoří konfiguraci CO komponentového systému. Konfigurace se zapisuje ve tvaru* $\mathbf{CO} = (\mathbf{K}, \mathbf{L})$.

Jakákoliv změna množin $\mathbf{L}$ a $\mathbf{K}$ vede k tzv. **rekonfiguraci** komponentového systému. Mimo jiné, v komponentovém systému je právě tolik propojení, kolik je dimenzionalita množiny $\mathbf{L}$. Tedy, $|\mathbf{L}| \leq \mathbf{m.n}$, kde m, n jsou dimensionality konečných množin $\mathbf{U^m_{i=1}I^i_{out}}$, $\mathbf{U^n_{j=1}I^j_{in}}$. Dimenze množiny L ovšem závisí na implementačně

definované vazební relaci $\Re$. To však znamená, že ne všechna rozhraní musí být podle této relace svázána. V komponentovém systému může být dokonce zavedeno více typů vazební relace $\Re$.

Obecně se může stát, že komponenta obsahuje primitivní rekurzivitu v propojení, která je definována následovně:

Definice 7. *Buď dána komponenta* $\mathbf{C} = (\mathbf{F}, \mathbf{I_{in}}, \mathbf{I_{out}})$. *Komponenta* $\mathbf{C}$ *obsahuje primitivní rekurzivitu, jestliže platí* $\mathbf{I_{out}} \cap \mathbf{I_{in}} \neq \emptyset$.

Primitivní rekurzivita je nežádoucí a mohla by zkreslovat výsledky systémových úloh nad komponentovými systémy. O rozhraních platí tyto obecné poznatky:

1. Komponenta $\mathbf{C}$ *nabízí* ostatním komponentám ty své operace, jimiž dodávanou informaci tyto komponenty potřebují. *Nabízené* interface komponenty $\mathbf{C}$ je potom těmito operacemi tvořeno. Komponenta $\mathbf{C}$ požaduje od ostatních komponent ty informace, které dodávají jejich operace. Tyto operace tvoří požadované interface komponenty $\mathbf{C}$.
2. Komponentový systém má zaručenou *ostrou* orchestraci (soulad) $\mathbf{O_{or}}$, jestliže komponenty nabízejí *právě* jen to, co se potřebuje a naopak, a jen to, co komponenty žádají, se jim *právě* poskytuje.
3. Komponentový systém má zaručenou *měkkou* orchestraci (soulad) $\mathbf{O_{mě}}$, jestliže: komponenty nabízejí to, co se potřebuje (a nejen to) a naopak, co komponenty žádají (a nejen to), se jim poskytuje.

Na základě kvality množin $\mathbf{I_{out}}$, $\mathbf{I_{in}}$ lze vyslovit následující tvrzení.

Tvrzení 3:
Jestliže množina $\mathbf{I_{out}} \cup \mathbf{I_{in}}$ je symetrická, potom komponentový systém má ostrou orchestraci $\mathbf{O_{or}}$.

Důkaz: Součin $\forall i, j (U_{i=1}^{m} I_{out}^{i} \times U_{j=1}^{n} I_{in}^{j}) = I_{out} \times I_{in}$ ilustruje spárování interface ve smyslu relace $\Re$. Naopak součin $\forall i, j (U_{j=1}^{n} I_{in}^{j} \times U_{i=1}^{m} I_{out}^{i}) = I_{in} \times I_{out}$ ilustruje spárování interface ve smyslu $\Re^{-1}$. Jestliže $\Re \subset \Re^{-1}$, potom má komponentový systém s rozhraními $\mathbf{I_{out}}$, $\mathbf{I_{in}}$ ostrou orchestraci $\mathbf{O_{or}}$ a množina $\mathbf{I}$ všech rozhraní je symetrická (viz obr. 3).

7.1 Systémové úlohy nad komponentovým systémem

Teorie systémů přináší mnoho úloh, které je možno řešit nad obecnými systémy a rovněž je možno je implementovat pro komponentové systémy. Mnohé z nich využívají speciální systémovou algebru. V ukázce se budu zabývat jen úlohami konstrukčního přístupu a úlohy aplikační ponechám stranou.

Mezi úlohy konstrukčního přístupu patří:

1. Úloha konstrukce komponentového systému.
2. Úlohy verifikace komponentového systému:
   a) Analýza interface a propojení dvojic komponent.
   b) Úloha o servisní logice (hledání funkcionálních řetězců komponent).
   c) Úloha o zátěži komponent.
   d) Úloha o orchestraci.
   e) Úlohy o dynamice.

Mezi úlohy aplikačního přístupu patří:

1. Úloha implementace komponentového systému.
2. Úloha o relaci komponentového systému a modelované domény.
3. Úloha o údržbě implementace komponentového systému.

Všechny úlohy náleží k obecným komponentovým systémům vyjma úloh o dynamice, které náleží jen dynamickým komponentovým systémům.

Výsledky úloh *konstrukčního přístupu* se chápou jako užitečná výpomoc informatikovi:

1. ve vývoji komponentového systému,
2. ve verifikaci výsledků informační analýzy modelované domény,
3. ve verifikaci komponentové architektury,
4. ve verifikaci dynamických vlastností komponentového systému.

Úlohu konstrukce komponentového systému budu řešit na základě integrace procesní metody Eriksson-Penker (Eriksson, Penker, 2000), Metamodelu vývoje komponentového systému (MM) a moderní objektové metodiky (OM) označované jako *E-P+MM+OM*. V úloze budou rozebrány nabízené pracovní postupy metamodelu a objektové metodiky s cílem konstrukce komponentového systému. Jako pomocná procesní metoda se používá metoda Eriksson-Penker (E-P).

Úloha *verifikace* komponentového systému, tj. výsledku informační analýzy dané problémové domény, má strukturální charakter. Na komponentách se pomocí konečného automatu bude vyšetřovat syntaxní korektnost notací všech interface (nabízených / vyžadovaných), analyzuje se výskyt rekurzivity interface komponent a analyzuje se existence dvojic komponent propojených prostřednictvím interface.

Na základě převodu komponentového systému na obecný orientovaný multigraf a pomocí grafových operací se budou hledat konečné (necyklické) cesty, které jako

konečné funkcionální řetězce komponent reprezentují servisní logiku komponentového systému. Zjišťují se četnosti výskytu komponent v různých funkcionálních řetězcích a usuzuje se na jejich zátěž (využití) v komponentovém systému.

Úlohy o *dynamice* budou analyzovat předpoklady komponentového systému, řešit v rovině informačního modelování některé dynamické operace v rámci funkcionálních řetězců, např. záměnu komponent, odstranění komponenty, přidání nové komponenty, změnu interface, modifikaci propojení, které se provedou za běhu komponentového systému. Pro dynamické komponentové systémy se samozřejmě uvažuje jediná instalace u zákazníka.

Na druhé straně, úlohy *aplikačního přístupu* pomáhají řešit některé problémy implementace komponentového systému u zákazníků, korekci komputerizace problémové domény a údržbu instalovaného komponentového systému na informační infrastruktuře zákazníka. Obecně jsou problémy instalace komponentového systému odlišné pro statický a dynamický komponentový systém.

7.2 Úloha o konstrukci komponentového systému

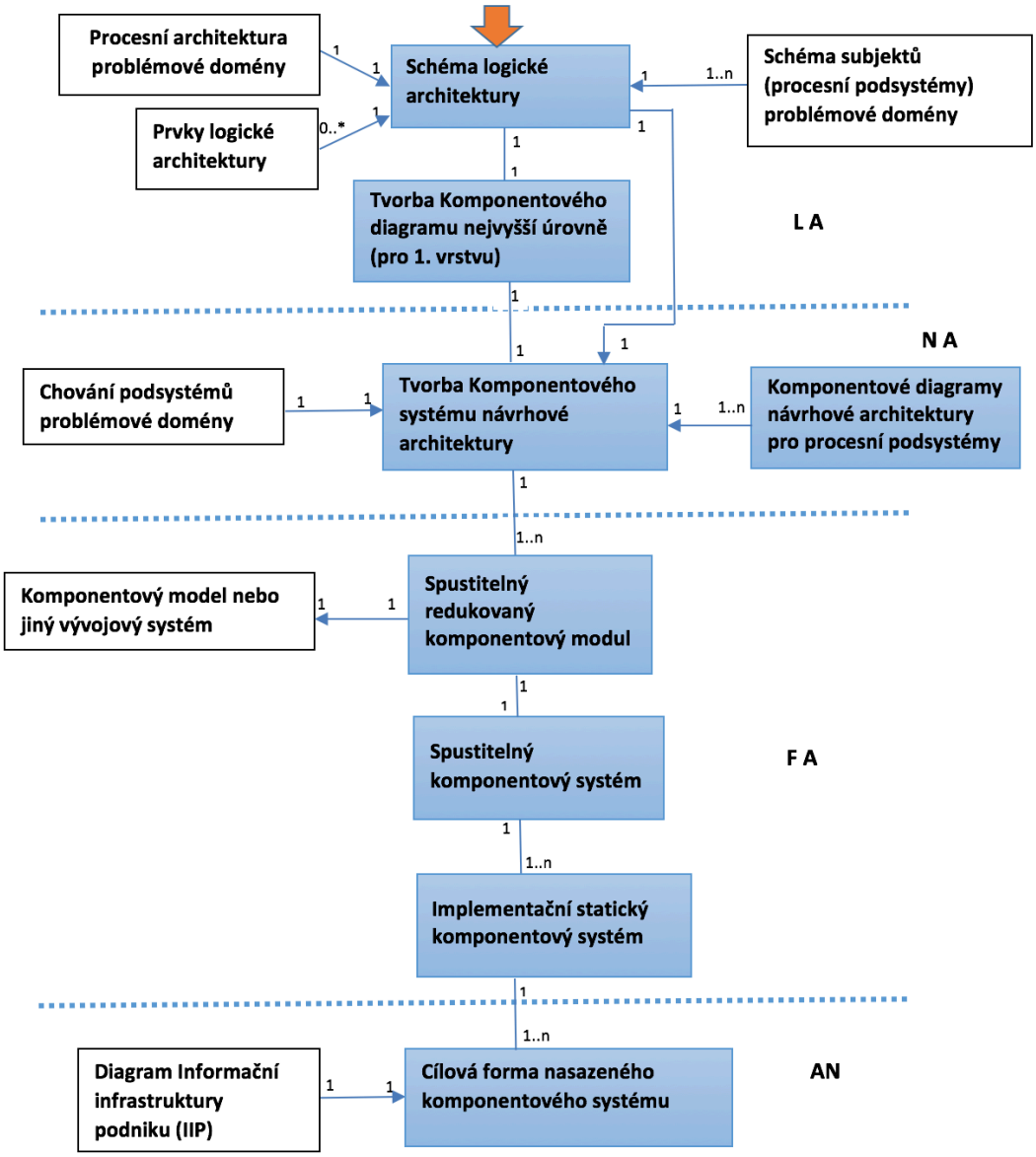
Navrhuji, aby základ úlohy vývoje komponentového systému a jeho integrace, byl založen na E-P (Eriksson-Penker) procesní metodě, moderní objektové metodice (OM) a řídícím metamodelu (MM). Vzniká tak účelová objektová metodika E-P+MM+OM (viz (Faldík, 2014)) pro vývoj Komponentového systému a jeho software. Tato integrace s vyspělou objektovou metodikou, např. typu UP, RUP, spočívá ve správném souladu pracovních postupů metamodelu s pracovními postupy použité objektové metodiky. Integrace je potom založena na konzistenci jejich vstupních a výstupních artefaktů. Východiskem informačního modelování vybrané procesní domény jsou vrstvené procesní diagramy produkované pro danou doménu pomocí E-P metody.

Pro vývoj komponentových systémů jsou používány zejména moderní objektové metodiky jako UP (Unified Process) nebo RUP (Rational Unified Process). V těchto metodikách není dominance komponentové architektury tvrdě prosazována. Často se objevují metamodely, které požadavek dominance komponentové architektury zabezpečují. Následující metamodel MM vývoje komponentového systému je hlavní součástí metodiky E-P+MM+OM pro vývoj Komponentového systému a jeho software.

Navržený řídící metamodel (viz obr. 9) se skládá ze čtyř částí – pracovních postupů:

- Logická architektura (LA),
- Návrhová architektura (NA),
- Fyzická architektura - Implementace (FA),
- Architektura nasazení (AN).

Uplatnění metamodelu pro vybranou problémovou doménu sice obvykle vyvolá implementační rozdíly, ale základní rysy metamodelu jsou zachovány. Nová metodika E-P+MM+OM má následující vstupní a výstupní artefakty (viz obr. 9):



Obrázek 9: Vstupní a výstupní artefakty metamodelu Komponentový vývoj software. Metamodel v notaci UML.

Přehled pracovních postupů metodiky E-P+MM+OM:

Pracovní postupy	**Původ**
Plánování softwarového projektu	OM
Procesní analýza domény	E-P
Logická architektura	MM
Požadavky	OM
Analýza a realizace případů užití	OM
Návrhová architektura	MM
Fyzická architektura	MM
implementace	
Architektura nasazení	MM

Pracovní postupy Plánování softwarového projektu, Požadavky a Analýza a realizace případů užití nebudu podrobně popisovat, protože jsou známé z pokročilých objektových metodik. Na rozdíl od toho, pracovní postupy metamodelu Komponentový vývoj software budou popsány podrobněji. Metamodel je formulován pro jeden podsystém problémové domény a pro jeho srozumitelnost je nutná jeho implementace na všechny procesní podsystémy.

Cílem pracovního postupu *Logická architektura* je využít procesního a datového pohledu na subjekty, které mohou být zakresleny hrubým procesním schématem bez vazeb mezi nimi. Za subjekty se považují procesní podsystémy modelované domény. Hrubé schéma procesních podsystémů pomůže k přenosu relevantních vlastností problémové domény do Schématu logické architektury. Toto schéma je doplněno mnoha obecnými prvky (ikony počítače, báze dat, řízení, …), čímž se logika výsledného software ještě zvýrazní. Zmíněné schéma je prvotním pohledem na architekturu výsledného software a přechází přes rovinu problémové domény (manažerský pohled) a rovinu softwarovou (analyticko-programátorský pohled). Pohled je velmi užitečný pro programátora, ačkoliv mu nedává žádné poznatky kromě logických.

V Logické architektuře se rovněž navrhuje Komponentový diagram nejvyšší vrstvy (1. vrstva), přičemž za komponenty jsou považovány procesní podsystémy jako nejvyšší procesy. Vazby mezi komponentami se určí pomocí procesního E-P diagramu 1. vrstvy.

Pracovní postup *Návrhová architektura* participuje na provedených pracovních postupech objektové metodiky: Požadavky, Analýza analytických a návrhových tříd a potom využívá již definovaný obsah, tj. vnitřní chování všech podsystémů v problémové doméně. Na základě rozsahu obsahu se provede mapování případů užití na komponenty, přičemž mohou vzniknout primitivní a složené komponenty. Tak vznikají Komponentové diagramy návrhové architektury pro všechny procesní podsystémy. Jejich kolekce vytváří Komponentový systém celé domény. Tento diagram je nejprve v redukované formě v pracovním postupu *Fyzická architektura* programován a testován. Je-li vše v pořádku, tak je celý Komponentový systém návrhové

architektury doprogramován a upraven na formu nasazení podle koncepce Informační infrastruktury zákazníka.

7.3 Popis pracovních postupů metodiky E-P+MM+OM

Logická architektura

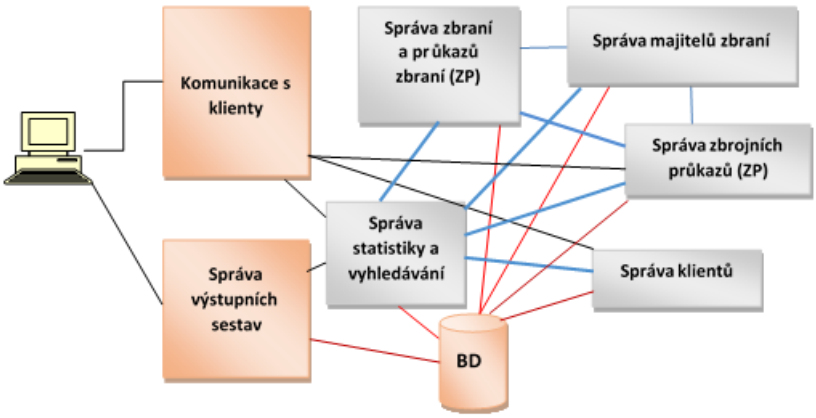
Logická architektura se uplatňuje na celou doménu. Architektura rozsáhlého software je jedním z relevantních atributů software. Pochopitelně, cesta, která vede k jejímu modelování a realizaci, je poměrně dlouhá a silně závislá na procesech (požadavcích) výchozí problémové domény. Vytváření komponentové architektury má na starosti metamodel Komponentový vývoj software. Tento metamodel vede postupně vývojáře přes své zbývající pracovní postupy. Logická architektura vychází z požadavků problémové domény a je nezávislá na informačních technologiích. Jelikož není možné neuvažovat o architektuře software, je logická architektura prostředkem přechodu od požadavků k návrhovému modelu architektury a potom ke konstrukci fyzické architektury výsledného software. Logická architektura může být postavena na podnikových procesních podsystémech, na jejich částech, na procesních setech, na procesních subsetech, na organizačních jednotkách, nebo na vymezených skupinách podnikových dat. Nejčastěji je ovšem používána procesní platforma.

Základem pro Logickou architekturu je publikace (Eeles, Cripps, 2011), která je značně vzdálená od problematiky objektových vývojových metodik. Základem pro výstavbu Schématu logické architektury jsou tzv. logické prvky a prvky problémové domény. Za prvky problémové domény se považují především její procesní podsystémy. Za logické prvky se považují např. ikona počítače, tiskárny, ikona báze dat, atd. Za tyto prvky mohou být považovány rovněž takové procesní prvky, které nejsou funkční z dané domény, např. proces přístupu, identifikace klientů, různé statistiky, atd. Na základě procesních podsystémů se může sestrojit tzv. prvotní pohled na problémovou doménu bez jakýchkoli vazeb mezi podsystémy. Později mohou být procesní podsystémy propojeny pomocí některé z obecných relací, např. relací vzájemné závislosti. Tyto vazby jsou určeny z Eriksson-Penkerova procesního diagramu nejvyšší úrovně, tj. z 1. vrstvy.

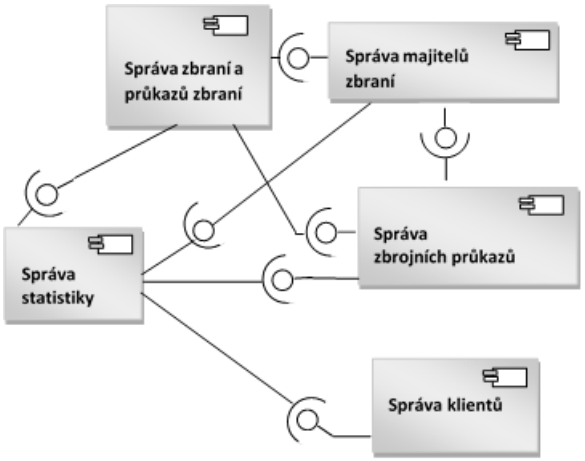
Příklad 1:

Následující Schéma logické architektury (viz obr. 10) je sestaveno pro doménu Evidence soukromých zbraní na teritoriu ČR. Logický prvek báze dat (BD) byl přidán pro zvýraznění Data-driven software.

Součástí Logické architektury je rovněž tvorba náčrtu Komponentového diagramu (viz obr. 11) nejvyšší vrstvy procesů. Komponenty, přičemž každý podsystém jako nejvyšší proces Eriksson-Penkerova procesního diagramu je mapován na jednu komponentu, přebírají vazby ze Schématu logické architektury a E-P diagramu a můžeme je využít jako základ pro náčrt rozhraní v Komponentovém diagramu nejvyšší vrstvy.



