VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

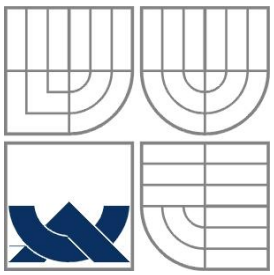
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELIGENT SYSTEMS

MODELEM ŘÍZENÝ NÁVRH KONFERENČNÍHO SYSTÉMU

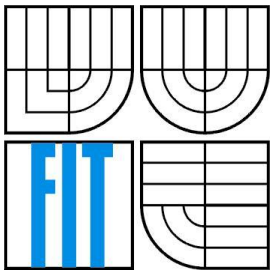
SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE Bc. MATĚJ CAHA
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELIGENT SYSTEMS

MODELEM ŘÍZENÝ NÁVRH KONFERENČNÍHO SYSTÉMU

MODEL BASED DESIGN OF THE CONFERENCE SYSTEM

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE
AUTHOR

Bc. MATĚJ CAHA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2013

Abstrakt

Modelem řízený návrh počítačových systémů je zaměřen na vytváření funkčních proveditelných modelů ve fázi návrhu systému, což vede k možnosti rychlého prototypování, testování a validace systému již v této fázi vývoje systému. Konkrétní implementace je poté vytvářena automaticky na základě překladových pravidel často specifikovaných vlastním formálním modelem. Místo transformace modelu na konkrétní implementaci může být funkční model zachován až k nasazení výsledného systému, čemuž se říká model continuity. Tímto způsobem lze efektivně využít metody modelování a simulace po celou dobu vyvíjení systému.

Pro tento účel je na fakultě informačních technologií na VUT v Brně vyvíjen nástroj SmallDEVS/PNtalk, který umožňuje programování modelů pomocí formalismů DEVS a OOPN. Objektově orientované Petriho sítě (OOPN) spojují výhody objektového programování a formálního aparátu Petriho sítí a dají se použít pro popis struktury a chování méně složitých systémů. Naopak DEVS je formalismus pro popis složitých diskrétních systémů řízených událostmi, kdy je možno systém skládat ze subsystémů (komponent), kde každý subsystém opět může být složen z dalších subsystémů. DEVS tedy definuje spojované komponenty, které mají propojení mezi svými vstupními a výstupními porty, a atomické komponenty, které mohou mít své chování popsány jiným formalismem, jako třeba OOPN, nebo konečný automat. Dohromady se jedná o silný modelovací aparát, jehož modelování/programování nám ulehčí nástroj SmallDEVS/PNtalk, postavený na čistě objektově orientovaném jazyce Smalltalk.

Abstract

Model-based design of computer systems is focused on creating functional executable models in the phase of design system, which leads to possibility of prototyping, testing and validation of the system at this stage of system development. The specific implementation is then created automatically based on the translation rules, which are often specified by its own formal model. The other way is to keep functional executable models in application, which is called model continuity. This is how you can effectively use the methods of system modeling and simulation throughout the developing process.

For this purpose a tool SmallDEVS/PNtalk was developed on faculty of information technology on VUT Brno. This tool allows programming models based on formalisms DEVS and OOPN. Object oriented Petri nets (OOPN) combine the advantages of OOP and formal apparatus of Petri nets. OOPN can be used to describe of structure and behavior of less complex systems. DEVS formalism is used to describe of complex discrete event systems, which may be composed of subsystems (components), where each subsystem can be an atomic component, or coupled component composed of other subsystems. Coupling of components is provided by input/output ports of component. Behavior of atomic components may be describe by other formalisms like OOPN or finite state machine. Altogether it is a powerful modeling apparatus and the framework SmallDEVS/PNtalk facilitates work on modeling and programming with it. SmallDEVS/PNtalk is based on purely object-oriented language Smalltalk.

Klíčová slova

Softwarové inženýrství, analýza a návrh počítačových systémů, , modelem řízený návrh, MBD, simulací řízený návrh, SBD, MDA, DEVS, OOPN, SmallDEVS, PNtalk, modelování a simulace.

Keywords

Software engineering, system analysis and design, model-based design, MBD, simulation-based design, SBD, MDA, DEVS, OOPN, SmallDEVS, PNtalk, modeling and simulation.

Citace

Caha Matěj: Modelem řízený návrh konferenčního systému, semestrální projekt, Brno, FIT VUT v Brně, 2013

Modelem řízený návrh konferenčního systému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Matěj Caha
9.1.2013

Poděkování

Děkuji svému vedoucímu práce panu Ing. Radku Kočímu, Ph.D. za uvedení do problematiky a za následné komentáře ke struktuře a obsahu této práce.

Obsah

1. Úvod

V této práci budou diskutovány aspekty návrhu počítačových systémů metodikou zvanou *modelem řízený návrh* (MBD - Model-based design). Shrneme zde dosavadní historii návrhu systémů, dále nastíníme princip metodiky MDA (Model Driven Architecture) dle specifikace skupiny OMG (1), poté navážeme na metodiku a technologii vyvíjenou na naší fakultě výzkumnou skupinou modelování a simulace systémů při ústavu inteligentních systémů (více v (2)).