Obrázek 10: Schéma logické architektury software pro doménu Evidence soukromých zbraní v ČR. Černé spojnice jsou vazby typu Klient-Podsystém, červené typu Podsystém-BD, modré typu Podsystém-Podsystém



Obrázek 11: Náčrt Komponentového diagramu nejvyšší vrstvy, tj. 1. vrstvy pro doménu Evidence soukromých zbraní v ČR

Návrhová architektura

Uplatňuje se postupně na všechny procesní podsystémy. Návrhová architektura v mnoha případech žádá jako vstupní artefakty především procesní diagram 2. vrstvy, Schéma logické architektury, balíčky případů užití a diagramy analytických tříd a související Sekvenční diagramy. Balíčky případů užití, Diagramy analytických tříd a související Sekvenční diagramy tvoří tzv. chování procesního podsystému.

Výstupním artefaktem je nejdříve Komponentový diagram návrhové architektury, pro procesní podsystém, s komponentami, které jsou vytvořeny mapováním vstupních artefaktů. Je to diagram 2. vrstvy. Jestliže jsou jeho komponenty příliš složité, mohou být členěny na konečný počet dceřiných komponent a vznikají další Komponentové diagramy nižších vrstev tvořící kolekci vzájemně závislých kompo-

nentových diagramů.

Může se ovšem stát, že E-P procesní diagram je vytvořen pro malé domény jen pro první vrstvu, ve které jsou za procesy považovány právě procesní podsystémy.

Postup v Návrhové architektuře se dá formalizovat následovně: Zabývejme se teď jediným procesním podsystémem S. Buď n přirozené číslo, počet procesů 2. vrstvy. Je-li již celý podsystém S informačně analyzován, tak jsme museli vytvořit:

$B_1, B_2, B_3, ..., B_n$	**n** balíčků případů užití, každý balíček pro jeden proces 2. vrstvy a jeho podřízené procesy 3. vrstvy
$T_1, T_2, T_3, ..., T_n$	**n** diagramů realizačních tříd,
$SD_1, SD_2, SD_3, ..., SD_m$	**m** Sekvenčních diagramů, kde $m \geq n$.

Mapování vstupních artefaktů se provede na základě následujícího algoritmu: Vytvoří se prázdné komponenty $C_1, C_2, ...C_n$. Tyto komponenty nemají zatím žádné propojení.

$\forall i$ (B_i se převede do C_i)	mapují se diagramy analytických tříd do komponent,
$\forall i$ (T_i se převede do C_i)	mapují se balíčky případů užití do komponent,
$\forall \leq$ m (SD_i se převede do C_i)	mapují se sekvenční diagramy do komponent.

Tímto algoritmem je vytvořen Komponentový diagram návrhové architektury 2. vrstvy pro podsystém S. Je vytvořena podstata každé komponenty - specifikace. Do specifikace každé z komponent chybí ovšem rozhraní. To určíme podle Tvrzení 4.

Tvrzení 4: Buďte r, s přirozenými čísly. Buď P proces z 2. vrstvy Eriksson-Penker procesního diagramu a $(p_1, p_2, ..., p_r)$ jeho podřízené procesy. Pro proces P platí následující řetězec jeho využití: $P \cong B_i \cong T_i \cong SD_i \cong C_i$, kde $\cong$ značí, že levý parametr se převede do pravého parametru. Podobná situace se uskuteční pro jiný proces $P' = (p'_1, p'_2, ..., p'_r)$ z 2. vrstvy, tedy $P' \cong B_j \cong T_j \cong SD_j \cong C_j$.

Jestliže v E-P diagramu 2. vrstvy je z procesu P vazba typu **output**, končící na procesu P' jako typu **input**, potom v komponentovém diagramu je vazba C_i —◎— C_j.

Jestliže v E-P diagramu 2. vrstvy je z procesu P' vazba typu **output**, končící na procesu P jako typu **input**, potom v komponentovém diagramu je vazba typu C_j —◎— C_i.

Komponentový systém návrhové architektury je nejdůležitějším komplexem produkovaným metamodelem Komponentový vývoj software. Je vstupem pro další dva pracovní postupy *Fyzická architektura* a *Architektura nasazení.*

Informačním modelováním můžeme dospět k těmto dílčím komponentovým diagramům:

- Komponentový diagram nejvyšší úrovně pro 1. vrstvu.

- Komponentové diagramy 2. vrstvy pro jednotlivé procesní podsystémy modelované domény.
- Případně k nižším komponentovým diagramům, které zjemňují Komponentové diagramy 2. vrstvy do jemnějších komponentových diagramů 3. vrstvy.

Tyto tři možnosti specifikují jeden podstrom grafu pro složení tzv. finálního Komponentového systému pro danou problémovou doménu. Je potřebné si všimnout, že výkonné komponenty jsou až na obálce stromu Komponentového systému.

Fyzická architektura

V tomto pracovním postupu se Komponentový systém návrhové architektury celé domény realizuje buď pomocí vhodného programovacího jazyka, nebo pomocí vybraného komponentového modelu.

Vstupním artefaktem je Komponentový systém návrhové architektury problémové domény. Výstupním artefaktem je pokusný spustitelný Komponentový modul v redukované formě a potom Implementační komponentový systém.

Spustitelný komponentový modul podrobíme několika prověrkám:

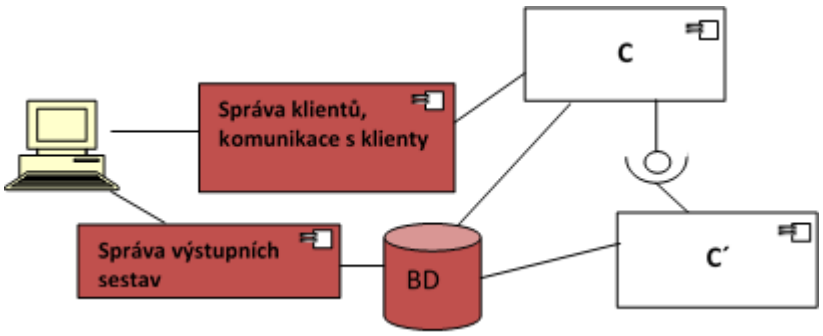
- Testování správnosti kvalifikace klientů software.
- Testování propojení komponent podle jejich rozhraní.
- Deklarace obsahu komponent.
- Realizace chodu Spustitelného komponentového modulu pro software.

Jak již bylo naznačeno, za *Spustitelný komponentový modul* považujeme tzv. redukovanou formu Komponentového systému modelované problémové domény (viz obr. 12):

- Vybereme dvě relevantní komponenty C a C´ z obálky stromu Komponentového systému. Mezi těmito dvěma komponentami se prověří realita jejich propojení na základě interface.
- Přidáme komponenty ze Schématu logické architektury, které pečují o komunikaci s klienty a výstupní sestavy.
- Přidáme bázi dat BD kvůli typu Data-driven aplikace.

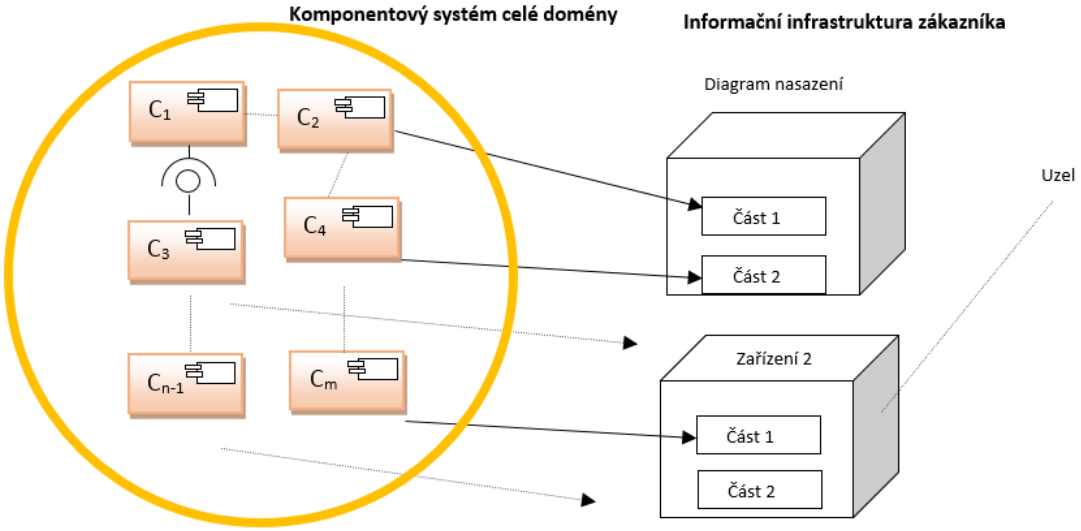
Jestliže naprogramovaný *Spustitelný redukovaný komponentový modul* funguje podle našich představ, můžeme přikročit k programování celé jeho obálky. Dojde vlastně k převodu Komponentového systému procesních podsystémů do kódu.

Jestliže jsme tento postup provedli pro všechny procesní podsystémy dané domény, tak dostáváme její Implementační komponentový systém.



Obrázek 12: Koncepce redukované formy Komponentového systému modelované problémové domény

Uvažujme jen teoretický Komponentový systém celé domény a teoretickou Informační infrastrukturu zákazníka. Obrázek 13 ilustruje vznik Implementačního schématu komponentového systému.



Obrázek 13: Teoretické implementační schéma Komponentového systému celé domény

Architektura nasazení

Architektura nasazení čerpá z problematiky fáze *Zavedení objektové metodiky* a již dokončuje všechny architektonické aktivity předešlých pracovních postupů ohledně modelování a završení tvorby kódu software. Je tedy zřejmé, že je značně dopracovaná (ne-li dokončená) architektura a kód software a jde již jen o nasazení software na informační infrastrukturu organizace. Základem postupu je dopracování fyzické architektury software. To může být obsahem dvou následujících kroků:

- Architektonická implementace.
- Tvorba Diagramu a modelu nasazení.

Krok architektonická implementace

Hlavním cílem architektonické implementace je rozpoznat architektonicky významné komponenty a jejich mapování na informační infrastrukturu organizace.

Krok tvorba Diagramu nasazení a Modelu nasazení

Nasazení je v objektové metodice chápáno jako přiřazení artefaktů jednotlivým uzlům v IIS organizace.

Diagram nasazení musí proto ukazovat nejen hardware, ale rovněž jak je daný software na tomto hardware nasazen.

Existují dvě verze Diagramu nasazení:

1. **Obecný Diagram nasazení** (Descriptor form for Deployment diagram), který mapuje **logickou architekturu**, která je dokončena v pracovním postupu **Logická architektura**, na tzv. **fyzickou architekturu**. Fyzická architektura je tedy silná asociace **logické architektury** a typů **hardware** a **artefaktů** (obecné uzly a artefakty, které můžeme později upřesnit jejich konkrétními instancemi-výskyty).
2. **Diagram nasazení** (Instance form for Deployment diagram) pro konkrétní nasazení software v dané organizaci s konkrétní informační infrastrukturou (IIS) musí obsahovat:
   - instance uzlů (specifikovaná část hardware z IIS organizace), relace mezi nimi,
   - instance komponent (zastupující konkrétní software – instance artefaktů),
   - relace mezi instancemi uzlů a instancemi artefaktů.

Tvorba potřebných deskripcí (konkrétního specifikovaného hardware, instancí komponent a relací), včetně uložení v prostředku CASE potom dotváří Model nasazení daného software.

Konkrétní Diagramy nasazení jsou dobře čitelné jak managementu, tak i analytikům a programátorům produkční softwarové firmy.

Ačkoliv zde není vysvětlen způsob využití E-P procesních diagramů pro tvorbu Komponentových diagramů 1., 2. a případně 3. vrstvy, je z textu patrno jejich základní poslání. Proces jejich souvislosti s komponentovými diagramy je uveden v publikaci (Faldík, 2014).

7.4 Úlohy verifikace komponentového systému

Tyto úlohy jsou užitečné pro zdárné ukončení informačního modelování problémové domény. Dají informatikovi možnost odstranění chyb a tak zabezpečit korektnost výsledků informačního modelování pro rozsáhlé domény.

7.5 Analýza interface a propojení dvojic primitivních komponent

Základem pro komunikaci komponent v komponentovém systému je interface komponent, přesněji interface nabízené (provided) a požadované (required). Předpokládejme, že notace obou typů interface jsou syntaxně stejné a mohou to být některé z následujících variant:

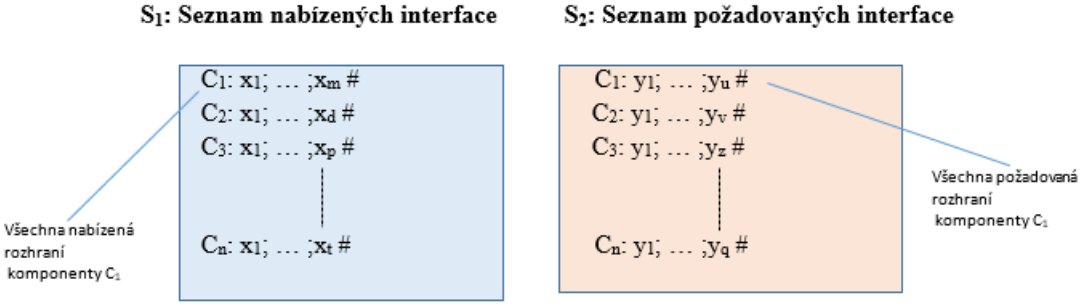
jmeno()	zápis bezparametrové operace
jmeno(u)	zápis operace s parametrem
jmeno(u,v,w)	zápis operace s více parametry.

Je-li interface seznamem notací více operací, potom se operace oddělují mezerou a interface se ukončují znakem #

Úlohu budeme chápat tak, že by měla informatikovi dát:

- informaci o syntakticky nesprávných notacích interface,
- informaci o primitivní rekurzivitě,
- seznam dvojic komponent propojených na základě interface,
- seznam komponent, které nejsou propojeny s jinou komponentou

Buď $\mathbf{x_i}$ notace nabízeného interface a $\mathbf{y_i}$ notace požadovaného interface komponenty $\mathbf{C_i}$. Buď $\mathbf{n}$ počet komponent komponentového diagramu, který byl v rámci informačního modelování sestrojen. *Tento diagram je informatikem doplněn o dva pomocné seznamy interface* $\mathbf{S_1}$ a $\mathbf{S_2}$. V seznamech (viz obr. 14) jsou interface dané komponenty seskupeny po operacích a odděleny středníkem.



Obrázek 14: Seznamy požadovaných a nabízených interface

Oba seznamy přesně adresují komponenty a notace jejich interface. Každá komponenta může mít konečný počet interface nabízených a konečný počet interface požadovaných.

Prověrku syntaxe svěříme konečnému deterministickému automatu
$\mathbf{M} = (\mathbf{S}, \Sigma, \delta, \mathbf{p_0}, \mathbf{F})$, kde je

S	množina stavů
Σ	abeceda notací interface
δ	zobrazení $S \times \Sigma \rightarrow S$
p_0	počáteční stav
F	množina koncových stavů
$Q, P_{1,2,3}$	jsou podmnožiny Σ^* obsahující postupně jméno operace, po znaku (, jména parametrů, po každé čárce a znaku)

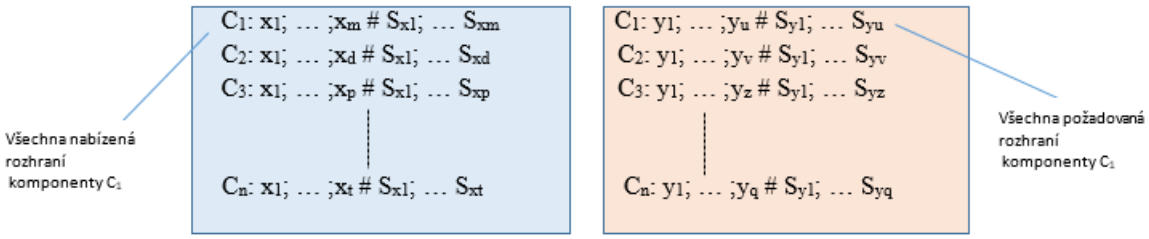
Automat **M** jsme pro danou syntaxi notací interface schopni prakticky sestrojit tak, aby pro syntaxně správné notace x_i a y_i splňoval předpisy:
$\mathbf{M(x_i)} = \{\mathbf{x_i}|(\mathbf{x_i}\epsilon\Sigma^*) \wedge\delta(\mathbf{p_0}, \mathbf{x_i})\epsilon\mathbf{F}\}$, $\mathbf{M(y_i)} = \{\mathbf{y_i}|(\mathbf{y_i}\epsilon\ \Sigma^*) \wedge\delta(\mathbf{p_0}, \mathbf{y_i})\epsilon\mathbf{F}\}$.

Algoritmus $\overline{A_1}$ prověrky správnosti syntaxe interface je následující:

- Automatu **M** postupně předložíme všechny notace interface jednotlivých komponent. O chybných notacích podá automat M zprávu prostřednictvím indikace chybného stavu.
- Automat **M** přidá do seznamů $\mathbf{S_1}$ a $\mathbf{S_2}$ pro každé x a y množinu stavů $\mathbf{S_x}$, $\mathbf{S_y}$, kterými prošel při přijetí/nepřijetí řetězců x a y.

Automat **M**, který provede předešlé operace vyšetření syntaxe, vytvoří tedy pomocí algoritmu $\overline{A_1}$ pro každé $\mathbf{x}$ a $\mathbf{y}$ konečné uspořádané množiny $\mathbf{S_x}$, $\mathbf{S_y}$ a přidá je do seznamů $\mathbf{S_1}$ a $\mathbf{S_2}$:
$\mathbf{S_x} = \{\mathbf{p_0}, \delta_{\mathbf{1}}^{\mathbf{x}}, \delta_{\mathbf{2}}^{\mathbf{x}}, \ldots\delta_{\mathbf{t}}^{\mathbf{x}}\}$ stavy po přijetí/nepřijetí řetězců $\mathbf{x}$ a $\mathbf{y}$.
$\mathbf{S_y} = \{\mathbf{p_0}, \delta_{\mathbf{1}}^{\mathbf{y}}, \delta_{\mathbf{2}}^{\mathbf{y}}, \ldots\delta_{\mathbf{u}}^{\mathbf{y}}\}$
Na základě těchto množin můžeme charakterizovat možnost propojení komponent algoritmem $\overline{A_2}$.

Dostaneme tak veškeré informace o interface každé komponenty a můžeme se soustředit na porovnání množin S_x a S_y u všech komponent a najít dvojice propojení (viz obr. 15).



Obrázek 15: Seznamy požadovaných a nabízených interface - porovnání

Tvrzení 5:

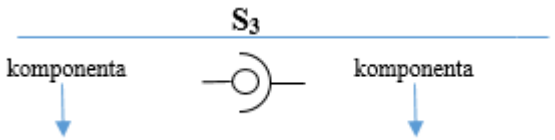
Buďte **C** a **C'** dvě komponenty komponentového systému **CS**. Buď **x** notace nabízeného interface komponenty **C** a **y** notace požadovaného interface komponenty **C'**.

Komponenty **C** a **C'** jsou propojitelné ve smyslu **C** ⌒ **C'** slabé konformity, jestliže platí: jména operací jsou stejná a $(S_x \supseteq S_y)$.

Komponenty **C** a **C'** jsou propojitelné ve smyslu **C** ⌒ **C'** silné konformity, jestliže platí: jména operací jsou stejná a $(S_x = S_y)$.

Algoritmus $\overline{\mathbf{A_2}}$ nalezení propojení všech dvojic komponent je velmi jednoduchý, protože jde jen o porovnání množin S_x a S_y pro interface **x** - nabízené a interface **y** - požadované.

Na základě úlohy *„Nalezení dvojic propojených komponent“* algoritmem $\overline{\mathbf{A_2}}$, se potom pro informatika vytvoří třetí seznam $\mathbf{S_3}$ ve tvaru:



7.6 Analýza interface a propojení dvojic komponent

Cílem provádění informační analýzy dané domény je dosažení Komponentového systému domény. Komponentový systém může být členěn na vrstvy a v každé vrstvě je několik dílčích komponentových diagramů. Vztahy mezi vrstvami jsou reprezentovány mateřskými a jejich dceřinými komponentami, tedy agregací. Vzhledem k tomu, že diagramy v nejnižší vrstvě obsahují jen primitivní komponenty, a proto se zásadně programují jen komponentové diagramy nejnižší vrstvy a vyšší vrstvy se považují za abstraktní, významné jen pro modelování a vazby.

Informatik v podstatě nemá problémy stanovit interface všech komponent bez ohledu na to, jsou-li primitivní nebo ne. Je tedy možné dokončit seznamy $\mathbf{S_1}$ a $\mathbf{S_2}$, potom rozšířit úlohu *„Nalezení dvojic propojených komponent“* s algoritmem $\overline{\mathbf{A_2}}$ na

úplný Komponentový systém modelované problémové domény a dát informatikovi seznamy s indikacemi chyb, které využije pro korekci svých výsledků.

Příklad 2:
V tomto příkladu se pokusíme realizovat algoritmy $\overline{\mathbf{A_1}}$, $\overline{\mathbf{A_2}}$ pro problémovou doménu **Správa soukromých zbraní v ČR**, která se člení na tři procesní podsystémy:

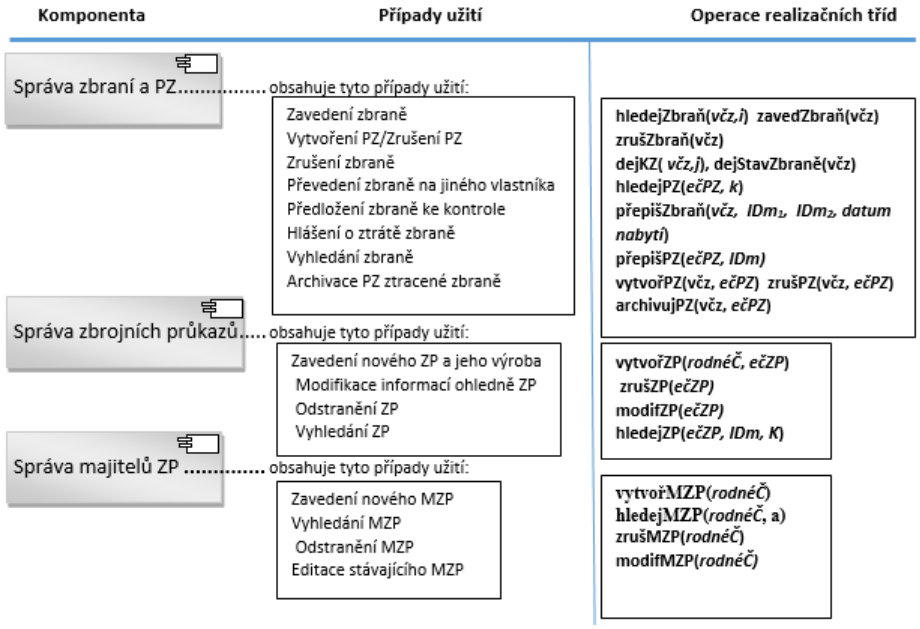
1. Správa zbraní a průkazů zbraní (PZ).
2. Správa majitelů zbrojních průkazů (MZP).
3. Správa zbrojních průkazů (ZP).

Z diagramu vybereme komponenty týkající se zájmové domény. Získáváme tak náčrt vazeb funkcionálních komponent (viz obr. 16).

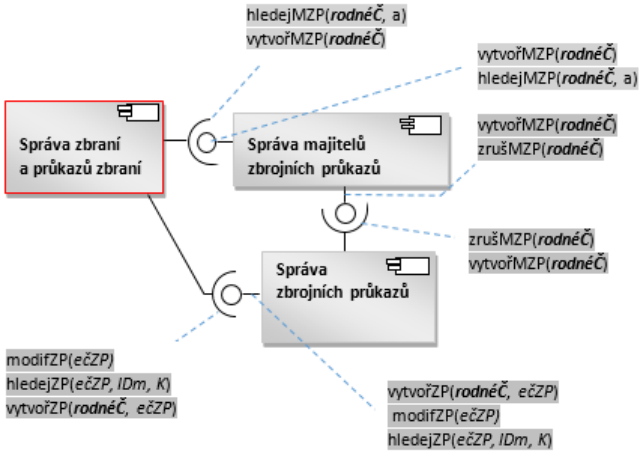
Obrázek 16: Komponentový diagram 1. vrstvy

Abychom zjistili jednotlivá interface, tak musíme:

- Stanovit, které případy užití budou v jednotlivých komponentách (viz obr. 17),
- jaké jsou operace objektových tříd, které tyto případy užití realizují.
- na základě operací sestavíme nabízená a požadovaná interface komponent (viz obr. 18).



Obrázek 17: Mapování případů užití do komponent, operace realizačních tříd

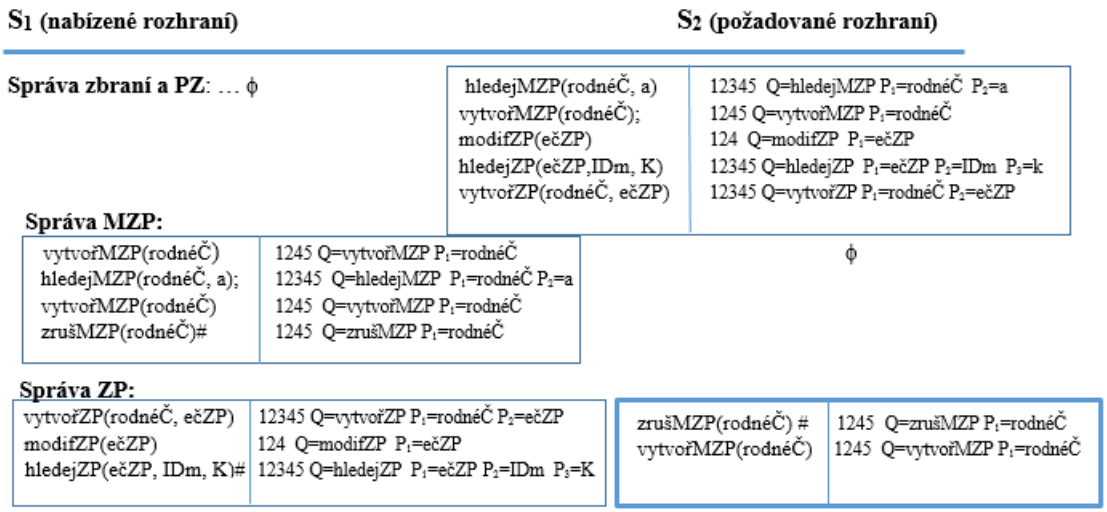


Obrázek 18: Stanovení interface komponent

Význam parametrů jednotlivých operací

i	ID majitele zbraně z evidence zbraně
včz	výrobní číslo zbraně
ečZP	evidenční číslo ZP
ečPZ	evidenční číslo PZ
a	ID majitele zbraně z jeho evidence
j	true/false –kontrola zbraně
k	kategorie zbraně z PZ
IDm	ID dvou MZP z jeho evidence
K	přípustné kategorie zbraní v ZP

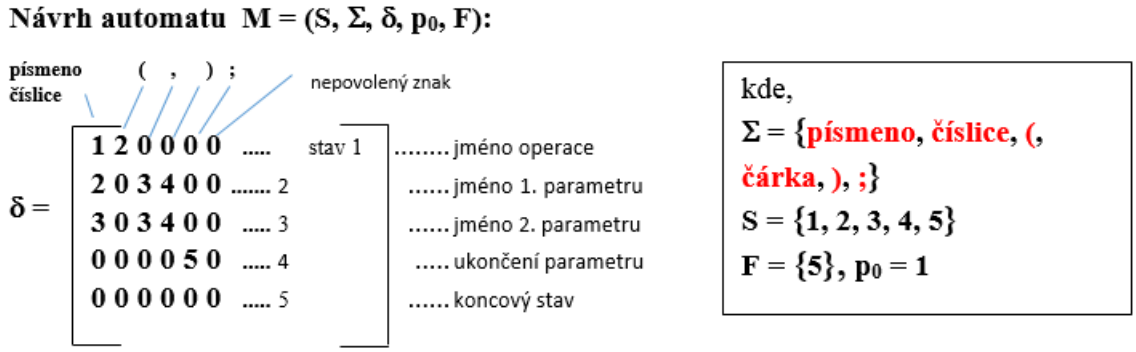
Informatik vytvořil následující seznamy (viz obr. 19), u kterých podle výsledků automatu M opravil syntax operací a podle dalších výsledků automatu doplnil stavové údaje jednotlivých operací:



Obrázek 19: Výsledné seznamy rozhraní

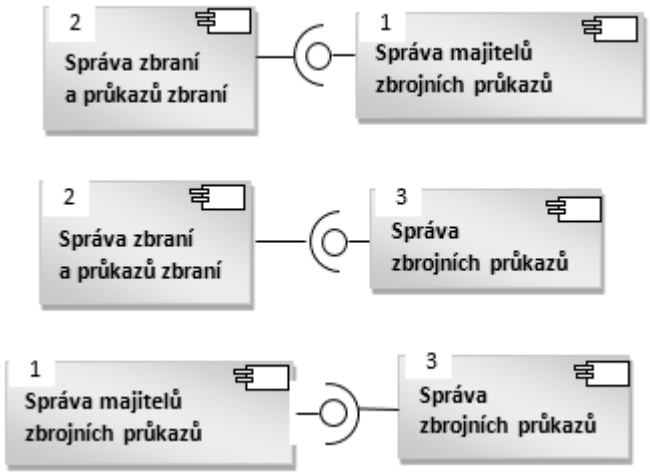
Na základě algoritmu $\overline{\mathbf{A_1}}$ automat M (viz obr. 20) zjistí syntaxní chybu ve dvou notacích operací vytvořMZP[rodČ), hledejZPečZP,IDm, k). *U první notace je použit nepřípustný znak, u druhé je nesprávné jméno operace.* Obě chyby jsou informatikem hned opraveny. Na základě automatu M se rovněž do seznamů S_1 a S_2 pomocí algoritmu $\overline{\mathbf{A_2}}$ doplní množiny S_x a S_y tak, jak je požadováno. Pokud probíhají stavy 1, 2, a 3, provádí se skladba jména operace v proměnné Q, a 1., 2. a třetího parametru v proměnných $\mathbf{P_1}$, $\mathbf{P_2}$ a $\mathbf{P_3}$.

Algoritmus $\overline{\mathbf{A_2}}$ doplní do seznamů S_1 a S_2 množiny stavů, které odpovídají přijetí operací realizačních tříd. Tím jsou oba dva seznamy připraveny k hledání dvojic komponent propojitelných na základě jejich interface v algoritmu $\overline{\mathbf{A_2}}$.



Obrázek 20: Návrh automatu M

Tento algoritmus páruje následující komponenty a vytváří tak seznam $\mathbf{S_3}$ (viz obr. 21).



Obrázek 21: Diagram spárovaných komponent

Ačkoliv je uvedený příklad jednoduchý, roste závažnost algoritmů $\overline{\mathbf{A_1}}$, $\overline{\mathbf{A_2}}$ se zvyšujícím se počtem komponent výsledného komponentového systému.

Úloha o servisní logice (hledání funkcionálních řetězců komponent)

Tato úloha je nesmírně důležitá pro vyšetření vnitřního chování komponent v komponentovém systému. Úlohu budeme chápat tak, že by měla informatikovi poskytnout:

- informaci o konečných necyklických funkcionálních řetězcích, reprezentujících servisní logiku komponentového systému,
- četnosti výskytu komponent v různých funkcionálních řetězcích, tedy velikost zátěže komponent v komponentovém systému.

Úloha o servisní logice se realizuje na orientovaném multigrafu, na který se komponentový systém **CS = (K, L, I)** převede.

Získaný multigraf $\mathbf{G} = (\mathbf{U}, \mathbf{H}, \mathbf{f}), \mathbf{f} : \mathbf{H} \rightarrow \mathbf{U^2}$, který je dán jako incidence uzlů a hran, je definován na základě následujícího transformačního předpisu **T**:

1. Každá komponenta $\mathbf{C} \epsilon \mathbf{CS}$ se stává uzlem (izolovaným/neizolovaným) grafu **G**.
2. Nechť komponenta $\mathbf{C_1}$ je již uzlem grafu **G**. Propojení ($\Im^{\mathbf{1}}_{\mathbf{out}}$, $\Im^{\mathbf{2}}_{\mathbf{in}}$) komponenty $\mathbf{C_1}$ na jinou komponentu $\mathbf{C_2}$ způsobí vznik uzlu $\mathbf{C_2}$ a orientované hrany ($\mathbf{C_1}$, $\mathbf{C_2}$).
3. Obrácené propojení ($\Im^{\mathbf{2}}_{\mathbf{out}}$, $\Im^{\mathbf{1}}_{\mathbf{in}}$), komponenty $\mathbf{C_2}$ na $\mathbf{C_1}$ způsobí vznik orientované hrany ($\mathbf{C_2}$, $\mathbf{C_1}$).

Transformační předpis převede nepropojenou komponentu na izolovaný uzel a propojené komponenty převede na hranu s dvěma uzly. Je pochopitelné, že význam grafu G stoupá se zvyšujícím se číslem komponent a propojení mezi nimi.

Předchozí úloha (Analýza interface a propojení dvojic komponent) připraví pro Úlohu o servisní logice splnění následujících podmínek:

1. Žádná komponenta nemá triviální rekurzivitu (informatik je odstraní).
2. Všechny notace interface jsou syntaxně správné a odpovídají operacím realizačních analytických tříd jednotlivých komponent.
3. Existuje seznam S_3 propojení dvojic komponent, který je korektní.
4. Neexistuje komponenta, která by nebyla ani v jednou propojení.

Tvorbu grafu G můžeme svěřit velmi schopné softwarové aplikaci. Jejím výsledkem je nejen graf G, ale rovněž množina grafových cest, které reprezentují funkcionální řetězce vzájemně propojených komponent. A navíc, aplikace může poskytnout mnoho dalších užitečných informací (cykly v propojení, nadbytečné interface, atd.). Vytvoří se tak, na základě informací v seznamech S_1, S_2, S_3, které byly produkovány pomocí algoritmů $\overline{\mathbf{A_1}}$, $\overline{\mathbf{A_2}}$, celková představa o životě komponentového systému.

Potom je na informatikovi, aby posoudil konzistenci dosažených výsledků se svou představou o funkcionalitě modelované domény a prohlásil je za korektní.

7.7 Hledání funkcionálních řetězců komponent

Definice 8. *Nechť* $\overline{\mathbf{A_3}}$ *je algoritmus založený na grafovém algoritmu* ***Nalezení všech cest z uzlu*** $\mathbf{C_i}$ ***do uzlu*** $\mathbf{C_j}$ ***, pro i=1, 2,..., j=1, 2,... , j > i****. Nechť je algoritmem* $\overline{\mathbf{A_3}}$ *vytvářen seznam* $\mathbf{S_4}$*, který obsahuje všechny nalezené cesty.*

Seznam $\mathbf{S_4}$ by měl odpovídat funkcionální povaze Komponentového systému.

Vytěžení komponent, tj. v kolika funkcionálních řetězcích se daná komponenta vyskytuje, umožní jednoduchý algoritmus $\overline{\mathbf{A_4}}$, napojený na $\overline{\mathbf{A_3}}$, který pro každou

komponentu její výskyty napočítá. Výsledek dává informatikovi procentovou informaci o využitelnosti jednotlivých komponent systémem na základě vztahu:

v/**z** .**100**% , kde **v** je počet výskytů komponenty v různých funkcionálních řetězcích, z je celkový počet funkcionálních řetězců. Počet výskytů je metrikou pro tzv. **důležitost** komponenty ve funkcionalitě problémové domény.

Příklad 3:
Uplatněním transformace seznamu $\mathbf{S_3}$, z předchozího příkladu, na orientovaný multigraf získáváme následující výsledek (viz obr. 22)

Algoritmus $\mathbf{S_3}$ uplatněný na tento graf dává dva funkcionální řetězce komponent: **1**–**3**–**2** a $\mathbf{1-2}$.

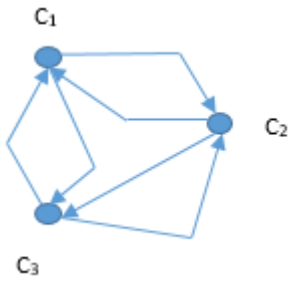
Obrázek 22: Orientovaný multigraf komponent

Komponenta **3** má jenom jeden výskyt. Komponenty **1**, **2** mají po dvou výskytech.

Úloha o orchestraci

Úloha využívá pojetí orchestrace komponentového systému tak, jak byla uvedena v **Tvrzení 3**. Úloha je proveditelná pomocí transformace komponentového systému na obecný multigraf. Totiž, ta rozhraní (poskytovaná/požadovaná), která se transformují na izolované uzly, již indikují, že v systému není ostrá orchestrace $\mathbf{O_{os}}$. Na druhé straně tyto uzly ale neindikují měkkou orchestraci $\mathbf{O_{mě}}$.

Tvrzení 5: Buď orientovaný graf **G** vzniklý transformací komponentového systému **CS** podle kapitoly 7. Pokud je tento graf symetrický, potom má komponentový systém ostrou orchestraci $\mathbf{O_{or}}$.



Obrázek 23: Symetrický orientovaný graf G

Důkaz:

Symetrický orientovaný graf G se vyznačuje vlastností, že pro každé dva uzly x, y je počet hran z uzlu x do uzlu y stejný, jako počet hran z uzlu y do uzlu x. Takovým grafem je např. následující graf (viz obr. 23).

Graf obsahuje tři komponenty C_1, C_2 a C_3. Vazby mezi komponentami jsou symetrické. Orchestrace je ostrá.

8 Využití interface automatů v modelování komponentových systémů

Mezi nástroji použitelnými k informačnímu modelování dynamických a inteligentních systémů, v době jejich návrhu a vývoje, se vývojáři často obracejí k systémům DEVS (Discrete Event System) založeným na diskrétních událostech. DEVS jsou obvykle definovány tak, aby byly schopné plně reprezentovat modelovaný diskrétní událostní systém. Za vzorový případ mohou sloužit Petriho sítě a Architektury založené na službách. Velmi často se konstatuje, že v roli DEVS se často používají formalismy vycházející z problematiky vyčíslitelnosti, přesněji z konečných automatů. Automaty jako takové, dokážou srozumitelněji popsat stavy a chování modelovaného dynamického systému, viz (Zeigler, Kim, Praehofer, 2000).

Za diskrétní událostní systém lze rovněž považovat komponentový systém. Rozhodl jsem se nejdříve využít pojetí tzv. událostní komponenty, zavedené v (Isazedech, Karimpour, 2008) a potom definovat konečný interface automat (interface automaton), pomocí kterého budu informačně modelovat komponentu a komponentové systémy.

Základní myšlenka přístupu k softwarovým komponentám a softwarovým komponentovým systémům prostřednictvím interface automatů (interface automata) spočívá ve stanovení zástupu stavové komponenty asociovaným interface automatem, který slouží jako modelovací a verifikační prostředek pro komponenty a komponentové systémy. Pomocí interface automatů se budou realizovat operace nad komponentami a komponentovými systémy v systému interface automatů bez ohledu na to, zde je operace dynamické nebo statické povahy. K tomu se využije specifický formální jazyk Ψ (viz 8.3), jako základní modelovací formalismus konečného interface automatu a systém D standardů a konvencí jak tento jazyk používat. A nakonec je pro jazyk Ψ a systém D vytvořen podpůrný *D-Framework* (viz 8.4). Formalismus, který je v podpůrném *Framework*-u zabudován, je orientován na informační modelování základních vlastností komponent a komponentových systémů (konstrukce interface automatu asociovaného s danou komponentou, propojení komponent, kompatibilita komponent) v pojetí interface automatů. Formalismus, který je v podpůrném *D-Framework*-u zabudován není orientován na exekutivu (spouštění) komponentových systémů.

Ukazuje se, že realizace běžných operací nad komponentami (operace v 2.10 a 7.1) je v systému interface automatů nejen možná, ale jsou realizovatelné rovněž operace, které se v komponentovém systému provádějí obtížně, zejména je-li komponentový systém robustní. Poznatky, ke kterým se v systému interface automatů dospěje, jsou potom interpretovány pro asociovaný komponentový systém.

Na druhé straně nemusí mít podpůrný framework „běhový“ charakter podobný existujícím komponentovým modelům (COM, DCOM, COM++, Java Beans) (Bock, 2010). Jeden z přístupů k základům práce s interface automaty je uveden v publikaci (Alfaro, Henzing, 2001).