Softwarové inženýrství se již spoustu let zabývá vývojem software a řešením problémů, které při vývoji vznikají, společně s vyvíjením nových technik k usnadnění a urychlení vývoje systémů. V současné době existuje mnoho metod vývoje SW, ze kterých si můžeme připomenout klasický Vodopádový model, pokročilejší Spirálový model, inkrementální iterační model vývoje, unifikovaný RUP nebo v neposlední řadě zmiňme agilní metodologie jako například Extrémní programování, nebo SCRUM.

Všechny zmíněné metodiky ve svých fázích vývoje více či méně využívají modely pro reprezentaci systému, nebo jeho částí. Modelování je pro nás důležité, neboť nám umožňuje postihnout ony vlastnosti systému, které nás zajímají. Pravděpodobně nejznámější modelovací prostředek je UML (Unified Modeling Language, (3)), který je však pouze informativní (vizualizační), lépe řečeno, neumožňuje ve své základní podobě provádění vytvořených modelů.

V oblasti softwarového inženýrství došlo v poslední době k rozvoji přístupu, který využívá modelování ve smyslu simulovatelných modelů a to v takém rozsahu, že je lze považovat za programovací jazyk (4). Tento přístup dostál mnoha označení, z nichž některá jsou *Model-based design* (MBD), také *Model-driven design* (MDD) a *Model-driven architecture* (MDA), nebo trochu komplexnější název *Model and Simulation based design* (MSBD). Všechny ale v podstatě značí to, že vývoj počítačového systému je podložen proveditelným modelem, například *xUML* (Executable UML, viz. kap. 3.1), nebo jinými formalismy, jako *Petriho sítě*, případně *DEVS*, které jsou probrány v kapitole 4. Tento proveditelný model může být testován již v průběhu analýzy, což je nesporná výhoda oproti klasickým metodám vývoje, kdy se systém testoval až po fázi implementace. Fáze implementace u modelem řízených metod návrhu může být potom prováděna manuálně, kdy programátoři vytvářejí kód na základě odsimulovaného modelu systému, nebo také (polo)automaticky, kdy musí být prvkům modelu jasně stanoveny pravidla transformace na zdrojový kód. Tuto transformaci mohou IT analytici částečně uzpůsobovat svým potřebám pomocí zavedení *Metaúrovní*, jako například *Meta-Object Facility* (MOF) u UML. Tato problematika transformace modelu je částečně popsána v kapitole 3.

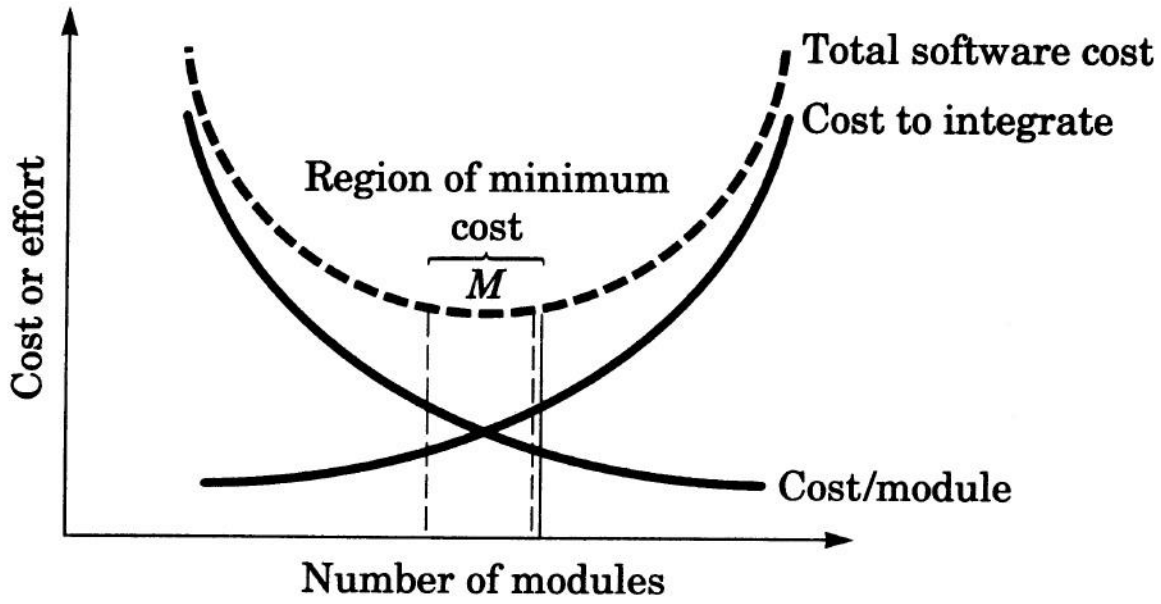
Kapitola 4 potom pojednává o významu modelování a simulace při vývoji systému a diskutuje výzkum probíhající na fakultě informačních technologií na VUT v Brně (viz. (4)).

2. Historie - cesta k modelem řízenému návrhu

Vývoj softwarového průmyslu od svého počátku dostál mnoha změn a postupně zde docházelo k vrstvení abstrakcí, díky kterých můžeme dnes psát programy o mnohem pohodlněji, než tomu bylo při psaní prvních počítačových programů. Mluvíme-li o programech ukládaných do paměti počítače, kdy měly počítače paměť počítanou v kB a výkon udávaný v kHz existovala v podstatě pouze jedna vrstva abstrakce, která mapovala jedna k jedné instrukční sadu jazyka na přímé provádění strojového kódu (5). Podobně tomu je i při návrhu počítačových systémů, kde také docházelo ke zvyšování počtu abstrakčních vrstev a vývoji různých metodik k ulehčení, ale hlavně urychlení, snížení chybovosti kódu a tedy snížení ceny vývoje. Pojďme si vývojem návrhu systémů tedy projít od začátku. Následující řádky popisující historii návrhu systémů jsou volně převzaty z (5).

Jak jsem psal v úvodu této kapitoly, na počátku vývoje počítačových systémů byly počítače a vývoj software dosti omezené. Každý programátor psal programy de facto ve strojovém kódu a vytvářel tak malé programy, které byly jednoduše srozumitelné jednomu člověku, který na nich pracoval. Počítače začaly být větší a výkonnější a společně s tím rostla i složitost vyvíjených systémů. S rostoucími nároky na složitost počítačových systému se programátoři museli spojit do týmů a pracovat na vývoji systému společně. Tím vznikla potřeba stanovit v týmu jistá pravidla psaní kódu. Program potřeboval být více strukturovaný čímž na začátku 70. let 20. století vznikl trend zvaný *Strukturované Programování*, který umožnil programátorům strukturovat kód do dílčích částí ve formě tzv. funkcí, případně procedur a tyto části používat na více místech programu. Pokročilejší strukturování vedlo ke vzniku ucelených funkčních bloků zvaných moduly. Programovací jazyky (jako třeba C, nebo C++) umožňující modulární programování jsou označovány jazyky třetí generace. Tento trend přinesl výhody dělení práce nad strukturovanými programy, avšak návrh systému zůstal tam kde byl, čili komplexní, nepřehledný, obtížný a náročný, což se s rostoucími programy ještě umocňovalo.

Možná tedy nebyl problém v implementaci kódu, ale v dřívější fázi životního cyklu softwaru (dále SW). Edward Yourdon a Larry Constantine to takto napadlo a uskutečnili posun směrem k *Strukturovanému Návrhu* SW. Navrhli dvě klíčová kritéria kvality návrhu: *propojení* a *koheze*. Obě dvě kritéria jsou míněna ve smyslu *nezávislosti*. Šlo jim tedy o to, aby při návrhu vznikly moduly, které jsou na sobě co nejméně závislé a to jak ve smyslu propojení, tak ve smyslu soudružnosti dat v modulu. Počet modulů je potřeba volit rozumně. Obrázek 2.1 ukazuje závislost ceny vývoje softwaru na počtu modulů, kdy s počtem modulů sice klesá cena za jeden vytvořený modulu, ale na druhou stranu roste cena za integrování těchto modulů do výsledného systému. Součtem těchto křivek dostaneme celkovou cenu vývoje, která je ve tvaru kolébky a je ideální volit takový počet modulů, který odpovídá dnu této kolébky (minimu funkce celkové ceny). Obě tyto vlastnosti jsou klíčové také pro modelem řízený návrh, ke kterému směřujeme. Dříve zmínění pánové také nabídli světu návrhový formalismus zvaný *Diagram Struktury,* který návrhářům umožnil grafickou vizualizaci navrhovaného kódu.



Obrázek 2.1 ukazuje závislost ceny vývoje SW (Cost or effort) na počtu modulů (Number of modules). Převzato z (6).

Zaměření na dřívější etapu životního cyklu SW pokračovalo dále a vyústilo ve *Strukturovanou Analýzu*, která byla popularizovaná lidmi Tom DeMarco (1978) a Edward Yourdon (1989). Principem bylo oddělení specifikace problému od jeho řešení, což vedlo k vytvoření dvou primárních modelů. První, zvaný *Základní Model*, sloužil jako reprezentace systému, bez zaměření na implementaci. Druhý, zvaný *Implementační Model*, byl funkční realizací základního modelu s důrazem na organizaci hardwaru, softwaru a zdrojového kódu. Zde má základy princip MDA se svým platformě nezávislým a platformě specifickým modelem, ale k tomu se dostaneme v kapitole 3. Při vytváření modelů v této etapě jsme se mohli setkat s abstrakcí typu *DFD* (Data Flow Diagram), *SD* (State diagram) a *ERD* (Entity Relationchip Diagram), které dnes v mírně upravené podobě potkáváme v modelovacím jazyce *UML* (Unified Modeling Language). V této době vznikl také *událostmi řízený* přístup, o který se zasloužili pánové Ward a Mellor (1985), kteří řekli, že systém nemusí být rozdělen s*hora-dolů,* ale *zvenku-dovnitř*. Jejich pohled byl založen na porozumění prostředí ve kterém systém běží a vytvoření specifikace chování tak aby vyhověla tomuto prostředí. Tento přístup se dá připodobnit k *Use-Case* diagramům, které opět najdeme v *UML*.