Přístup, který je prezentován v této práci, je podobný přístupu v definici interface automatu, ale odlišný v následujících směrech:

- odlišný formalismus pro základní definice pojmů (komponenta, interface automat, vlastnosti interface automatů),
- odlišné pojetí základních vlastností interface automatů (kompozice, kompatibilita),
- zavedení formálního jazyka Ψ a programování operací nad systémem interface automatů,
- zavedení standardů a konvencí v systému D,
- zavedení a podstata podpůrného modelovacího frameworku,
- praktická realizace frameworku.

8.1 Formální definice událostní komponenty

Stavové pojetí komponenty musí být takové, aby k ní asociovaný Interface automat mohl imitovat všechny její vlastnosti.

Na předmětnou komponentu budou kladeny následující požadavky:

1. Je diskrétní povahy.
2. Je řízena množinou svých událostí.
3. Je vybavena interakčními porty typu output (pro hodnoty nabízené, označení P_{out}) a porty typu input (pro hodnoty požadované, označení P_{in}). Počet portů je konečný, tj. komponenta C má r portů input a s portů output.
4. Komponenta je charakterizována událostmi dvou typů: „naplnění portu $\mathbf{p_{out}}$“ interakční hodnotou h_1, tedy $h_1 \epsilon V$, $h_1 \rightarrow p_{out}$ „naplnění portu p_{in}“ interakční hodnotou h_2, tedy
   $h_2 \epsilon V$, $h_1 \rightarrow p_{in}$
   V je univerzum interakčních hodnot.

Na základě uvedených vlastností komponenty se může se uvést její formální definice, která bude vhodným východiskem pro definici asociovaného interface automatu.

Definice 9.

Za komponentu C	považujeme uspořádanou sedmici $\mathbf{C} = (\mathbf{S}, \mathbf{P}, \mathbf{E}, \mathscr{C}, \beta, \mathbf{S_0}, \mathbf{S_k})$, kde
$\mathbf{S}$	je množina stavů, $\mathbf{S_0}$ je počáteční stav, s je průběžný stav a $\mathbf{S_k}$ je koncový stav
$\mathbf{P}$	je konečná množina portů $\mathbf{P} = \mathbf{P_{in}} \cup \mathbf{P_{out}}, \mathbf{P_{in}} \cap \mathbf{P_{out}} = \emptyset$,
$\mathbf{E}$	je konečná množina událostí,
$\mathscr{C}$	je funkční kód komponenty $\mathbf{C}$,
$\beta : \mathbf{P} \leftarrow \mathbf{2^v}$	je mapovací funkce přiřazující portům hodnoty z vybraného univerza $\mathbf{V}$.

Komponenta je součástí komponentového systému a je vybavena porty pro uložení interakčních hodnot (interface) z vybraného univerza. Pro jednoduchost můžeme za reprezentaci univerza V zvolit množinu operací tříd, které jsou nositeli funkcionality komponent. Komponenta je tedy schopna realizovat svou událost naplněním některého ze svých portů. Jistou konečnou množinou událostí může dojít k naplnění všech portů komponenty, nebo k naplnění všech portů komponent komponentového systému. Je přirozené, že takto definované události mění stav jak komponenty, tak i komponentového systému. Porty s hodnotami buď přibývají, nebo ubývají anebo jsou hodnoty daných portů nezadané. V tomto smyslu je komponenta stavovou entitou a rovněž komponentový systém je stavovým systémem. Na tuto kvalitu se nesmí zapomínat u reprezentačních interface automatů a systémů interface automatů.

Komponenty mají následující vlastnosti:

- Každá komponenta C má svůj počet $\mathbf{r} = \mathbf{dim}(\mathbf{P^C_{in}})$ portů typu input a $\mathbf{s} = \mathbf{dim}(\mathbf{P^C_{out}})$ portů typu output.
- Každá komponenta má svůj stav, daný naplněním jejích portů hodnotami z univerza V. Komponenta může být ve stavu počátečním $\mathbf{S_O}$, průběžném s, nebo koncovém $\mathbf{S_k}$. Jestliže komponenta má všechny porty prázdné, je ve stavu počátečním S_O =(0,0). Má-li naplněny všechny porty hodnotami, je ve stavu konečném $S_k = (r, s)$. Není-li ani v jednom z předchozích stavů, je ve stavu průběžném. Hodnota stavu může být tedy prezentována dvěma celými čísly (u,v), která říkají, kolik input a output portů je naplněno hodnotami, $u \leq r$, $v \leq s$.
- Každá komponenta C má specifické hodnoty ve svých portech $\mathbf{hod}(\mathbf{p^C_{in}}), \mathbf{hod}(\mathbf{p^C_{out}})$.
- U žádné komponenty $\mathbf{C}$ by nemělo platit $\exists(\mathbf{p^C_{in}}, \mathbf{p^C_{out}}) | \mathbf{hod}(\mathbf{p^C_{in}}) = \mathbf{hod}(\mathbf{p^C_{out}})$. Tato rekurzivita v komponentě C je pro propojení komponent nežádoucí.
- Dvě komponenty $\mathbf{C_1}$ a $\mathbf{C_2}$ jsou propojeny na základě hodnot v jejich portech, jestliže platí: $\mathbf{hod}(\mathbf{p^C_{2in}}) \subseteq \mathbf{hod}(\mathbf{p^C_{1out}})$ nebo $\mathbf{hod}(\mathbf{p^C_{1in}}) \subseteq \mathbf{hod}(\mathbf{p^C_{2out}})$.

8.2 Formální definice interface automatu

Dále chci ukázat, že speciální interface automaty (patřící do Teorie vyčíslitelnosti), orientované na práci s interface komponent dávají velmi široké možnosti pro řešení všech zmíněných úloh nad komponentami a komponentovými systémy (viz podkapitola 2.10).

Následující část se bude soustředit na volbu dostatečně účinného interface automatu a návrh framework-u zabezpečujícího práci s interface automaty a jejich systémy.

Interface automat (interface automaton) je asociován s jednou komponentou a všechny automaty pro jednotlivé komponenty komponentového sytému vytváří systém interface automatů. Hlavním vodítkem pro definici interface automatů jsou společné a odlišné vlastnosti komponent. Pro definici interface automatu využijeme především to, že komponenta má stavy, má porty a je řízena událostmi.

Na interface automat by měly být kladeny následující požadavky:

- Měl by dostatečně reprezentovat syntax a sémantiku komponenty.
- Měl by dostatečně reprezentovat propojení a složení komponent.
- Měl by sledovat chování komponenty na základě prováděných událostí.
- Měl by umožnovat verifikaci propojení komponent.
- Měl by být schopen realizovat transformace známých operací nad komponentami a komponentovými systémy, které byly uvedeny v předchozím textu.
- Měl by umět realizovat některé speciální operace, např.
  - Comp $(M, N) \rightarrow Q$, což je operace kompozice (composition – složení) dvou automatů do jednoho výsledného.
  - Operace Decomp $(Q) \rightarrow (M, N)$ je dekompozicí (decomposition – rozkladem) automatu Q na dva automaty M, N.

Definice 10.

Buď **C** komponenta systému **CS**. Interface automat asociovaný s touto komponentou, je uspořádaná sedmice $\mathbf{A} = (\mathbf{Q}, \mathbf{P}, \mathbf{E}, \not\subset, \delta, \mathbf{q_0}, \mathbf{F})$, kde

Q	je množina stavů, $\mathbf{q_0} = (\mathbf{u_0}, \mathbf{v_0})$ je počáteční stav, $\mathbf{q} = (\mathbf{u}, \mathbf{v})$ je průběžný stav a $\mathbf{F} = \{\mathbf{u_k}, \mathbf{v_k}\}$ je množina koncových stavů, $\mathbf{F} \subseteq \mathbf{Q}$.
P	$\mathbf{P} = \mathbf{P_{in}} \cup \mathbf{P_{out}}$, $\mathbf{P_{in}} \cap \mathbf{P_{out}} = \emptyset$ je konečná množina portů a její členění
E	je konečná množina událostí, $\mathbf{E} = \mathbf{E_{in}} \cup \mathbf{E_{out}}$, kde $\mathbf{E_{in}}$ a $\mathbf{E_{out}}$ jsou disjunktní a definovány následovně: $\mathbf{E_{in}} = \{(\mathbf{e}, \mathbf{h_1}) \vert \mathbf{e} \epsilon \mathbf{E_{in}}, \mathbf{h_1} \epsilon \beta(\mathbf{e})\} \cap \mathbf{h1} \rightarrow \mathbf{p_{in}}$ $\mathbf{E_{out}} = \{(\mathbf{e}, \mathbf{h_2}) \vert \mathbf{e} \epsilon \mathbf{E_{out}}, \mathbf{h_2} \epsilon \beta(\mathbf{e})\} \cap \mathbf{h_2} \rightarrow \mathbf{p_{out}}$, kde $\mathbf{h_1} \rightarrow \mathbf{p_{in}}$ a $\mathbf{h_2} \rightarrow \mathbf{p_{out}}$ jsou vstupní a výstupní operace pro porty $\mathbf{p_{in}}$ a $\mathbf{p_{out}}$. $\beta : \mathbf{P} \leftarrow \mathbf{2^V}$ je zobrazení pro přiřazení interakčních hodnot portům z univerza **V**,
$\not\subset$	funkční kód asociované komponenty,
$\delta(q, e, q')$	je množina transakčních kroků tzv. transakční funkce, které automat provádí na základě výskytu události $\mathbf{e} \epsilon \mathbf{E}$ ve stavu $\mathbf{q} \epsilon \mathbf{Q}$ a vyrábí nový stav $\mathbf{q'} \epsilon \mathbf{Q}$, tedy: $\delta \subseteq (\mathbf{Q} \times \mathbf{E} \times \mathbf{Q})$.

Poznámka:

1. Nechť (u, v) jsou počty aktuálně naplněných portů P_{in}, P_{out} automatu A. Dvojice (u, v) představuje stav, v němž se automat A nachází.
2. Množiny $\mathbf{P_{in}}$ a $\mathbf{P_{out}}$ nemají žádný společný port.
3. Množina portů $\mathbf{P_{in}}$ se naplňuje pomocí událostí z množiny $\mathbf{E_{in}}$, množina portů $\mathbf{E_{out}}$ se naplňuje pomocí událostí z množiny $\mathbf{E_{out}}$.
4. Události v $\mathbf{E_{in}}$ a v $\mathbf{E_{out}}$ nemají žádnou společnou událost.
5. Na vstupu automatu A se řetězce událostí zapisují ve tvaru (**p:in:h**) nebo (**p:out:h**), kde p je jméno portu, h jeho hodnota a in, out jsou typy portů.
6. Podstatou asociace komponenty C a jejího reprezentačního modelu, tj. s ní asociovaného interface automatu IA (C), jsou především:
   - Stavy Q.
   - Množina E událostí a s událostmi asociovaných operací nad porty automatu.
   - Množina P portů.
   - Přiřazení hodnot portům pomocí zobrazení β : P $\leftarrow 2^V$.

Jestliže základní vlastností komponentového systému je propojení komponent na základě interface, tak role této vlastnosti musí být stejně závažná rovněž v systému interface automatů.

Záznam prvků konkrétního interface automatu A nazveme deskripce automatu A, a označíme des(A).

Deskripce musí zachytit co se vlastně s dílčími interface automaty v systému interface automatů děje.

Taková deskripce potom spočívá v postupném zachycení:

- stavů q_0, q_1,..., q_n, kterými automat A prošel,
- událostí (e_1,..., e_n) a asociovaných operací, které byly provedeny,
- portů a jejich hodnot hod(p_{in}, 1), ..., hod(p_{in}, r) hod(p_{out},1), ..., hod(p_{out}, s).

Následující dva příklady ilustrují grafickou podobu některých pojmů interface automatů:

- transakční kroky δ
- převod systému komponent na systém interface automatů.

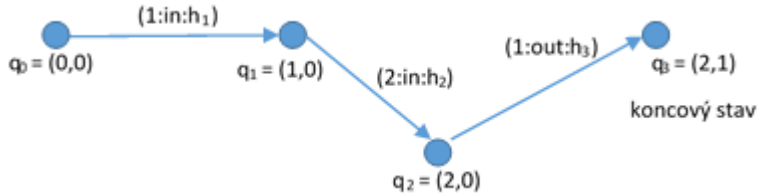
Příklad 4:

Buď dán interface automat $\mathbf{A} = (\mathbf{Q}, \mathbf{P}, \mathbf{E}, \mathscr{O}, \delta, \mathbf{q_0}, \mathbf{F})$ o dvou portech typu input a jednoho portu typu output. Počáteční stav automatu je (0,0). Ukázkově je nakreslen jeho graf pro transakční kroky, provedené na základě zpracování vstupních řetězců $1 : in : h_1$, $2 : in : h_2$, $1 : out : h_3$, které naplní porty automatu zadanými hodnotami.

Řešení:

- Grafická ilustrace interface automatu a legenda pro jeho input a output porty.

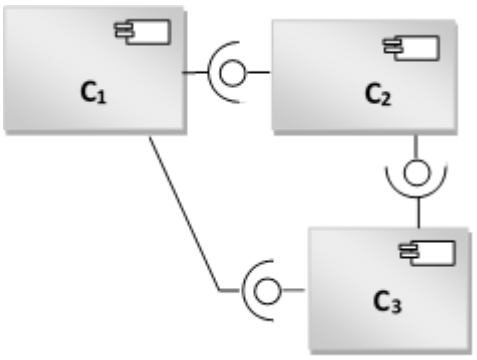
- Graf pro transakční kroky automatu A:



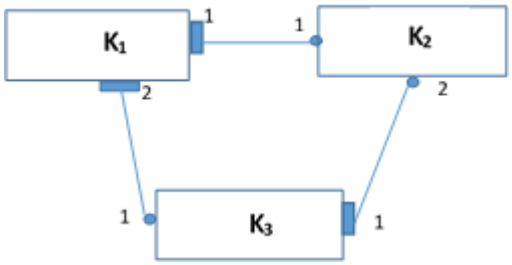
Poznámka

- Řetězce vstupních hodnot (1:in:h_1), (2:in:h_2), (1:out:h_3) patří do regulárního jazyka automatu A.
- Stavy q_0, q_1, q_2, q_3 jsou formovány na základě zpracování vstupních řetězců.

Příklad 5:
Nechť jsou dány tři komponenty C_1, C_2 a C_3 v notaci UML, tvořící komponentový systém (viz obr. 24). Úkolem je nakreslit pomocí interface automatů K_1, K_2 a K_3 grafický model komponentového systému (viz obr. 25).



Obrázek 24: Komponentový systém



Obrázek 25: Grafický model komponentového systému v notaci interface automatů

8.3 Nástroje pro vývoj interface automatů

Interface automaty budeme považovat za modely komponent, pomocí kterých namodelujeme rovněž komponentové systémy. Potřebujeme pouze podpůrný, vývojový modelovací nástroj typu Framework. Za základ zmíněného vývojového nástroje pro systémy interface automatů navrhnu formální jazyk Ψ, na jehož platformě vytvoříme nejen reálné interface automaty a operace s nimi, ale i systémy interface automatů a operace v nich.

Označení příkazu	Sémantika příkazu	Notace příkazu	Poznámky
Ψ_1	Zřízení prázdného automatu, P=ϕ, E=ϕ, stav je S_0	Ψ_1 (K) Ψ_1 (K_1, K_2)	Přidá se buď jen jeden, nebo více automatů do I-systému
Ψ_2	Zřídí se **i** portů input a **j** portů output	Ψ_2 (K:**i**:in) Ψ_2 (K:**j**:out)	Porty jsou označeny svým pořadím: 1, 2, ….
Ψ_3	Odebrání portů	Ψ_3 (K:1:in, 1:out)	Odebere se první port input a první port output
Ψ_4	Naplnění hodnoty portu na základě události **e**	Ψ_4 (K:1:in:h_1) Ψ_4 (K:3:out:h_2)	Porty automatu K se naplní danými hodnotami
Ψ_5	Odstranění hodnoty portu	Ψ_5 (K, 3:in) Ψ_5 (K, 1:out)	Portům se odstraní hodnota
Ψ_6	Odstranění rekurzivity	Ψ_6	Funguje pro celý systém I-automatů
Ψ_7	Ψ7 Propojení dvou automatů $\overline{\Psi_7}$ rozpojení dvou automatů	Ψ_7 (K_1:1:in, K_2:1:out) $\overline{\Psi_7}$ (K_1:1:in, K_2:1:out)	Automaty K_1 a K_2 budou propojeny / rozpojeny na základě portů K_1:1:in, K_2:1:out
Ψ_8	Verifikace orchestrace	Ψ_8	Funguje pro celý systém I-automatů
Ψ_9	Nalezení funkcionálních řetězců	Ψ_9	-,,-
Ψ_{10}	Vytížení automatů	Ψ_{10} Ψ_{10} (K_1, K_2, …, K_n)	Aplikuje se buď na celý systém, nebo jen na vybrané automaty
Ψ_{11}	Přidání/Odstranění automatu ze systému	Ψ_{11} (K) $\overline{\Psi_{11}}$ (K)	Daný automat K se odstraní včetně všech jeho vazeb
Ψ_{12}	Převod komponentového systému z notace UML na systém I-automatů	Ψ_{12}	Daný systém komponent se převede na systém I-automatů
Ψ_{13}	Převod I-systému automatů na komponentový systém v notaci UML	(deskripce systému I-automatů)	Funguje pro celý systém I-automatů
Ψ_{14}	Kompozice automatů K_1, K_2 na automat K, tj. Comp (K_1, K_2) → K	Ψ_{14} (K_1, K_2, K)	Kompozice se aplikuje na dva automaty K_1, K_2
Ψ_{15}	Dekompozice automatu K na K_1, K_2, tj. Decomp (K) → (K_1, M_2)	Ψ_{15} (K, K_1, K_2)	Dekompozice se aplikuje na automat K

Obrázek 26: Formální notace dílčích vývojových příkazů formálního jazyka Ψ

Vývojový nástroj rovněž umožní konstruovat protokol des(A), tj. deskripce všech operací, které provede. Konstruovaný protokol je materiálem, který bude pro interface automaty používán jako jejich digitální reprezentace.

Formální notace dílčích vývojových příkazů formálního jazyka Ψ jsou zavedeny předcházející tabulkou (viz obr. 26).

Poznámka:

- Příkazy Ψ_i, i=1, 2, ..., 15 jsou dvou typů. První skupina nepotřebuje ke svému provedení jiné pomocné příkazy. Takovými jsou $\Psi_{1,2,3,4,5,7}$. Na rozdíl od nich příkazy $\Psi_{6,8,9,10,11,12,13,14,15}$ potřebují ke svému provedení jiné příkazy a působí jako makropříkazy. Např. makro Ψ_{12} potřebuje ke své realizaci příkazy $\{\Psi_1, \Psi_2, \Psi_4, \Psi_7\}$.
- Předchozí tabulkou jsem navrhl vývojový nástroj D pro vývoj systémů interface automatů založený na formálním jazyku Ψ. Pro funkcionalitu tohoto systému bude nutné navrhnout a vytvořit specifikovaný podpůrný D-Framework, který realizaci příkazů jazyka Ψ zabezpečí. Tento úkol je diskutován v následující podkapitole.
- Množina příkazů jazyka Ψ je účelově rozšiřitelná podle potřeby frameworku.

8.4 Design pro D-Framework

Nejdříve stručně pojednám o obecném pojetí frameworku a potom připomenu dvě známé typy implementací.

Definice 11. *Buď D vývojový systém obecné množiny N informačních struktur založený na jazyku Ψ. D-Framework se potom chápe jako podpůrná infastruktura prostředků pro uplatnění standardů a konvencí vývojového systému D ve vývoji informačních struktur množiny N. Čtveřice (D, Ψ, N, D-Framework) reprezentuje plně zabezpečený produkční systém prvků množiny N.*

S uplatněním *D-Frameworku*, tj. CSF (Component support Framework) pro vývoj komponentových systémů, se v odborné literatuře nesetkáváme často.