Mezitím, co byly vyvíjeny strukturované metody, si návrháři uvědomili, že mohou existovat i jiné způsoby rozdělení. Přišli na to, že objekty v systému jsou mnohem více stabilní, než funkce nad nimi prováděné. Vznikly tedy *Objektově Orientované* (OO) *Metody*, které se snaží přizpůsobit reálnému světu tím, že implementují objekty, které mají své vlastnosti a funkce, neboli metody, které mohou provádět. Knihy o OO programování autorů Booch (1986), Mayer (1988) a dalších byly následovány knihami o *OOD (Object-oriented Design)* a *OOA (Object-oriented Analysis)*. Tyto přístupy byly aplikovány hned v počátku životního cyklu SW, kde jsou prováděna nejdůležitější rozhodnutí ve formě specifikace chování a pravidel systému. MDA se ztotožňuje s touto aplikací a zavádí ji také do prvotní fáze vývoje.

Objektově orientovaný přístup se stal populárním a lidé začali vymýšlet, jak by šel vývoj za pomocí objektů ještě zjednodušit. Booch v publikaci *Object Oriented Analysis and Design with Applications* (1993) detailně popsal metodu OO vývoje. Principy užívání *vzorů*, nebo *archetypů*, jako základ pro platformě specifickou implementaci, byly obhajovány i mnoha dalšími (např.: Beck, Shlaer, Mellor, Buschmann, Meunier, Coad). Celé to završila čtveřice Gamma, Helm, Johnson a Vlissides, kteří roku 1994 publikovali knihu *Design Patterns*, která je považována za nejznámější v oboru *návrhových vzorů*. Ivor Jacobson se svým přístupem řízeným *případy užití* (angl. Use-Case) roku 1992 pozvednul povědomí o důležitosti organizačních požadavků ve smyslu lehce modelovatelných a sledovatelných v průběhu životního cyklu. Roku 1991 vyvinul Dr. James Raumbaugh s kolektivem techniku zvanou *OMT (Object Management Technique)* ve které navrhli soubor notací pro podporu vytváření modelů analýzy a návrhu. Tato technika se v 90. letech stala dominantní.

Shlaer a Mellor s jejich *rekurzivním návrhem* zdůraznili význam vytváření precizních *PIM* (Platform Indipendent Model), které mohou být systematicky, případně automaticky, překládány až k vygenerování zdrojového kódu cílového systému. Dále prosadili nový přístup strukturování systému a to na základě tematických celků, čili sdružení tematicky shodných prvků do jednoho modulu zvaného *doména*. Výsledkem tohoto rozdělení systému jsou moduly vysoce nezávislé na ostatních, tak jak bylo požadováno již před 20 lety při *Strukturovaném Návrhu* pány Yourdon a Constantine. Podobným směrem se vydal kolektiv autorů v čele s Bran Selic, kteří v roce 1994 představili metodu nazývanou *Real-time Object-oriented Modeling (ROOM)* zaměřenou na vývoj RT systémů s důrazem na definování rozhraní a komunikačních protokolů. Oba tyto přístupy umožňují grafické znázornění, které je dnes standardizováno v UML.

Aktuální stav návrhu a vývoje systémů je považován za sjednocení těch nejlepších metod z historie a může být popsán následujícími vlastnostmi:

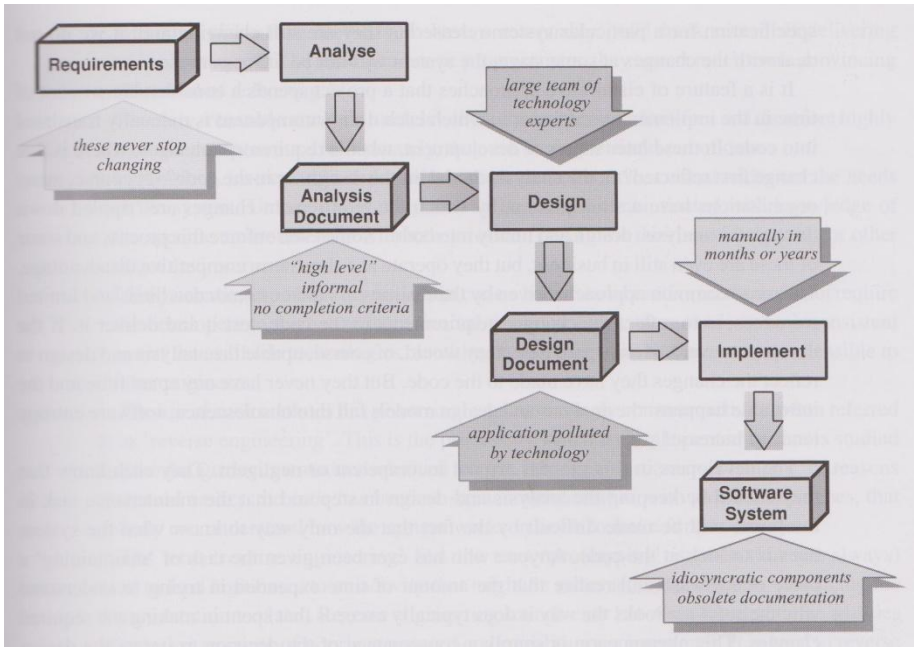
1. Rozdělení systému na domény, obsahující prvky/objekty zapadající do stejného tématu, které jsou téměř nepropojené a kohezní;
2. Oddělení platformě nezávislého chování od platformě specifického chování, jako rozšíření dříve definovaného oddělení *základního modelu* od *implementačního modelu*;
3. Definice vzorem řízeného mapování pro systematické vytváření *platformě specifických* modelů z *platformě nezávislých* modelů. Toto mapování může být manuální, ale je snaha o plnou automatizaci;
4. Používání abstraktního, ale úzce sémanticky definovaného formalismu *xUML* (executable UML), které zajišťuje modelům preciznost a možnost jejich testování/simulaci již v době modelování systému.

Tímto jsme prošli historií návrhu systémů a v následující kapitole se zaměříme na metodiku MDA.

3. Metodika Model Driven Architecture

MDA je metodika vývoje SW založená na proveditelných modelech. Je vyvíjena skupinou OMG (1), která má pod sebou i mnoho dalších metodik a technologií zaměřených na interoperabilitu a portabilitu objektově orientovaných aplikací, uveďme pár známějších např. UML, XML, XMI, COBRA.

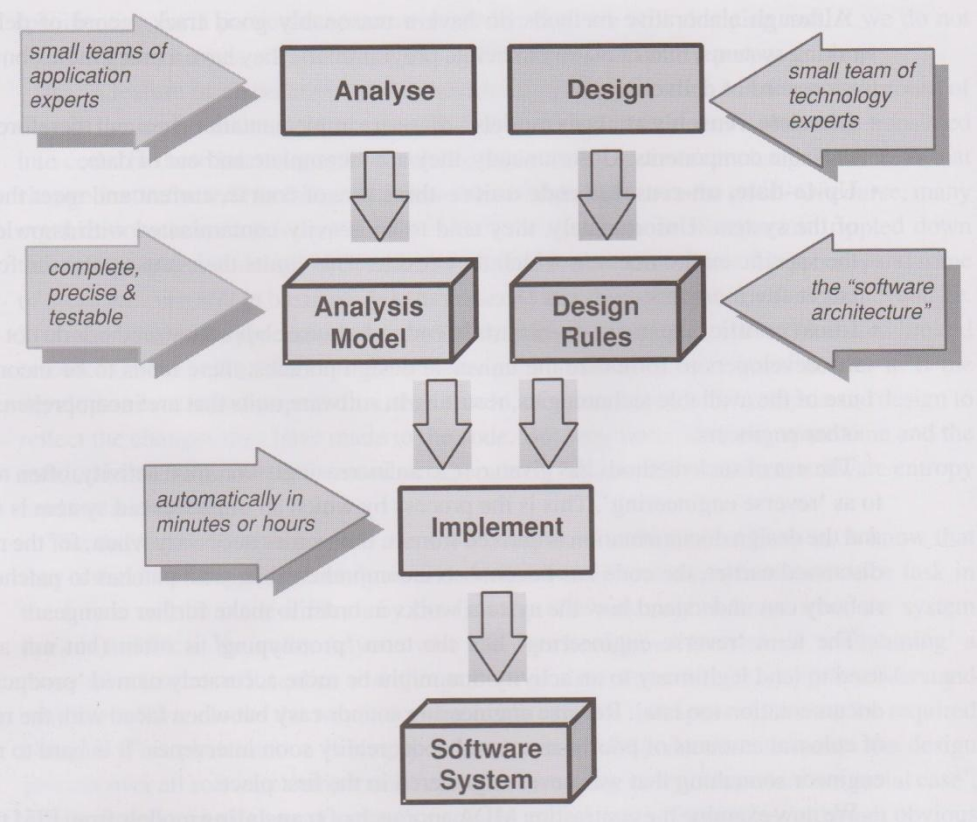
Metodika MDA je tedy technika vývoje, kdy je striktně oddělen model systému ve smyslu požadovaného chování a funkcionality od implementačních detailů závislých na použitých technologiích. Model funkcionalit a chování systému, zvaný *PIM* (viz kap. 3.2), stačí vytvořit pouze jednou. Tento model je poté mapován pomocí překladových pravidel na *PSM* (viz kap. 3.3) a konkrétní implementaci na dané platformě, zvanou *PSI* (Platform Specific Implementation). Pro jeden *PIM* tedy může existovat mnoho *PSM*, které definují transformaci modelu *PIM* na platformě závislou implementaci *PSI*. Nutno říct, že princip MDA spočívá také právě ve formě modelu *PIM*, který je proveditelný a tedy testovatelný v průběhu vývoje.



Obrázek 3.1 Diagram vývoje systému pomocí klasické vývojové metody Vodopád. Převzato z (5).

Klasické vývojové metody (např. Vodopád) stanoví problém (specifikace systému), který dále rozvíjí zaváděním návrhových a implementačních detailů, až dojdou k řešení, které představuje výsledek vývoje SW, čili konkrétní implementaci na dané platformě. Vývoj SW klasickou metodou Vodopád probíhá, jak znázorňuje Obrázek 3.1, postupně od specifikace požadavků (které se bohužel neustále mění), přes fázi analýzy, kdy jsou neformální požadavky na systém jasně a stručně přepsány do dokumentu, ze kterého návrhový tým vytvoří model systému, který specifikuje veškeré chování systému pomocí různých diagramů (např. diagram tříd, diagramy interakce, atp.). Tento model je pouze ilustrativní a musí být manuálně programátory transformován do výsledného kódu systému. Je to rozvíjející proces (*Elaborative procces*).

Naopak v metodice MDA je stanoven problém ve formě formálního funkčního modelu *PIM*. Požadavky na systém jsou analyzovány a je z nich vytvořen funkční formální model, který se dá testovat. Současně s tím může být vytvořen (resp. zakoupen, nebo znovupoužit) model návrhu, přesněji model transformace funkčního modelu na konkrétní implementační kód, v závislosti na použité platformě. Tato transformace by měla být plně automatická, čímž zabere mnohem méně času, než manuální implementace u klasických metod. Tento překladem řízený proces (*Translation-based process*) je znázorňuje Obrázek 3.2. (5)



Obrázek 3.2 Diagram vývoje systému pomocí MDA. Převzato z (5).

Dalším problémem klasických metod softwarového inženýrství je převod statického modelu systému (nejčastěji vytvořeného pomocí UML) na funkční implementaci SW. Tuto konverzi provádějí programátoři ve fázi implementace. V jejich práci jim mohou pomáhat různé (polo)automatizované transformace vybraných modelů, ze kterých lze získat kostru programu, či přímo kompletní kód nějakého modulu. Zde se objevuje jedno velké nebezpečí, které vzniká právě jednosměrností transformace (4). Je velmi řídký jev, že programátor při dodatečných úpravách vyvíjeného systému zanese úpravy do všech artefaktů vývoje, čili analýzy, návrhu a samotné implementace. Tím vzniká nekonzistence vygenerovaného a upravovaného kódu vůči dříve navrhnutým modelům. U metodik řízených modelem se vždy upravuje model samotný, který je opět formálně verifikován a případně znovu generován do zdrojového kódu. (5)

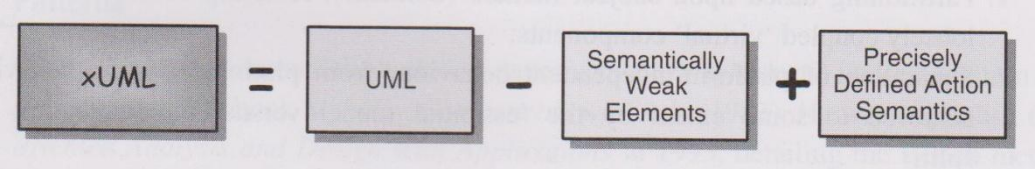
Koncept MDA využívá, rozšiřuje a integruje většinu již existujících a používaných specifikací skupiny OMG, jako třeba UML (Unified Modeling Language), MOF (Meta-Object Facility), XML (Extensible Markup Language) a IDL (Interface Definition Language). (7)

3.1 Srovnání UML a xUML

MDA se od klasických vývojových metod odlišuje také použitím modelovacích jazyků a modelu obecně. Zatímco u klasických metod je ve fázi analýzy a návrhu využíván model pouze coby reprezentativní a informativní formalismus, u MDA je tento model obdařen proveditelností, je tedy možné ho otestovat ihned po vytvoření. Tento přístup pomáhá odhalit chyby návrhu hned na počátku vývoje systému. (5)

UML (Unified Modeling Language) je modelovací jazyk, který poskytuje sjednocenou notaci pro reprezentaci různých aspektů objektově orientovaných systémů. Jazyk UML byl také specifikován skupinou OMG a jeho kompletní specifikace je dostupá zde (3). Náplní této práce však není rozbor jazyka UML, přesto se na dalších řádcích se předpokládá alespoň základní znalost UML.