První implementace, v níž v roli D působí komponentový model a v roli N zase komponentové systémy, se vyskytuje v publikaci (Crnkovic, Larson, 2002). Potvrzením této skutečnosti je citace z uvedené publikace:

Komponentový model definuje množinu standardů a konvencí používaných vývojářem, zatímco komponentový framework podporuje infrastrukturu pro komponentový model

Druhá implementace D-Frameworku, kterou uvádím, je implementace, kde vývojový nástroj D pro systémy interface automatů je založen na formálním jazyku Ψ. Za množinu N je považována množina systémů interface automatů. Na základě druhé implementace, může být definována velmi schopná báze informačního modelování interface automatů. Obě implementace se dají ilustrovat následujícím obrázkem (viz obr. 27).

Obrázek 27: Implementace D-Frameworku

Pohledy na vývoj a implementaci podpůrného D-frameworku se v odborné literatuře značně liší. Zajímavé jsou následující dva pohledy uvedené v (Crnkovic, Larson, 2002):

1. *D-Framework* je realizován jako množina vzájemně souvisejících objektových tříd s dědičností, agregací a kompozicí, jejichž instance jsou využívány pro zabezpečení standardů a konvencí vývojového systému D. Takto je definován framework pro systém ADO (Access Data Object) pro řízení báze dat u firmy Microsoft.
2. *D-Framework* je realizován jako množina předdefinovaných, vzájemně souvisejících komponent pro zabezpečení standardů a konvencí vývojového systému D.

Konstrukce D-frameworku pro vývojový nástroj D systémů interface automatů byla součástí výzkumného projektu IGA (PEF_DP_2015_012) na Mendelově univerzitě v Brně, Provozně ekonomické fakultě, Ústavu informatiky, jehož jsem řešitelem.

8.5 Využití formálního jazyka Ψ pro vývoj interface automatů

Příklady, které uvádím, nejsou obtížné, ale názorné. Začne se převodem komponentového systému v notacích UML na systém interface automatů (Příklad 6), pokračuje se příklady vybraných operací. V těchto příkladech jsou důležité programy v jazyku Ψ a protokoly deskripce systémů interface automatů, které se buď konstruují, nebo modifikují.

Příklad 6:
Následující makro Ψ_{12} způsobí vznik systému interface automatů, který je transformací uvedeného Komponentového diagramu v notacích UML. Tento komponentový diagram náleží doméně Správa soukromých zbraní v ČR. Současně se

vytváří protokol des(K_1, K_2, K_3) deskripce jednotlivých interface automatů a protokol deskripce celého systému interface automatů.

Vývojový program pro Ψ_{12}:

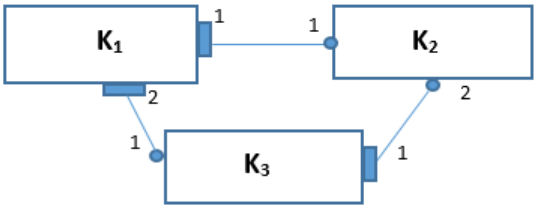
Ψ_1 (K_1, K_2, K_3)

Ψ_2 (K_1:2:in)
Ψ_2 (K_2:2:out)
Ψ_2 (K_3:1:in, K_3:1:out)

Ψ_4 (K_1:1:in:h_1)
Ψ_4 (K_2:1:out:h_1)
Ψ_4 (K_2:2:out:h_2)
Ψ_4 (K_3:1:in:h_2)
Ψ_4 (K_1:2:in:h_3)
Ψ_4 (K_3:1:out:h_3)

Ψ_7 (K_1:1:in, K_2:1:out)
Ψ_7 (K_2:2:out, K_3:1:in)
Ψ_7 (K_1:2:in, K_3:1:out)

Vzniklý systém I-automatů



Pro interface automaty K_1, K_2, a K_3 se vytvoří následující protokol výsledné deskripce:

Jméno	stav	naplněný port	prázdný port	propojení
K_1	2/0	1:in:h_1	-	K_1:1:in, K_2:1:out
		2:in:h_3	-	K_1:2:in, K_3:1:out
K_2	0/2	1:out:h_1	-	K_1:1:in, K_2:1:out
		2:out:h_2	-	K_2:2:out, K_3:1:in
K_3	1/1	1:in:h_2	-	K_2:2:out, K_3:1:in
		1:out:h_3	-	K_1:2:in, K_3:1:out

Příklad 7:

V tomto příkladu ukáži provedení operace $\overline{\Psi_{11}}$ pro odstranění interface automatu K_3. Odstranění tohoto automatu značí postupné odstranění deskripce automatu a souvislostí se zbývajícími automaty:

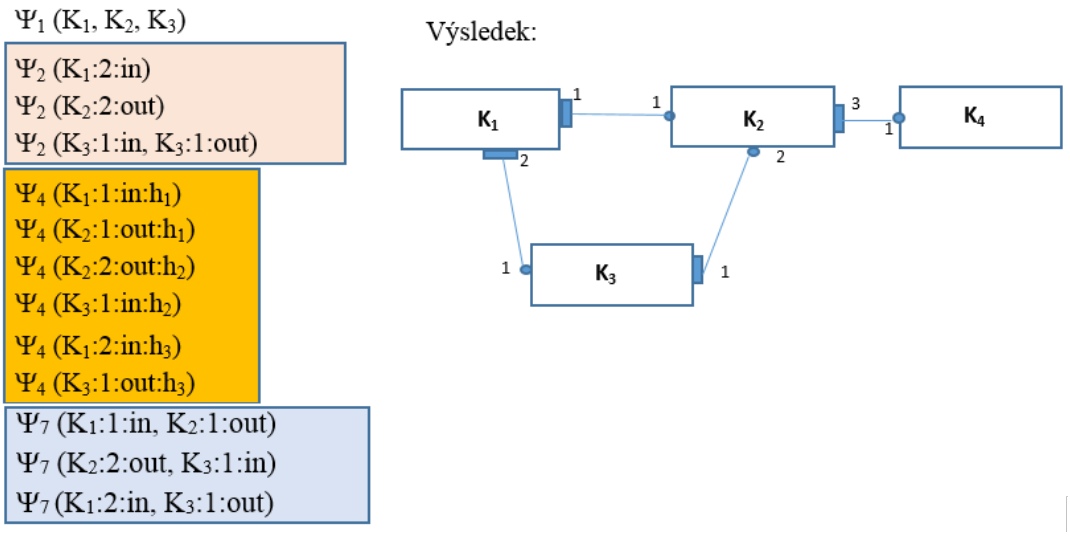
- Odstranění portů automatu K3.
- Pro K1 se odstraní port 2:in.
  Pro K2 se odstraní port 2:out.
- Odstraní se propojení automatu K3 na automaty K1, K2.

Dodatek:
Ψ_3 (K3:1:in) Ψ_3 (K3:1:out)
Ψ_3 (K1:2:in)
Ψ_3 (K2:2:out)
$\overline{\Psi_7}$ ((K1:2:in, K3:1:out)
$\overline{\Psi_7}$ (K2:2:out, K3:1:in)

Protokol deskripce automatů K_1 a K_2:

K_1	1/0	1:in:h1	-	K1:1:in, K2:1:out
K_2	0/1	1:out:h1	K1:1:in, K2:1:out	

Příklad 8:
V tomto příkladu je ukázáno provedení operace přidání automatu K_4 a jeho propojení s automatem K_2.



Ψ_1(K4)
Ψ_2 (K2:3:in)
Ψ_2 (K4:1:out)

Ψ_4 (K4:1:out:h4)
Ψ_4 (K2:3:out:h4)
Ψ_7 (K2:3:in, K4:1:out)

Provádění příkazů Ψ_1 až Ψ_{15} umožní výstavbu realizačního Frameworku, který by mohl být uložen jak v operačním systému počítače, tak i ve vybraném objektovém programovacím jazyku.

Příklad 9:

V tomto příkladu je proveden překlad komponentového diagramu pro doménu Personalistika (viz Obrázek 1) a vytvořen protokol deskripce interface automatů. (Názvy jednotlivých komponent byly z důvodu přehlednosti zkráceny)

Ψ_1(PersonalSam, ZpracovaniSluzCest, AbsenceZam, ZpracMezd, EaZDochazky)
Ψ_1(VyberovaRizeni, PersonalEvidZam, PlanPracMist)

Ψ_2 (PersonalSam:1:in, PersonalSam:1:out)
Ψ_2 (ZpracovaniSluzCest:1:in, ZpracovaniSluzCest:2:out)
Ψ_2 (AbsenceZam:1:in)
Ψ_2 (ZpracMezd:3:in, ZpracMezd:1:out)
Ψ_2 (EaZDochazky:5:out)
Ψ_2 (VyberovaRizeni:2:in, VyberovaRizeni:1:out)
Ψ_2 (PersonalEvidZam:6:in, PersonalEvidZam:2:out)
Ψ_2 (PlanPracMist:2:out)

Ψ_4 (PersonalSam:1:in:h1), Ψ_4(PersonalEvidZam:1:out:h1)
Ψ_4 (PersonalSam:1:out:h2), Ψ_4(PersonalEvidZam:1:in:h2)
Ψ_4 (ZpracovaniSluzCest:1:out:h3), Ψ_4(PersonalEvidZam:2:in:h3)
Ψ_4 (ZpracovaniSluzCest:2:out:h4), Ψ_4(ZpracMezd:1:in:h4)
Ψ_4 (ZpracovaniSluzCest:1:in:h5), Ψ_4(EaZDochazky:1:in:h5)
Ψ_4 (AbsenceZam:1:in:h6), Ψ_4(EaZDochazky:5:out:h6)
Ψ_4 (ZpracMezd:1:out:h7), Ψ_4(PersonalEvidZam:3:in:h7)
Ψ_4 (ZpracMezd:2:in:h8), Ψ_4(PersonalEvidZam:2:out:h8)
Ψ_4 (ZpracMezd:3:in:h9), Ψ_4(EaZDochazky:2:out:h9)
Ψ_4 (EaZDochazky:3:out:h10), Ψ_4(PersonalEvidZam:4:in:h10)
Ψ_4 (EaZDochazky:4:out:h11), Ψ_4(VyberovaRizeni:1:in:h11)
Ψ_4 (VyberovaRizeni:1:out:h12), Ψ_4(PersonalEvidZam:5:in:h12)
Ψ_4 (VyberovaRizeni:2:in:h13), Ψ_4(PlanPracMist:1:out:h13)
Ψ_4 (PlanPracMist:2:out:h14), Ψ_4(PersonalEvidZam:6:in:h14)

Následuje sada příkazů, která propojí jednotlivé automaty mezi sebou.

Ψ_7 (PersonalSam:1:in, PersonalEvidZam:1:out)
Ψ_7 (PersonalSam:1:out, PersonalEvidZam:1:in)
Ψ_7 (ZpracovaniSluzCest:1:out, PersonalEvidZam:2:in)
Ψ_7 (ZpracovaniSluzCest:2:out, ZpracMezd:1:in)
Ψ_7 (ZpracovaniSluzCest:1:in, EaZDochazky:1:in)
Ψ_7 (AbsenceZam:1:in, EaZDochazky:5:out)
Ψ_7 (ZpracMezd:1:out, PersonalEvidZam:3:in)
Ψ_7 (ZpracMezd:2:in, PersonalEvidZam:2:out)
Ψ_7 (ZpracMezd:3:in, EaZDochazky:2:out)
Ψ_7 (EaZDochazky:3:out, PersonalEvidZam:4:in)
Ψ_7 (EaZDochazky:4:out, VyberovaRizeni:1:in)
Ψ_7 (VyberovaRizeni:1:out, PersonalEvidZam:5:in)
Ψ_7 (VyberovaRizeni:2:in, PlanPracMist:1:out)
Ψ_7 (PlanPracMist:2:out, PersonalEvidZam:6:in)

Následuje protokol deskripce interface automatů pro doménu Personalistika:

Jméno	**stav**	**naplněný port**	**prázdný port**	**propojení**
PersonalSam	1/1	1:in:h_1	-	PersonalSam:1:in, PersonalEvidZam:1:out
		1:out:h_2	-	PersonalSam:1:out, PersonalEvidZam:1:in
ZpracovaniSluzCest	1/2	1:in:h_5	-	ZpracovaniSluzCest:1:in, EaZDochazky:1:in
		1:out:h_3	-	ZpracovaniSluzCest:1:out, PersonalEvidZam:2:in
		2:out:h_4	-	ZpracovaniSluzCest:2:out, ZpracMezd:1:in
AbsenceZam	1/0	1:in:h_6	-	AbsenceZam:1:in, EaZDochazky:1:out
ZpracMezd	3/1	1:in:h_8	-	ZpracMezd:2:in, PersonalEvidZam:2:out
		1:in:h_4	-	ZpracovaniSluzCest:2:out, ZpracMezd:1:in
		2:in:h_9	-	ZpracMezd:3:in, EaZDochazky:2:out
		2:out:h_7	-	ZpracMezd:1:out, PersonalEvidZam:3:in
EaZDochazky	0/5	3:out:h_{10}	-	EaZDochazky:3:out, PersonalEvidZam:4:in
		4:out:h_{11}	-	EaZDochazky:4:out, VyberovaRizeni:1:in
		2:out:h_9	-	ZpracMezd:3:in, EaZDochazky:2:out
		5:out:h_6	-	AbsenceZam:1:in, EaZDochazky:5:out
		1:out:h_5	-	ZpracovaniSluzCest:1:in, EaZDochazky:1:in
VyberovaRizeni	2/1	1:in:h_{11}	-	EaZDochazky:4:out, VyberovaRizeni:1:in
		2:in:h_{13}	-	VyberovaRizeni:2:in, PlanPracMist:1:out
		1:out:h_{12}	-	VyberovaRizeni:1:out, PersonalEvidZam:5:in
PersonalEvidZam	6/2	1:in:h_2	-	PersonalSam:1:out, PersonalEvidZam:1:in
		2:in:h_3	-	ZpracovaniSluzCest:1:out, PersonalEvidZam:2:in
		3:in:h_7	-	ZpracMezd:1:out, PersonalEvidZam:3:in
		4:in:h_{10}	-	EaZDochazky:3:out, PersonalEvidZam:4:in
		2:in:h_{12}	-	VyberovaRizeni:1:out, PersonalEvidZam:5:in
		2:in:h_{14}	-	PlanPracMist:2:out, PersonalEvidZam:6:in
		1:out:h_1	-	PersonalSam:1:out, PersonalEvidZam:1:in
		2:out:h_8	-	ZpracMezd:2:in, PersonalEvidZam:2:out
PlanPracMist	0/2	1:out:h_{13}	-	VyberovaRizeni:2:in, PlanPracMist:1:out
		2:out:h_{14}	-	PlanPracMist:2:out, PersonalEvidZam:6:in

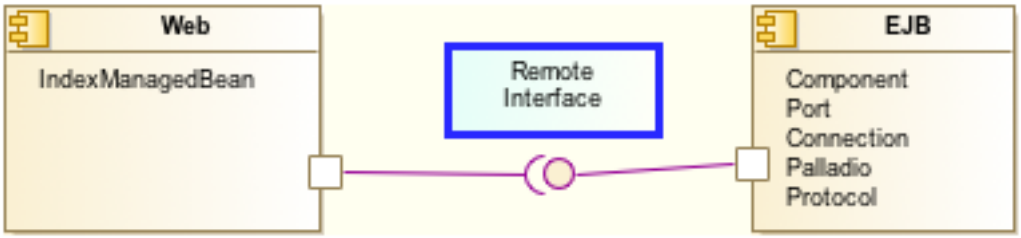
Na základě prohledávání lineárního protokolu lze snadno realizovat operace nad komponentovými systémy uvedené v 2.10 a 7.1.

8.6 Implementace D-Frameworku v jazyce Java EE

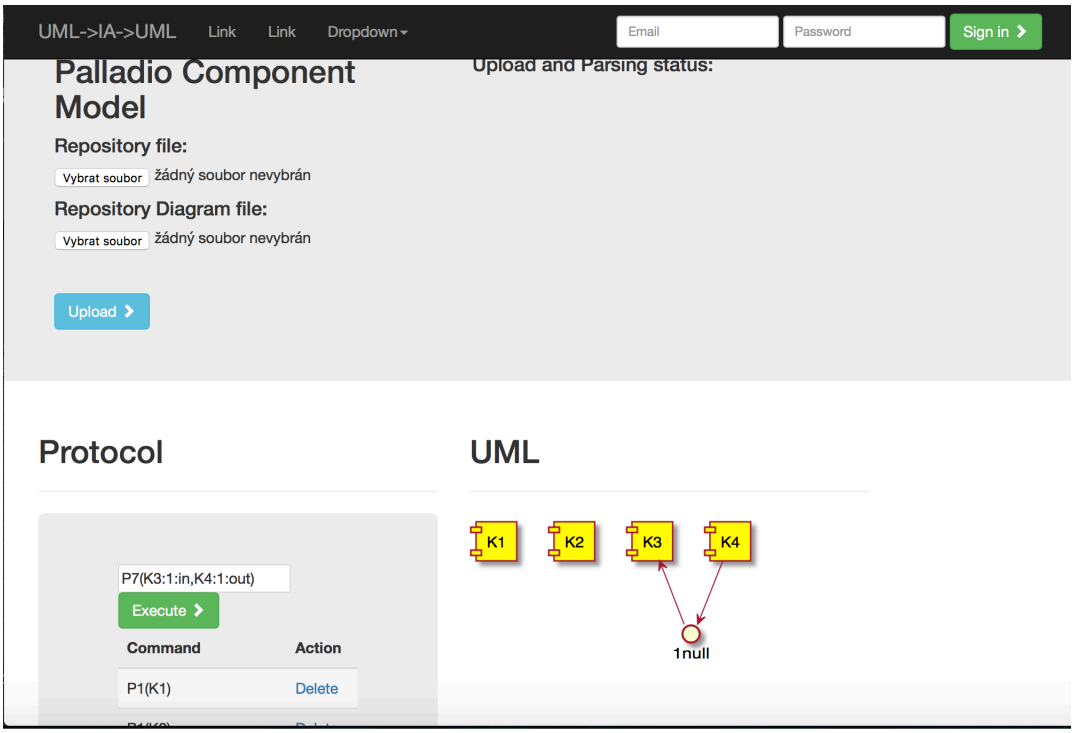
Z důvodu podpory co nejvíce platforem, bylo rozhodnuto, že se bude jednat o webovou aplikaci. Aplikace byla implementována v jazyce Java Enterprise Edition za použití specifikace EJB3 (Enterprise Java Beans 3). Aplikace se skládá ze ze tří komponent (viz obrázek 28). První komponenta (EJB) implementuje business logiku. Nachází se zde třída Protocol, která implementuje interpretaci vstupních příkazů v jazyce Ψ. Třída Protokol sestavuje vnitřní reprezentaci modelu komponentového systému pomocí tříd Component, která představuje komponentu. Třída komponent má mimo jiné atribut pole portů (třída Port), přes které je možné realizovat propojení. Dále třída Connection, která propojuje komponenty na základě jejich portů. Aplikace dále umožňuje importování modelu komponentového systému z verifikačního nástroje Palladio. Tyto soubory jsou založeny na jazyce XMI (XML Metadata Interchange), který je standardizován pro výměnu metadat skupinou OMG (Object Management Group). Pokud si uživatel vybere tuto volbu, bude využita třída Palladio, která umožňuje převod do jazyka interface automatů.

EJB komponenta má za účelem podpory distributivity rozhraní (Remote Interface) implementováno vně komponenty. Interface typu Remote umožuje běh komponent na různých serverech.

Poslední část je Web komponenta, která v sobě obsahuje prezentační logiku. Zajišťuje vykreslení stránky a ošetření jednotlivých jednotlivých akcí (např. odeslání příkazu přes formulář), které umožňují interakci uživatele s aplikací (viz obrázek 29).



Obrázek 28: Komponetový diagram D-Frameworku



Obrázek 29: Webové rozhraní D-frameworku

9 Využití interface automata v modelování kontraktů v System of Systems

Pojem System of sytems (SoS) označuje kolekci systémů propojenou mezi sebou za účelem vzniku nové funkcionality, kterou není schopen žádný z těchto systémů samostatně poskytnout. Každý podsystém (constituent system – CS), který je součástí SoS, je do jisté míry nezávislý, a to tak, že má svůj vlastní management, cíle a zdroje, které mohou být koordinovány k dosažení globálního cíle daného SoS. Tato nezávislost podsystémů, které jsou uvnitř SoS, vytváří prostor pro verifikaci globálně vzniklé funkcionality.