Jak už bylo řečeno, UML je pouze notační nástroj. UML samotné nemá přesně definovanou sémantiku značek, umožňuje použití různých značek v jiném významu v jiných diagramech (např.: stavový model může být použit k popsání Use-Case diagramů, diagramů objektu, nebo podsystémů) (3). Tato vlastnost je pro MDA nežádoucí, modelování v MDA potřebuje striktní a precizní syntaxi a sémantiku notačních elementů, aby bylo u každého modelu přesně jasné co znamená a jaké chování reprezentuje. Toho bylo dosaženo vytvořením specifikace xUML (Executable UML), která je založena na jádře jazyka UML a doplněna Action Semantics (OMG, 2002), specifikující přesné chování používaných UML modelovacích elementů a odstraňuje mnoho nejasností elementů jazyka UML. Vzniká tak sjednocený modelovací jazyk, jehož pomocí lze vytvářet proveditelné modely. Vztah specifikace xUML s UML znázorňuje Obrázek 3.3. (5)



Obrázek 3.3 Vztah xUML a UML. Modelovací jazyk xUML čerpá z výhod klasického UML, ale odstraňuje diagramy a elementy, které nemají přesně danou sémantiku (chování), zato přidává a doplňuje přesnou specifikaci syntaxe a sémantiky zbylých elementů. Převzato z (5).

3.2 Platformě nezávislý model PIM

V MDA je pojem *platforma* používaná pro odkázání na technologické a návrhové detaily, které jsou irelevantní pro funkcionalitu SW. Platformě nezávislý model (PIM) je formální model popisující funkcionalitu systému, nebo jeho dané oblasti. PIM řeší koncepční otázky, ale nezohledňuje, jak bude vypadat technologická implementace. Pomocí dříve popsaného xUML tak můžeme vytvořit formální, precizní a proveditelný model PIM. Testování vlastností systému na úrovni PIM modelu přináší urychlení odladění případných chyb. K PIM modelu se často přidává PIM Metamodel do kterého mohou IT analytici vložit instrukce k překladu na PSM, aby výsledná implementace splňovala jejich výkonnostní požadavky. (5)(7)

Základ PIM (jako modelu jedné domény) je diagram tříd, od kterého se odvíjejí další diagramy jako stavový diagram, nebo diagram aktivit (5). V MDA je model PIM vytvářen pomocí xUML. Tento přístup ale obecně není jediný, jak uvidíme v kapitole 4.

3.3 Platformě specifický model PSM

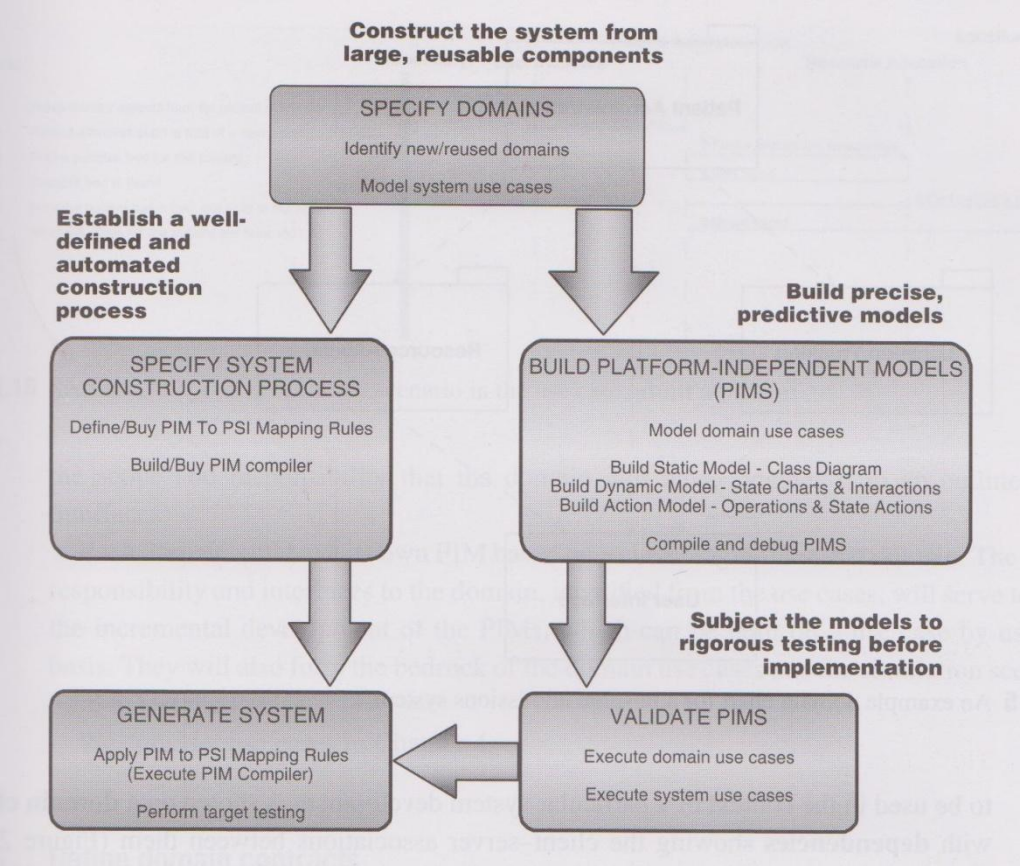
Platformě specifický (závislý) model (PSM) reprezentuje veškerou funkčnost systému z PIM modelu s přidanou vazbou na konkrétní realizační platformu (např. COBRA, C++, JAVA). PSM je vytvořen z PIM aplikováním transformačních pravidel, tento proces se nazývá *mapování*. Mapování samotné je také definováno formálním modelem. Pro jeden PIM může existovat více PSM modelů. (5)