Existuje mnoho architektonických vzorů popisujících SoS architekturu (Perry, Holt, Payne, Bryans a kol., 2014). Jeden z nejvýznamnějších je tzv. Contract pattern, který specifikuje rozhraní podsystému (constituent systems). Kontrakt definuje chování podsystému pomocí operací, hodnot, preconditions, postoconditions a invariantů. Popropojení takových kontraktů může být použito k verifikaci chování SoS jako celku.

Contract pattern využívá jazyk SysML a COMPASS Modelling Language (CML) (Bryans, Payne, Holt, Perry, 2013). SoS architektonická struktura v Contract patternu je definována pomocí SysML a doplněna kontraktově specifickými výrazy v CML. CML může být použito ke specifikaci preconditions, postconditions a invariantů kontraktu (Bryans, Fitzgerald,Payne, Kristensen, 2014). SoS popsaný pomocí SysML, doplněné o definice v CML, může být kompletně převeden do CML. SoS v takovéto podobě umožňuje zpracování nástrojů k analýze SoS, jako je například model checking a theorem proving (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014).

Jak je výše uvedeno, Contract pattern je založen na kombinaci SysML a CML notace. SysML je rozšířený a umožňuje snadné pochopení pro velké množství inženýrů ze zainteresovaných stran, zatímco CML je zamýšleno pro specialisty SoS inženýry, kteří rozumějí formální notaci. (Ingram, Payne, Fitzgerald, Couto, 2015).

Bohužel využití CML je limitující ze dvou důvodů. První je důvod je, že CML omezuje snadné osvojení Contract patternu komunitami specializujícími se na modelování systémů s využitím SysML. Druhý důvod je, že z důvodu kompletní reprezentace SysML po převodu SoS do CML, je nutné verifikovat kompatibilitu kontraktů, které jsou poskytnuty tvůrcům konstituentních systémů. Současné nástroje neumožňují simulaci a verifikaci kompatibility provádět staticky. (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014).

9.1 Contract pattern

Jednotlivé CS (Constituent system) mohou odpovídat několika kontraktům a každý CS může implementovat jednotlivé kontrakty způsobem, který si vlastník CS zvolí. Contract pattern (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014) je tzv. enabling pattern, který se skládá z několika pohledů (viewpoints) (tabulka 1), které jsou definovány pomocí SysML a CML.

Tabulka 1: Popis jednotlivých pohledů (diagramů) v Contract Pattern

Název	**Účel pohledu**
Contractual SoS Definition Viewpoint (CSDV)	Indentifikuje kontrakty, z nichž se skládá SoS.
Contract Conformance Viewpoint (CCV)	Identifikuje konstituentní systémy, které tvoří daný SoS a označuje kontrakty, kterým konstituentní systém odpovídá. Zahrnuje všechny kontrakty identifikované v *Contractual SoS Definition Viewpoint*.
Contract Connections Viewpoint (CConnV)	Zobrazuje propojení a rozhraní mezi kontrakty uvnitř SoS. Zahrnuje všechny kontrakty identifikované v *Contractual SoS Definition Viewpoint*.
Contract Definition Viewpoint (CDV)	Definuje operace, stavové proměnné a stavové invarianty pro každý kontrakt identifikovaný v *Contractual SoS Definition Viewpoint*.
Contract Protocol Viewpoint (CPV)	Definuje chování kontraktu identifikovaném v *Contractual SoS Definition Viewpoint* z pohledu uspořádání zpráv a volání operací mezi ostatními CS v SoS.

Jak bylo uvedeno výše, existuje ještě Interface pattern, který je vhodný pro definici dat a interakce mezi CS. Interface pattern je bohužel nedostačující pro modelování interního chování jednotlivých CS. Účel Contract patternu je umožnění specifikace omezení chování, které každý CS musí poskytovat.

Omezení Contract patternu vyplývá z toho, že přesně nedokáže definovat, která operace by měla být viditelná a která by viditelná být neměla. Toto může vést v praxi k tomu, že se všemi operacemi bude zacházeno jako s viditelnými nebo budou chápany jako input operace. Takový přístup jde však proti přístupu slabé vazby,

který říká, že jednotlivé komponenty by měly být mezi sebou minimálně závislé, což omezuje flexibilitu celého systému. Například může být těžké nahradit jeden CS nějakým jiným, pokud není jasně specifikovaný interface těchto CS. Z tohoto důvodu se v praxi Contract pattern doplňuje ještě diagramem Interface Definition View, který je součástí Interface pattern, což značně komplikuje transparentnost a použitelnost Contract pattern.

9.2 OCL jako rozšíření SysML v Contract pattern

Myšlenka využití OCL v SysML je založena na faktu, že SysML je definováno jako UML profil. Toho je dosaženo využitím stereotypů a omezení, použitých na specifické elementy UML modelu.

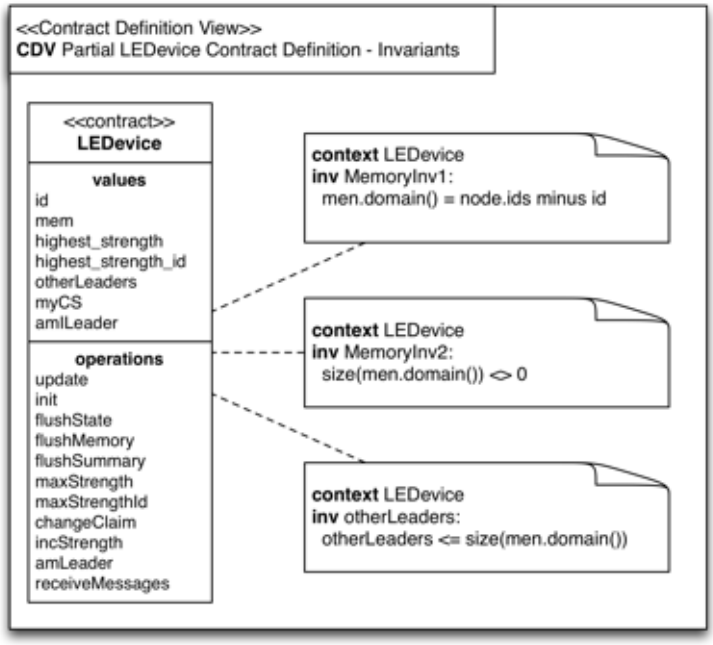
Zdá se, že OCL se zdá použitelné na diagram Contract Definition Viewpoint a Contract Protocol Viewpoint, protože jsou v podstatě podobné diagramu tříd a stavovému diagramu z UML. Zbývající diagramy jako Contractual SoS Definition Viewpoint, Contract Conformance Viewpoint a Contract Connections Viewpoint jsou spíše konceptuálního charakteru a nedosahují požadovaného detailu deskripce, kde by mohlo být OCL použitelné.

Contract Definition Viewpoint

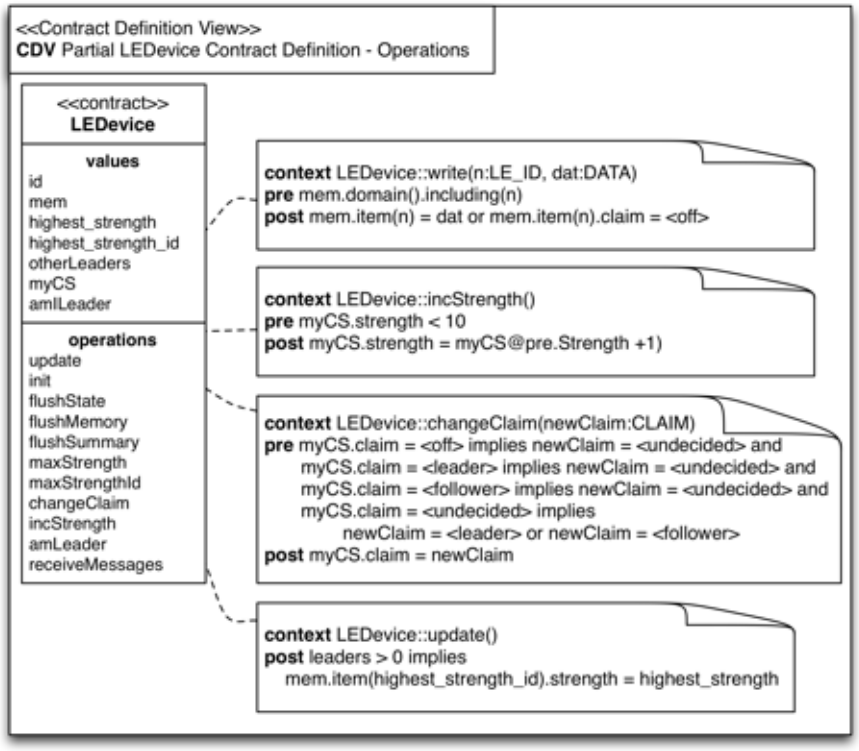
V publikaci (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014) CDV používá CML výrazy ke specifikaci invarianatů, preconditions a postconditions operací kontraktů. Pro zapsaní těchto OCL je využito možnosti SysML poznámek připojených k danému kontraktu.

Z důvodu zpřehlednění bylo provedeno rozdělení kontraktu pro LE Device na dva diagramy, které jsou na obrázcích 30 a 31. První diagram 30 zobrazuje tři invarianty kontraktu pro LE Device. Tyto invarianty jsou definovány odděleně v SysML poznámkách. Každý invariant je definován pomocí OCL. Na obrázku 30, můžeme vidět invarianty, které poskytnutí omezení (constraints), která jsou vztažena na *mem* proměnnou.

Obrázek 31 zobrazuje kontrakt pro LE Device a jeho operace, na které jsou kladena omezení. Je možné si také všimnout, že na obrázku se nachází několik operací, které nemají žádné omezení. Jak bylo uvedeno výše, je využito SysML poznámek pro každé OCL omezení nebo invariant, každý OCL výraz se odkazuje k jedné operaci. OCL statement odkazuje na kontext, v tomto případě na kontrakt, ke kterému se vztahuje (v tomto případě LE Device). Dle obrázku je možné si všimout, že nejvíce OCL výrazů odkazuje na operaci *write*.



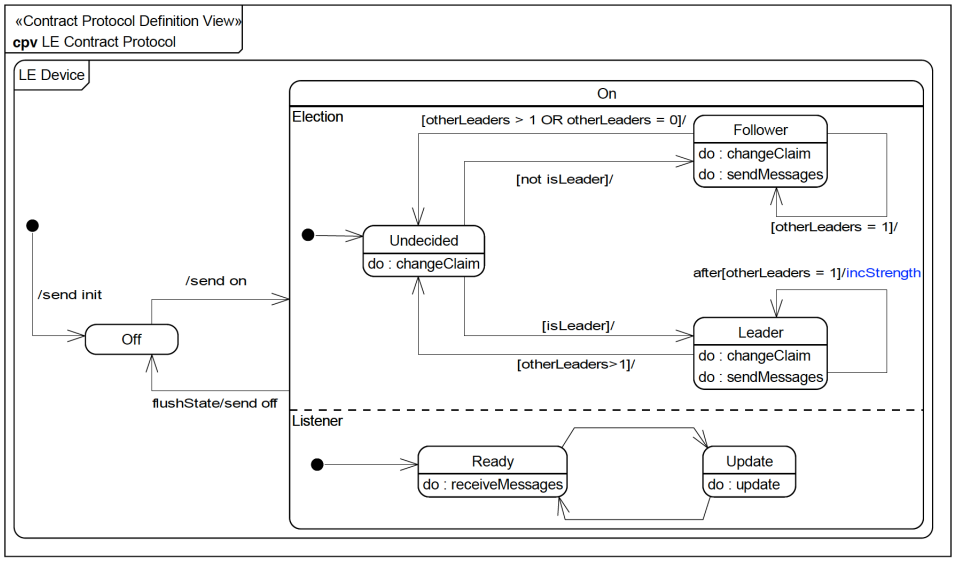
Obrázek 30: Příklad Contract Definition View s OCL invarianty



Obrázek 31: Příklad Contract Definition View s OCL operacemi

Contract Protocol Definition Viewpoint

Dále bylo cíleno na CPV. Původně v tomto pohledu bylo použito CML pro popis omezení (constraints) jednotlivých přechodů. I v tomto případě mohou být tato omezení definována pomocí OCL. Obrázek 32 zachycuje chování LE Device s doplněnými přechody. V tomto příkladu CML a OCL výrazy jsou syntakticky shodné, a proto tento diagram zůstal nezměněn. V této sekci bylo demonstrováno použití OCL v Contract pattern. OCL oproti CML přináší výhodu snadného osvojení SoS komunitou.



Obrázek 32: Příklad Contract Protocol Definition View

9.3 Verifikace kompatibility kontraktů pomocí interface automatů

Pro verifikaci kompatibility kontraktů je prezentována aplikace *interface automatu*, který počítá s viditelností operací. Tyto operace se mohou vykytovat jako vstupní (input), výstupní (output) nebo skryté (hidden). Verifikace pomocí *interface automatu* také umožňuje komplexní pohled na celý SoS na základě protokolů pro komunikaci komponenent (kontraktů). Návrh *interface automatu* pro kontrakt v prezentované případové studii je založen na návrhu *interface automatu* prezentovaného autory Samir Chouali a kol. (Chouali, Mountassir, Mouelhi, 2010). Tito autoři rozšířili původní definici *interface automatu* o pre a post podmínky (conditions), které se také mohou vyskytovat v definici kontraktů. Jejich přístup byl dále rozšířen o definice proměnných, které zachycují různé stavy komponenty na základě volání operací. Verifikace kompatibility dvou kontraktů je zajištěna verifikací jejich odpovídajících rozhraní.

Následující definice prezentuje *interface automat* pro kontrakt.

Definice 12. *(Interface automat pro kontrakt)*
Nechť C je komponenta komponentového systému CS, který reprezentuje kontraktovou specifikaci pro konstituentní systém v SoS. Interface automat asociovaný s touto komponentou je devítice
$A(C) = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, V_A, Pre_A, Post_A, \delta_A \rangle$*, která se skládá z*

- S_A *je množina stavů*
- $I_A \subseteq S_A$ *je množina iniciálních stavů.*
- Σ_A^I, Σ_A^O *a* Σ_A^H *jsou disjunktní množiny vstupních, výstupních a skrytých akcí.*
- V_A *je množina proměnných kontraktu.*
- Pre_A *je množina preconditions pro operace (akce) kontraktu. Preconditions jsou popsány pomocí OCL.*
- $Post_A$ *je množina postconditions pro operace (akce) kontraktu. Preconditions jsou popsány pomocí OCL.*
- δ_A *je množina přechodových stavů, která se skládají z operací (akcí)* $a \in \Sigma$ *a stavů* v $s \in S$*. Přechod je realizován za předpokladu splnění preconditions* v Pre_A *a postconditions* v $Post_A$*, které mohou zahrnovat proměnné z* V_A*.*

Následující definice prezentuje kompozici pro dva *interface automaty*, které reprezentují kontrakty.

Definice 13. *(Kompozice interface automatů reprezentujících kontrakt)*
Nechť $A_1 = \langle S_1, I_1, \Sigma_1^I, \Sigma_1^O, \Sigma_1^H, V_1, Pre_1, Post_1, \delta_1 \rangle$
a $A_2 = \langle S_2, I_2, \Sigma_2^I, \Sigma_2^O, \Sigma_2^H, V_2, Pre_2, Post_2, \delta_2 \rangle$ *jsou dva interface automaty.*
Množina $\Sigma_1 \cup \Sigma_2$ *je značena jako* $Shared(A_1, A_2)$*. Interface automaty* A_1 *a* A_2 *jsou sestavitelné, pokud platí, že*
$(\Sigma_1^I \cap \Sigma_2^I) = (\Sigma_1^O \cap \Sigma_2^O) = (\Sigma_1^H \cap \Sigma_2) = (\Sigma_1 \cap \Sigma_2^H) = \emptyset$.
Pokud dva interface automaty A_1 *a* A_2 *jsou sestavitelné, pak platí* $Shared(A_1, A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_2^I \cap \Sigma_1^O)$.

Následující definice prezentuje synchronizovaný produkt dvou *interface automatů* reprezentujících kontrakty.

Definice 14. *(Synchronizovaný produkt pro dva interface automaty reprezentující kontrakty)*
Nechť $A_1 = \langle S_1, I_1, \Sigma_1^I, \Sigma_1^O, \Sigma_1^H, V_1, Pre_1, Post_1, \delta_1 \rangle$
a $A_2 = \langle S_2, I_2, \Sigma_2^I, \Sigma_2^O, \Sigma_2^H, V_2, Pre_2, Post_2, \delta_2 \rangle$ *jsou dva sestavitelné interface automaty. Produkt* $A_1 \otimes A_2$ *je definován jako* $\langle S_1 \times S_2, I_1 \times I_2, \Sigma^I, \Sigma^O, \Sigma^H, V, Pre, Post, \delta \rangle$ *tak, že:*
Produkt $A_1 \otimes A_2$ *je definován jako* $\langle S_1 \times S_2, I_1 \times I_2, \Sigma^I, \Sigma^O, \Sigma^H, V_1 \cup V_2, Pre, Post, \delta \rangle$ *tak, že:*

- $\Sigma^I = (\Sigma_1^I \cup \Sigma_2^I) \backslash Shared(A_1, A_2)$;

- $\Sigma^O = (\Sigma_1^O \cup \Sigma_2^O) \backslash Shared(A_1, A_2)$;
- $\Sigma^H = \Sigma_1^H \cup \Sigma_2^H \cup Shared(A_1, A_2)$;
- $((s_1, s_2), pre, a, post, (s_1', s_2')) \in \delta$ *pokud*
  - $a \notin Shared(A_1, A_2) \wedge (s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge s_2 = s_2' \wedge pre = pre_1 \wedge post = post_1$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge s_1 = s_1' \wedge pre = pre_2 \wedge post = post_2$
  - $a \in Shared(A_1, A_2) \wedge ((s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge a \in \Sigma^I) \wedge ((s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge a \in \Sigma^O) \wedge pre = (pre_2 \wedge pre_1) \wedge post = (post_1 \wedge post_2)$
  - $a \in Shared(A_1, A_2) \wedge ((s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge a \in \Sigma^O) \wedge ((s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge a \in \Sigma^I) \wedge pre = (pre_1 \wedge pre_2) \wedge post = (post_2 \wedge post_1)$
- $Pre = Pre_1 \cup Pre_2 \cup \{(pre_1 \wedge pre_2) | pre_1 \in Pre_1 \wedge pre_2 \in Pre_2\}$;
- $Post = Post_1 \cup Post_2 \cup \{(post_1 \wedge post_2) | post_1 \in Post_1 \wedge post_2 \in Post_2\}$.

Následující definice prezentuje synchronizovaný produkt dvou *interface automatů* reprezentujících kontrakt.

Definice 15. *(Ilegální stavy dvou interface automatů pro kontrakt)*
Nechť A_1, A_2 jsou propojitelné interface automaty. Množina ilegálních stavů v produktu je značena jako $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ z $A_1 \otimes A_2$ je definována jako $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1, A_2).(a \in \Sigma_{A_1}^O(s_1) \wedge a \notin \Sigma_{A_2}^I(s_2)) \vee (a \in \Sigma_{A_2}^O(s_2) \wedge a \notin \Sigma_{A_1}^I(s_1))\} \cup \{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \forall ((s_1, s_2), pre, a, post, (s_1', s_2')) \in \delta_{A_1 \otimes A_2}.((pre \equiv false) \vee (post \equiv false))\}$.

Množina neplatných (illegal) stavů obsahuje stavy, ve kterých sdílené akce mezi *interface automaty* nejsou synchronizované. Toto může nastat, pokud kontrakt (komponenta) vyžaduje nebo poskytuje akci, která buďto není poskytovaná nebo požadovaná. V dalším případě může neplatný stav nastat, pokud kontrakt (komponenta) požaduje nebo poskytuje akci, která je poskytovaná nebo požadovaná okolním prostředím, ale tato akce je nekompatibilní na sémantické úrovni.

9.4 Překlad The Leader Election Case Study do *rozšířených interface automatů*

Výše uvedený přístup bude demonstrován na případové studii, která vychází z reálné situace v průmyslu (Bryans, Fitzgerald,Payne, Kristensen, 2014). Tato případová studie se zabývá sítí audiovisuálních zařízení (AV). Tyto zařízení spolu komunikují prostřednictvím sítové vrstvy, která zajišťuje komunikaci jednotlivých zařízení.

Každé z těchto zařízení může být spravováno nezávisle. Každé zařízení musí také odpovídat kolekci kontraktů, z nichž každý specifikuje požadované chování, které musí být dosaženo, aby se zařízení mohlo stát součástí SoS.

Tato případová studie byla původně použita v publikaci (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014), ve které jsou identifikovány tři kontrakty, a to: Browsing Device, Streaming Device a LE Device. Pro zjednodušení bude vylepšený Cotract pattern prezentován pouze na LE Device a Transport Layer. V publikaci (Bryans, Fitzgerald,Payne, Miyazawa, Kristensen, 2014) je kontrakt pro LE Device definován za použití SysML diagramů a CML výrazů pro popis omezení vztahujících se na kontrakt.

Aby bylo možné konvertovat kontrakt do rozšířeného interface automatu, je nutné použít Contract Definition View, který popisuje preconditions, values a operations (input, output, hidden). Dále je nutné použít Contract Protocol Definition View pro identifikaci stavů a přechodů mezi nimi. Následující kód zobrazuje rozšířený interface automat pro LE device a Transport Layer.

LE device

- $S_{LD} = \{Off, OnFollower, OnLeader, OnUndecided, OnReady, OnUpdate\}$
- $I_{LD} = \{Off\}$
- $\Sigma^I_{LD} = \{receiveMessages\}$
- $\Sigma^O_{LD} = \{sendMessages\}$
- $\Sigma^H_{LD} = \{changeClaim, flushState, update, maxStrength, maxStrengthId,$
  $incStrength, init, flushMemory, flushSummary, isLeader, write, turnOn, turnOff\}$
- $V_{LD} = \{id, mem, highest\_strength, highest\_strength\_id, otherLeaders, myCS, isLeader\}$
- $Pre_{LD} = \{LDPreCC, LDPreW, LDPreIS\}$ where: {
  context $LEDevice :: changeClaimm(newClaim : Claim)$
  pre LDPreCC: $myCS.c =< off > \implies newc =< undecided >$
  $and\ myCS.c = < undecided > \implies (newc = < leader > or\ newc = < follower >)$
  $and\ myCS.c = < leader > \implies newc = < undecided >$
  $and\ myCS.c = < follower > \implies newc = < undecided >$

  context $LEDevice :: write(n : LE_Id, dat : DATA)$ ***pre LDPreW:*** $n\ in\ set\ dom\ mem$
  context $LEDevice :: incStrength()$ ***pre LDPreIS:*** $myCS.s < 10$
  }
- $Post_{LD} = \{LDPostCC, LDPostW, LDPostIS\}$ where: {
  context $LEDevice :: changeClaimm(newClaim : Claim)$
  post LDPostCC: $myCS.c = newClaim$
  context $LEDevice :: write(n : LE_Id, dat : DATA)$
  post LDPostW: $mem(n) = dat\ or\ mem(n).c = < off >$
  context $LEDevice :: incStrength()$
  post LDPostIS: $myCS.s = myCS\ .s + 1$
  }
- $\delta_{LD} = \{$
  - $Off : \textbf{\textit{turnOn}} : OnReady$
  - $OnReady : \textbf{\textit{receiveMessages}} : OnUpdate$
  - $OnUpdate : \textbf{\textit{update}} : OnReady$
  - $OnReady : \textbf{\textit{turnOff}} : Off$
  - $Off : \textbf{\textit{turnOn}} : OnUndecided$
  - $OnUndecided : LDPreCC : \textbf{\textit{changeClaim}} : LDPostCC : OnFollower$
  - $OnFollower : LDPreCC : \textbf{\textit{changeClaim}} : LDPostCC : OnUndecided$

- $OnUndecided : LDPreCC : \textbf{\textit{changeClaim}} : LDPostCC : OnLeader$
- $OnLeader : LDPreCC : \textbf{\textit{changeClaim}} : LDPostCC : OnUndecided$
- $OnFollower : \textbf{\textit{sendMessages}} : OnFollower$
- $OnLeader : \textbf{\textit{sendMessages}} : OnLeader$
- $OnUndecided : \textbf{\textit{turnOff}} : Off$
- $OnFollower : \textbf{\textit{turnOff}} : Off$
- $OnLeader : \textbf{\textit{turnOff}} : Off$

}

Transport Layer

- $S_{TL} = \{Init, Ready, CreateMessage, AddtoQueue, GetMessage, CreateUnreachableMessage, TurnDeviceOn, TurnDeviceOff, SendtoDevice, ReceivedMessage\}$
- $I_{TL} = \{Init\}$
- $\Sigma^I_{TL} = \{sendMessages\}$
- $\Sigma^O_{TL} = \{receiveMessages\}$
- $\Sigma^H_{TL} = \{init, addToQueue, getNextMsg, createMessage, AddToQueue, setDeviceOn, setDeviceOff, ready\}$
- $V_{TL} = \{queue, devOn\}$
- $Pre_{TL} = \{TLPreGNM, TLPreSDOF, TLPreSDON\}$ where: {
  context $TransportLayer :: getNextMsg()$ ***pre TLPreGNM:*** $queue \rightarrow notEmpty$
  context $TransportLayer :: setDeviceOff(indevId : LE_Id)$
  pre TLPreSDOF: $devOn[devId] \rightarrow notEmpty$
  context $TransportLayer :: setDeviceOn(indevId : LE_Id)$
  pre TLPreSDON: $devOn[devId] \rightarrow notEmpty$
  }
- $Post_{TL} = \{TLPostI, TLPostATQ\}$ where: {
  context $contextTransportLayer :: Init()$ ***post TLPostI:*** $devOn.domain() = node_idsand$ $devOn.range = falseandqueue.size() = 0$

  context $contextTransportLayer :: addToQueue(m : MSG)$***post TLPostATQ:*** $queue.size() = queue@pre.size() + 1andqueue.lastItem() = mandqueue@pre = queue(1, ..., queue.size())$
  }
- $\delta_{TL} = \{$
  - $Init : \textbf{\textit{init}} : TLPostI : ready$
  - $Ready : \textbf{\textit{sendMessages}} : ReceivedMessage$
  - $ReceivedMessage : \textbf{\textit{createMessage}} : CreateMessage$
  - $CreateMessage : \textbf{\textit{addToQueue}} : TLPostATQ : AddtoQueue$
  - $AddtoQueue : \textbf{\textit{ready}} : Ready$
  - $Ready : TLPreGNM : \textbf{\textit{getNextMsg}} : GetMessage$
  - $GetMessage : \textbf{\textit{receiveMessages}} : SendtoDevice$
  - $SendtoDevice : \textbf{\textit{ready}} : Ready$
  - $GetMessage : \textbf{\textit{createMessage}} : CreateUnreachableMessage$
  - $SendtoDevice : \textbf{\textit{createMessage}} : CreateUnreachableMessage$
  - $CreateUnreachableMessage : \textbf{\textit{addToQueue}} : AddToQueue$
  - $AddToQueue : \textbf{\textit{ready}} : Ready$
  - $Ready : TLPreSDON : \textbf{\textit{setDeviceOn}} : TurnDeviceOn$
  - $TurnDeviceOn : \textbf{\textit{ready}} : Ready$

- $Ready : TLPreSDOF : \boldsymbol{setDeviceOff} : TurnDeviceOff$ 81
- $TurnDeviceOff : \boldsymbol{ready} : Ready$ 82

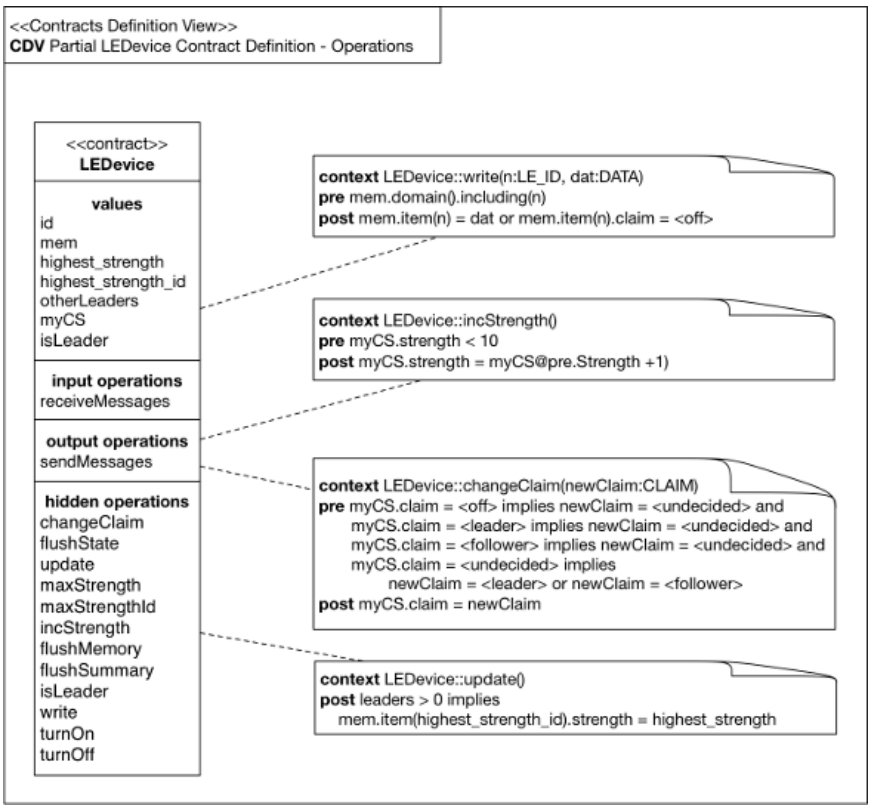
} 83
84

Na řádcích 3-6 a 45-49 si můžeme všimnout rozdělení operací do kategorií input, output a hidden. Využití OCL pro definici preconditions a postconditions je zřejmé na řádcích 8-25 a 51-65. Přechody mezi jednotlivými stavy jsou patrné na řádcích 23-41 a 66-83 rozšířených interface automatů.

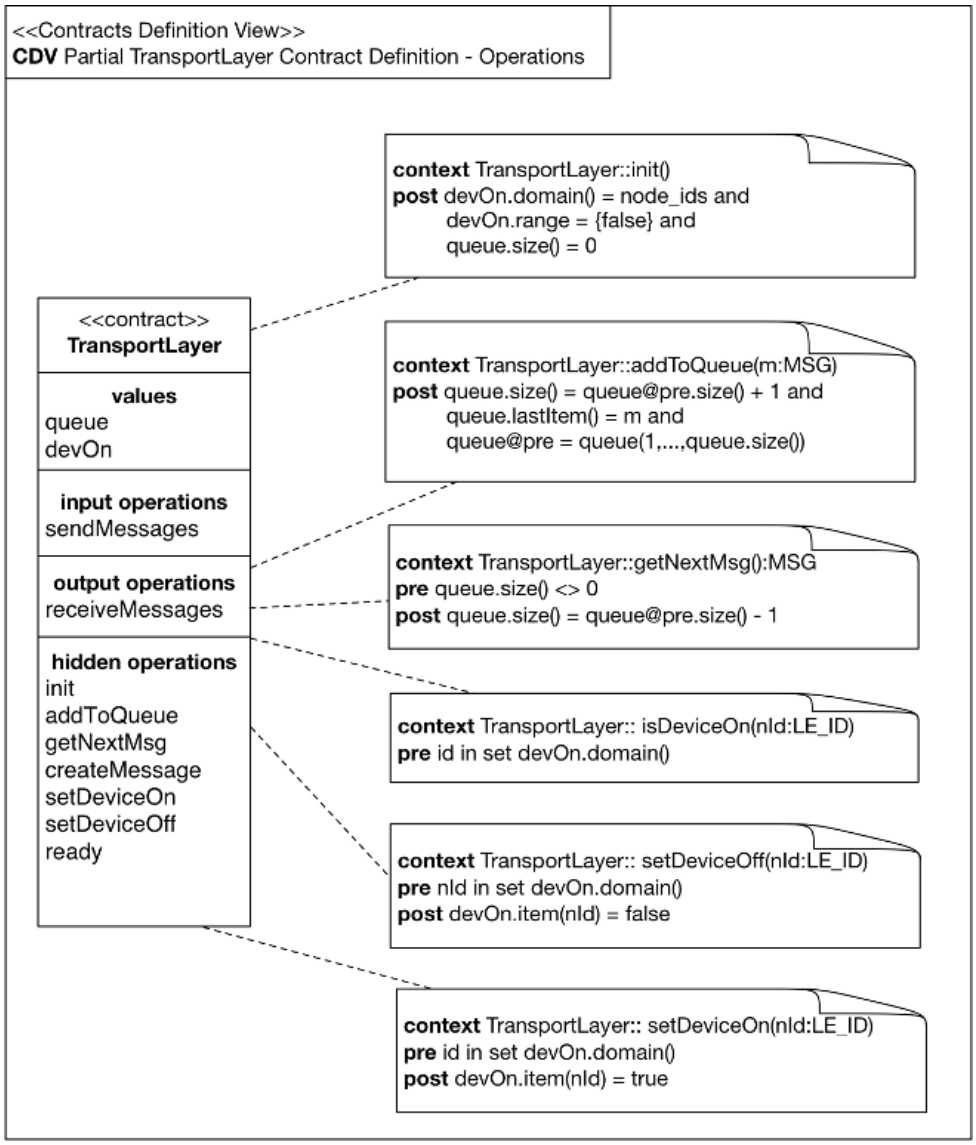
Klasifikace operací je klíčový přístup pro pozdější verifikaci celého SoS, proto vyžaduje při překladu pečlivé zvážení. Navrhované rozšíření vylepšuje diagram Contract Definition View, který je součástí Contract Pattern. Na obrázku 33 a 34 je patrné, že vylepšený Contract Definition View nyní rozlišuje operace podle toho, zda jsou input, output nebo hidden, což umožňuje nezávislost Contract patternu na Interface patternu, protože takto vylepšený Contract Definition View přesně určuje viditelnost operací.

Při překladu SoS popsaného pomocí původního Contract pattern do CML, ve kterém je pak možné provádět další detailnější verifikaci, bylo ponecháno rozhodnutí o viditelnosti operace na softwarovém architektovi, který tento systém převáděl. Při použití výše uvedeného rozšiřujícího Contract Pattern je však klasifikace metod provedena už v návrhové části SoS. Tento přístup zvyšuje transparentnost a relevantnost verifikace.

Nad SoS popsaném tímto způsobem můžeme provádět verifikaci kompatibility, jako je například ověření, že konstituční systémy jsou propojitelné na základě sémantiky jejich akcí. Kromě toho, s ohledem na synchronizovaný produkt, jsme schopni zjistit nesrovnalosti mezi sekvencí volání operací, které jsou dány jejich komunikačními protokoly.



Obrázek 33: Vylepšený Contract Definition View o klasifikaci operací pro LE Device



Obrázek 34: Vylepšený Contract Definition View o klasifikaci operací pro Transport Layer

Pokud porovnáme *rozšířený interface automat* s notací CML (viz níže Transport Layer), můžeme si všimnout, že v jazyce CML nedochází ke klasifikaci operací. Dále má také notace CML oproti *rozšířenému interface automatu* značně sníženou čitelnost.

```
process TransportLayer =
begin

state
  nodeOn : map NODE_ID to bool := {n |-> false | n in set node_ids}
  queue : seq of MSG := []
operations
  Init:() ==> ()
  Init() ==
  (
    nodeOn := {n |-> false | n in set node_ids};
    queue := []
  )
  post dom nodeOn = node_ids and
       rng nodeOn = {false} and
           queue = []

  addToQ: MSG ==> ()
  addToQ(m) ==
  (
    queue := queue ^ [m]
  )
actions
  TransportLayerLoop = mu X @ (TransportLayerAct;X)
  TransportLayerAct = (Reader [] Writer [] NodeMngt)
  Reader =  n_send?fr?too?payload ->
              (setNodeOn(fr);
                (dcl m:MSG @
                  m := createMSG(fr,too,payload); addToQ(m); Skip)
                  )
end

%%
```

Přínos zde nastíněného přístupu můžeme vidět ve dvou směrech. V prvním směru je navržena substituce CML notací OCL, která je použita jako rozšíření SysML v Contract pattern. Užití OCL poskytuje přirozenější doplnění diagramů v Contract pattern. Dále byla provedena klasifikace metod na input, output a hidden, která zvyšuje transparentnost a nezávislost Contract pattern na Interface pattern.

V druhém směru byl navržen vylepšený interface automat, jehož původním autorem je L. Alfarem and T. Henzingerem (Alfaro, Henzinger, 2001) a dále rozšířen o pre a post conditions autory Samir Chouali a kol. (Chouali, Mountassir, Mouelhi, 2010). Interface automat byl přizpůsoben a rozšířen o proměnné (values) kontraktu z důvodu využití těchto proměnných v preconditions a postconditions. Takto rozšířený interface automat umožňuje verifikaci kompozice kontraktů.

10 Diskuze

Práce se vyznačuje zavedením formalismů do problematiky komponent a komponentových systémů. Jsou formalizovány základní pojmy, jako je komponenta, komponentový systém a další, které jsou v literatuře (Brown, 2000), (Crnkovic, Larson, 2002), (D'Souza, Wills, 1999), (Szyperski, Gruntz, Murer, 2002) definovány verbálně. Tato formalizace přináší oproti verbální deskripci jednoznačnost definic v oblasti komponentových systémů. Na druhé straně kladou určité znalostní požadavky na uživatele těchto formalismů.

V práci je dále využita Teorie vyčíslitelnosti (interface automaty) pro formalizaci komponentového systému popsaného pomocí UML a provádění vybraných úloh založených na Teorii systémů. Na základě Teorie systémů jsou definovány základní operace nad komponentovými systémy, které lze provádět s komponentovým systémem popsaným v jazyce interface automatů. Tento přístup se odlišuje od jiných autorů (Alfaro, Henzinger, 2001), (Alfaro, Henzinger, 2004) a (Isazedech, Karimpour, 2008) využívajících interface automaty, kteří pouze definují kroky verifikace, ale ne už úlohy, které lze nad komponentovými systémy provádět. Jádrem uvedeného přístupu je reprezentace komponentového systému v notaci UML pomocí systému interface automatů. Ukazuje se, že jako jedna z možných reprezentací komponentového systému v interface automatech je pomocí formálního jazyka Ψ, který byl v této práci pro interface automaty navržen. Transformace komponentového systému v notaci UML do systému interface automatů je doprovázena tvorbou deskripce interface automatů (Faldík, 2015), (Faldík, 2014). Toto je odlišný přístup od výše uvedených autorů, protože ti se reprezentaci komponentového systému ve výše uvedených publikacích dále nevěnují.

Z pohledu problematiky Cyber-physical system (System of Sytems) bylo navrženo nahrazení jazyka CML jazykem OCL, který je použit jako rozšíření SysML, které je používáno v Contract pattern. Použití OCL místo CML více vyhovuje pohledům SysML v Contract pattern a dělá je tím přehlednější. Navíc byl vylepšen samotný Contract pattern, a to tak, že byly rozděleny operace na vstupní, výstupní a skryté, což odpovídá požadavkům na vazby mezi jednotlivými komponentami a tomu, že by měly mít mezi sebou slabou vazbu, tedy nejmenší množství závislostí. Toto rozšíření činí Contract pattern transparentnějším a nezávislejším na Interface pattern. Výsledek je dále prezentován na doméně Chytrých elektrických sítí (Smart Grids) (Faldik, Payne, Fitzgerald, Buhnova, 2017).

Další přínos je v návrhu rozšířeného interface automatu, který je založen na definici autorů Choulali a kol. (Chouali, Mountassir, Mouelhi, 2010), kteří ho rozšířili o definici preconditions a postconditions. Jejich definice byla v této práci přizpůsobena a rozšířena o hodnoty (values) kontraktu, které reprezentují stav. Reprezentace kontraktů pomocí rozšířených interface automatů umožňuje verifikaci propojení kontraktů.

Poznatky dosažené v této práci byly také využity v rámci projektu GAČR (Grantová agentura České republiky), Měření podnikové udržitelosti ve vybrany-

´ch odvětvích (GA14/23079S), kde jedním z úkolů bylo vytvořit webovou aplikaci pro hodnocení podnikové udržitelnosti a benchmarking (Faldík, Trenz, Hřebíček, Kassem, 2015). Architektura aplikace byla navržena jako komponentová z důvodu snadné rozšířitelnosti a udržitelnosti aplikace. V první fázi byl navržen a implementován webový portál pro pivovary a bioplynové stanice (Faldík, Trenz, Hřebíček, Kassem, 2015a), (Kassem, Trenz, Hřebíček, Faldík, 2016a), (Kassem, Trenz, Hřebíček, Faldík, 2016b). V průběhu řešení projektu došlo k několika vylepšením a rozšířením o modul pro zpracovatelský průmysl (Hřebíček, Trenz, Faldík, Kassem, 2016).

10.1 Navržené vylepšení

Z pohledu formalizace komponentových systémů pomocí interface automatů je možné rozšíření úloh na úlohy plynoucí ze specifikace těchto systémů odvozených z Teorie systémů (viz kapitola 7.1) a provedení rozšíření jazyka Ψ o tyto verifikační úlohy. Dále by bylo možné toto rozšíření implementovat do D-frameworku.

V průběhu tvorby disertační práce byla navázána spolupráce s výzkumnou skupinou profesora Fitzgeralda z Newcastle University, který se specializuje na Cyber-physical systems. Navržený formalismus v 9. kapitole bude dále vyvíjen a optimalizován. V budoucnu by měl také vzniknout plugin do verifikačního nástroje Symphony tool (The Symphony IDE, 2017).

Dále na základě disertační práce lze vytvořit moderní výukové materiály k současnému pojetí komponentových systémů pro předměty Softwarové inženýrství 1, 2, které jsou vyučovány na Provozně ekonomické fakultě Mendelovy univerzity v Brně (PEF MENDELU).

Disertační práce je využitelná softwarovými firmami. Na jejím základě si může softwarová firma vytvořit svoji vlastní firemní metodiku vývoje robustních komponentových systémů. Taková firma může na základě výsledků této práce vybudovat svůj verifikační framework pro robustní komponentové systémy.

10.2 Ekonomický přínos

Na přínos komponentové architektury lze pohlížet ze dvou pohledů, a to z pohledu firmy vyvíjející komponentový software a z pohledu zákazníka (např. firmy) kupujícího tento software. Každý tento pohled má svoje výhody a nevýhody. Pro doplnění kontextu uplatnění výsledků této práce je přiložena SWOT analýza (Faldík, 2013).

Tabulka 2 posuzuje rozhodnutí pro vývoj komponentové architektury z pohledu firmy vyvíjející SW. Tabulka je členěna na potencionální silné a slabé stránky tohoto vývoje, které může do jisté míry SW firma ovlivnit. Dále je členěna na příležitosti a hrozby, které mohou mít na SW firmu dopad z vnějšího okolí a které nelze snadno ovlivnit.

Tabulka 2: SWOT analýza CBS z pohledu dodavatele SW

Silné stránky (vnitřní původ)	**Slabé stránky (vnitřní původ)**
Znovupoužitelnost – využití jedné komponenty ve více systémech umožňuje soustředit se na jiné aktivity než vývoj té stejné položky.	*Vyšší nároky na SW architekta* – komponenta není jednorázový produkt a musí být navržena s ohledem na využití i v jiných systémech.
Zrychlení fáze vývoje – za předpokladu, že jde o znovupoužití komponenty.	*Vyšší nároky na programátory* – je nutné se seznámit s některým z komponentních frameworků.
Zjednodušení outsourcingu vývoje SW – kontrakt komponenty definuje očekávanou poskytovanou službu.	*Znovupoužitelnost* – vyšší nároky na modelování problémových domén.
Zapojení distributorů SW – funkčnost je jasně spojena s určitou komponentou. Distributor daného CBS může rozhodnout, které komponenty zahrne do cílového CBS svého klienta.	*Možnost vlastních úprav funkcionality komponenty od jiného SW výrobce* – v případě, že jde o blackbox komponenty.
Snadná konfigurace CBS	
Příležitosti (vnější původ)	**Hrozby (vnější původ)**
Rostoucí poptávka po systémech SW na míru.	*Nahraditelnost nově vzniklou technologií.*
Rostoucí podpora CBSE velkými producenty SW.	*Neochota výrobců SW vyrábět robustní SW s transparentním rozhraním.*
Vznik dalších robustních SW produktů založených na komponentové architektuře.	
Vývoj metod a modelů podporujících CBSE.	

Vzhledem k tomu, že přínos práce je v oblasti verifikace komponentových systémů na úrovni návrhu, může být ekonomický přínos pro firmu využívající tento přístup značný, protože včasné odhalení chyb už v návrhové fázi může ušetřit zbytečné investice do implementace softwaru, který by nebyl navržen správně. Dále verifikace na úrovni návrhu architektury může prodloužit životnost softwaru a výrazně snížit náklady na vydávání pozdějších záplat. Z pohledu zákazníka je zase dodán kvalitnější a spolehlivější SW a vzhledem k znovupoužitelnosti mohou být ušetřeny náklady na vývoj (viz tabulka 3).

Ekonomický přínos v oblasti verifikace Cyber-physical systémů (System of Systems) je stejný jako v případě komponentových systémů zmíněných v předchozím odstavci. Navíc vzhledem k tomu, že byl vylepšen Contract pattern tak, že byl upraven takovým způsobem, aby byl nezávislý na návrhovém vzoru Interface pattern. Jeho použití je snazší. Jazyk CML byl nahrazen jazykem OCL, který je znám široké komunitě softwarových architektů, čímž dochází ke snížení požadavků na softwarového architekta. V tomto případě není nutné provádět školení na jazyk CML a další návrhový vzor, jako je Interface pattern.

Tabulka 3: SWOT analýza CBS z pohledu zákazníka

Silné stránky (vnitřní původ)	**Slabé stránky (vnitřní původ)**
Rychlejší dodání produktů – z důvodu opakované použitelnosti komponent.	*Požadavky na HW* – některé CBS mohou mít vyšší nároky na HW z důvodu např. použitého komponentového modelu.
Snadnější distribuce stávajícího systému jinou společností – je nutné definovat rozhraní (co nabízí nebo může nabídnout).	*Vyšší požadavky na dodavatele* – ne každý dodavatel může navrhnout a implementovat požadovanou komponentu.
Zrychlení a zlepšení kvality testovací fáze – zákazník může požadovat takovou komponentu od dodavatele, která již existuje v jiném CBS.	*Speciální požadavky zákazníka musí být projednány s výrobcem a distributorem SW.*
Zvýšení kvality SW vývoje a řízení outsourcingu – pokud má být distribuován stávající SW systém.	
Transparentnost dodávaného softwaru – funkce komponenty je definována rozhraním, které by mělo být kompatibilní.	
Umožňuje snadnou rekonfiguraci CBS – může být použita dynamická architektura, která se rekonfiguruje za chodu systému.	
Příležitosti (vnější původ)	**Hrozby (vnější původ)**
Zájem ostatních zákazníků o CBS – což může vést k silné poptávce a tlaku na dodavatele s následným zvýšeným zájmem o CBSE.	*Ukončení tohoto typu SW architektury* – způsobené např. evolucí.
Připravenost SW výrobců vytvářet vzájemně kompatibilní SW.	*Vznik technologických překážek v dalším rozvoji CBSE.*
Vývoj metod a modelů podporujících CBSE.	
Vývoj dalších typů komponentních systémů, které by umožnily snadnější integraci komponent do CBS.	

11 Závěr

Objektové paradigma chápe členění software v souladu se svými pilíři a komponenta je objektovou třídou. Současný vývoj teorie a praxe je veden na objektové platformě. Uplatňování komponentových architektur se začíná promítat do vyspělých objektových metodik, např. UP a RUP a dalších. Pracovní postupy metodik nejsou ještě hluboce propracovány a není v nich výrazná orientace na dominanci vývoje software na platformě komponentové architektury. Mnoho rozdílných pohledů uplatnění komponentové architektury přesto nepřekáželo ustanovení koncepce specifického přístupu CBD (Component-based Development). Postupné používání vývojových metamodelů se ukazuje jako čím dál více časté.

Komponenty a komponentové systémy se mohou popisovat verbálně a formálně. Formální popisy jsou přesnější, ne často se v odborné literatuře ovšem objevují. V popisech se často prolínají jak rovina softwarová (nejnižší), tak i rovina informačního modelování. Současné pohledy na komponenty jsou poměrně široké, neustále se doplňují podle pokroků v praxi a teorii. Relevantními z nich jsou:

- Komponenta a její rozhraní
- Komponenta a její funkcionalita
- Komponenta – jednotka komponentové architektury (assembling)
- Komponenta podléhá pravidlům objektového paradigmatu
- Komponenta a její komplexita
- Komponenta a její implementace
- Komponenta a její opakovatelné využití (reusing)
- Specifikace komponenty
- Implementace komponenty versus její specifikace

V práci je řešen problém reprezentace komponentového systému v notaci UML pomocí systému interface automatů. Na základě dosažených výsledků je demonstrováno na několika příkladech, že tato reprezentace je možná zejména pomocí formálního jazyka Ψ, který je pro interface automaty sestrojen. Výsledkem transformace komponentového systému v notaci UML do systému interface automatů je protokol deskripce interface automatů. Tato deskripce je potom využita jak k provádění operací nad jednotlivými interface automaty, tak nad celým reprezentačním systémem interface automatů. Nutnou podmínkou provedení zmíněných aktivit je existence podpůrného frameworku. Formální jazyk Ψ a podpůrný framework představují dostatečně silné nástroje informačního modelování pro komponentové systémy. Návrh formálního jazyka a podpůrného frameworku je původní.

12 Reference

Alfaro, L., Henzinger, T. *Interface automata.* Proceedings of the joint 8th European software conference. New York: ACM Press, 2001, vol 26, pp 109–120.

Alfaro, L., Henzinger, T. *Interface-based design.* Marktoberdorf summer school, 2004, pp 50-65. Kluwer, April 12–15 .

Allen, R., Garlan, D. *The Wright architectural specification language.* Report CMU-CS-96-TB. Garnegie Mellon University, School of Computer Science, Pitsburg, 1996.

Arlow, J., Neustad, I. *UML 2 a unifikovaný proces vývoje aplikací.* Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

Böck, H. *Platforma NetBeans. Podrobný průvodce programátora.* Vyd. 1. Brno: Computer Press, 2010, 320 s. ISBN 978-80-251-3116-9.

Brim, L., Černá, I., Vařeková, P., Zimmerová, B. *Interaction automata as a verification-oriented component-based System specification.* ACM SIGSOFT Softw Eng Notes 31(2):1–8, 2006. DOI 10.1145/1108768.1123063.

Brown, A., W. *Large-scale, Component-based Development.* London: Prentice-Hall, Inc., 2000. ISBN 0-13-088720-X.

Bryans, J., Payne, R., Holt, J., Perry, S. *Semi-formal and formal interface specification for system of systems architecture.* In: Systems Conference (SysCon), 2013 IEEE International, pp. 612–619, DOI:10.1109/SysCon.2013.6549946.

Bryans, J., Fitzgerald, J., Payne, R., Kristensen, K. *Semi-formal and formal interface specification for system of systems architecture.* INCOSE International Symposium 24(1), 2014, pp. 166–181, doi:10.1002/j.2334-5837.2014.tb03142.x.

Bryans, J., Fitzgerald, J., Payne, R., Miyazawa A., Kristensen, K. *Semi-formal and formal interface specification for system of systems architecture.* In: System of Systems Engineering (SOSE), 2014 9th International Conference on, doi:10.1109/SYSOSE.2014.6892466.

Chouali, S., Mountassir, H., Mouelhi S. *An I/O Automata-based Approach to Verify Component Compatibility: Application to the CyCab Car.* Electronic Notes in Theoretical Computer Science 238(6), 2010, pp. 3–13, doi:10.1016/j.entcs.2010.06.002.

Crnkovic, I., Chaudron, M., Larsson, S. *Component-based development process and component lifecycle.* In International Conference on Software Enginee-

ring Advances, ICSEA'06, Tahiti, French Polynesia, 2006.

Crnkovic, I., Larson, M.,P.,H. *Building Reliable Component-based Software Systems*. London: Artech House, Computing library. 2002 ISBN 1580535585, 9781580535588.

D'Souza, D., Wills, A. *Objects, components, and frameworks with UML: the catalysis approach*. Reading, Mass.: Addison-Wesley, 1999, xxv, 785 p. ISBN 0201310120.

Eeles, P., Cripps, P. *Architektura software. Nepostradatelný průvodce návrhem softwarové architektury, která funguje*. Brno: Computer Press., 2011 ISBN 978-80-251-3036-0.

Eriksson, H., Penker, M. *Business modeling with UML. Business Paterns at Work*. USA: John Wiley & Sons, Inc. 2000 ISBN 0-471-29551-5.

Faldík, O. *Komponentové systémy v tvorbě software*. Diplomová práce. Brno: Mendelova univerzita v Brně, Provozně ekonomická fakulta, Ústav informatiky, vedoucí prof. Mišovič, 2014.

Faldik, O., Payne, R., Fitzgerald J., Buhnova, B. *Modelling System of Systems Interface Contract Behaviour*. Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017, DOI: 10.4204/EPTCS.245.1.

Faldík, O., Trenz, O., Hřebíček, J., Kassem, E. *Web Application for sustainability and Reporting for Czech Breweries*. In ŠIRŮČEK, M. - ŠKATULÁROVÁ, I. Enterprise and Compatitive Environment - Conference Proceedings. 1. vyd. Brno: Mendelova Univerzita v Brně, 2015, s. 198–203. ISBN 978-80-7509-342-4. URL: *https* : $//ece.pefka.mendelu.cz/sites/default/files/imce/ece_2015_final.pdf$.

Faldík, O., Trenz, O., Hřebíček, J., Kassem, E. *Web Environmental Information System for Corporate Performance Evaluation and Reporting*. In Adjunct Proceedings of the 29th International Conference on Informatics for Environmental Protection . Kodaň: University of Copenhagen, 2015a, s. 90–93. ISBN 978-87-7903-712-0.

Faldík, O. *Formalization and Verification of Component-Based Systems in UML via Interface Automata*. STÁVKOVÁ, Jana, ed. PEFnet 2015: abstracts : European scientific conference of doctoral students : Brno, November 19, 2015. Brno: Mendel University in Brno, 2015. ISBN 978-80-7509-362-2.

Faldík, O. *Component Based Systems and System Tasks Based on General Systems Theory*. STÁVKOVÁ, Jana, ed. PEFnet 2014: abstracts : [European scientific

conference of doctoral students : Brno, November 20, 2014]. Brno: Mendel University in Brno, 2014. ISBN 978-80-7509-152-9.

Faldík, O. *Component-based software architecture in relationship to enterprise management*. STÁVKOVÁ, Jana, ed. PEFnet 2013: abstracts : [European scientific conference of doctoral students : Brno, November 18, 2013]. Brno: Mendel University in Brno, 2013. ISBN 978-80-7375-906-3.

Garlan, D., Monroe, R.T., Wile, D. *Architectural description of component-based systems*. Foundations of Component-Based Systems, Cambridge University Press, New York, NY, 2000.

Hřebíček, J., Trenz, O., Faldík, O., Kassem, E. *Rozšiřující modul pro zpracovatelský průmysl pro webovou aplikaci WEBRIS*. MENDELOVA UNIVERZITA V BRNĚ. Rozšiřující modul pro zpracovatelský průmysl pro webovou aplikaci WEBRIS. HŘEBÍČEK, J. – TRENZ, O. – FALDÍK, O. – KASSEM, E. 2016.

Ingram, C., Payne, R., Fitzgerald, J., Couto, L. D. *Model-based Engineering of Emergence in a Collaborative SoS: Exploiting SysML and Formalism*. NCOSE International Symposium 25(1), 2015, pp. 404–419, doi:10.1002/j.2334-5837.2015.00071.x.

Isazedech, A., Karimpour, J. *: A new formalism for mathematical description and verification of component-based Systems*. Berlin: Springer Science + Business Media, LLC 2008.

The Architecture Working Group of the Software Engineering Committee *Recommended practice for architectural description of software intensive systems* Technical Report IEEE P1471–2000, Standards Department, IEEE, Piscataway, New Jersey, USA, 2000. ISBN 0-738-12518-0.

ISO/IEC 19501 *Unified modeling language specification, version 1.4.2.* Document formal/05-04-01, 2005: The Object Management Group.

Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Development Process*. Amsterdam: Addison Wesley Longman, Inc., 1998 ISBN 0-201-57169-2.

Kanisová, H., Muller, M. *UML srozumitelně*. Brno: Computer Press, 2007 ISBN 80-251-1083-4.

Kassem, E., Trenz, O., Hřebíček, J., Faldík, O. *Sustainability Assessment Using Sustainable Value Added*. Sustainability Assessment Using Sustainable Value Added. In Procedia - Social and Behavioral Sciences 220. Amsterdam: Elsevier Science BV, 2016a, s. 177–183. ISSN 1877-0428. URL: http://www.sciencedirect.com/science/article/pii/S1877042816305833.

Kassem, E., Trenz, O., Hřebíček, J., Faldík, O. *Web Portal for Corporate Sustainability Evaluation, Modelling and Benchmarking*. Web Portal for Corporate Sustainability Evaluation, Modelling and Benchmarking. In Proceedings of the 8th International Congress on Environmental Modelling and Software (iEMSs). Volume 4. Manno: International Environmental Modelling & Software Society (iEMSs), 2016b, s. 1254–1260. ISBN 978-88-903574-5-9. URL: *http* : *//www.iemss.org/sites/iemss*2016/*vol*4.*php*.

Král, J., Žemlička, M. *Autonomous components*. In SOFSEM 2000. Theory And Practice of Informatics, 2000, volume 1963 of Lecture Notes in Computer Science: 375–383.

Král, J., Žemlička, M. *Software confederations and alliances*. CEUR Workshop, In CAiSE Short Paper Proceedings, 2003, volume 74: 229–232.

Kruchten, P. *The Rational Unified Process, An introduction*. USA: Pearson Education., 2003. ISBN 0-321-19770-4.

Merunka, V. *Datové modelování*. 1. vyd. Praha: Alfa Publishing, 2006, 177 s. Informatika studium. ISBN 80-86851-54-0.

Mišovič, M., Faldík O. *Applying of Component system development in object methodology*. Acta univ. agric. etsilvic. Mendel. Brun., 2013,Vol. 61, Issue 7, pp. 2515-2522. ISSN 1211–8516.

Mišovič, M., Faldík O. *Applying of component system development in object methodology, case study*. Acta univ. agric. etsilvic. Mendel. Brun., 2013a, sv. 61, č. 7, s. 2523–2531. ISSN 1211-8516.

Negroponte, N., 2001 *Digitální svět, Being digital*. Praha: Management Press, vyd. 1., 2001. 207 s. ISBN 80-726-1046-5.

Object Management Group *The Object Constraint Language*. 2014. [online]. [cit. 2015-02-17]. Dostupné z: http://www.omg.org/spec/OCL/.

Perry, S., Holt, J., Payne, R., Bryans, J., Ingram, C., Miyazawa, A., Couto, L.D., Hallerstede, S., Malmos, A.K., Iyoda, J., Cornelio, M., Peleska, J. *Final Report on SoS Architectural Models*. Technical Report, COMPASS Deliverable, D22.6. 2014 Available at http://www.compass-research.eu/.

Rychlý, M., Weiss, P. *Modeling of Service-oriented Architecture: From Business process to Service realization*. Third International conference on Evaluation of Novel Approaches to Software Engineering Proceedings. Poland: Institute for Systems and Technologies of information, Control and Communication. 2008 ISBN 978-989-8111-28-9.

Sommerville, I. *Software Engineering (the* 9^{th} *edition)*. London: Publisher PEARSON, 2010. ISBN 10: 0-13-705346-0.

Szyperski, C., Gruntz D., Murer S. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, second edition, 2002. ISBN-13: 978-0321753021.

The COMPASS Project *The Symphony IDE, 2017*. Newcastle, UK: 2017 [cit. 2017-05-08]. Dostupné z: http://symphonytool.org.

Zeigler, B., P., Kim T., G., Praehofer, H., *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. 2nd ed. San Diego: Academic Press, c2000. ISBN 0127784551.