Existuje více přístupů generování PSM modelů. V prvním z nich, je PSM generován lidmi, týmem odborníků, který z PIM vytvoří PSM na základně cílové platformy. Tento přístup je však neefektivní. Přechodem k MDA jsme chtěli dosáhnout urychlení vývoje SW. Další přístup generování PSM je automatizované generování. Tento přístup již splňuje naše požadavky a je vhodný pro svou rychlost. Základem je, aby mapovací proces byl velmi dobře a precizně popsán/namodelován. Potom je možné úpravy v PIM modelu ihned přeložit a otestovat. Nedoporučuje se dělat zásah do vygenerovaného kódu, neboli PSI (Platform Specific Implementation), protože tím ztratí svou absolutní soudržnost s PIM modelem. (5)

3.4 Souhrn MDA

MDA je proces vývoje systémů, který je dobře specifikovaný pod záštitou skupiny OMG, a ve světě používaný. Reprezentuje promyšlenou syntézu dobře otestovaných metodik softwarového vývoje. Proces vývoje systémů pomocí MDA se dá zjednodušeně popsat následujícími kroky, případně kresbou Obrázek 3.4.

1. Stanovit problém, získat požadavky na systém, identifikovat nebo znovupoužít funkční celky zvané *domény*
2. Vytvořit nebo znovupoužít precizní, prediktivní modely PIM odpovídající specifikací domén z předchozího kroku (diagramy tříd, interakce, chování elementů) - první část pravé větve diagramu v Obrázek 3.4;
3. Podrobit modely PIM důkladnému testování a validaci před samotnou realizací - druhá část pravé větve v Obrázek 3.4;
4. Vytvořit, znovupoužít nebo koupit modely PSM pro automatický převod (mapování) PIM modelů na konkrétní implementaci PSI - levá větev diagramu v Obrázek 3.4;
5. Generovat výsledný systém z otestovaných a validovaných PIM modelů pomocí PSM modelu a provézt konečné testování na cílové platformě.



Obrázek 3.4 MDA proces a dílčí artefakty vývoje. Převzato z (5).

4. Modelování a simulace v procesu vývoje systémů

V předchozí kapitole jsem se zaměřil na existující uznávaný modelem řízený princip návrhu a vývoje SW zvaný *Model Driven Architecture*. V této kapitole poukážu na možnost zajít s teorií modelování a simulací ještě dále, než tomu je u MDA. Inspirací pro tuto kapitolu mi byl výzkum výzkumné skupiny Modelování a Simulace na fakultě FIT VUT v Brně (2). Tato skupina, kromě jiného, zkoumá možnosti vývoje počítačových systémů přístupem kombinace modelem řízeného návrhu *MBD* (model-based design) s interaktivním inkrementálním vývojem (angl. exploratory programming), který spočívá v inkrementálním přechodu od modelu k realitě. Ve spojení se simulací daného modelu, případně jeho částí v průběhu vývoje, získáváme simulací řízený vývoj *SBD*, který umožňuje rychlé prototypování a simulování modelovaného systému, což je oproti klasické metodě obrovská výhoda. Formální model navíc umožňuje pokročilé funkční charakteristiky pomocí skládání spojitých a diskrétních stavebních bloků.

Jak již bylo naznačeno v kapitole 2, aktuální vývoj v oblasti softwarového inženýrství spočívá v posuvu od statických k dynamickým modelům a jejich simulaci, které umožňují efektivní analýzu procesů odehrávajících se ve vyvíjeném systému. Výše zmíněná výzkumná skupina se snaží dodržet tento trend s tendencí zachovat proveditelný model v průběhu celého vývoje až k cílovému nasazení. S tímto přístupem přišel již Ziegler v (8) a pojmenování tohoto přístupu se ustálilo na pojem *model continuity*. Zde si všimněme rozdílnost od metodologie MDA (popsané v kap. 3), která využívaný model systému nakonec transformuje do klasického zdrojového kódu (většinou OOP).

K udržení proveditelného modelu až po nasazení systému do reálného prostředí často pomohou techniky simulace *HIL (hardware-in-the-loop simulation)* a *SIL (software-in-the-loop simulation)*, které umožňují propojení reálných a simulovaných komponent v průběhu vývoje systému (4).

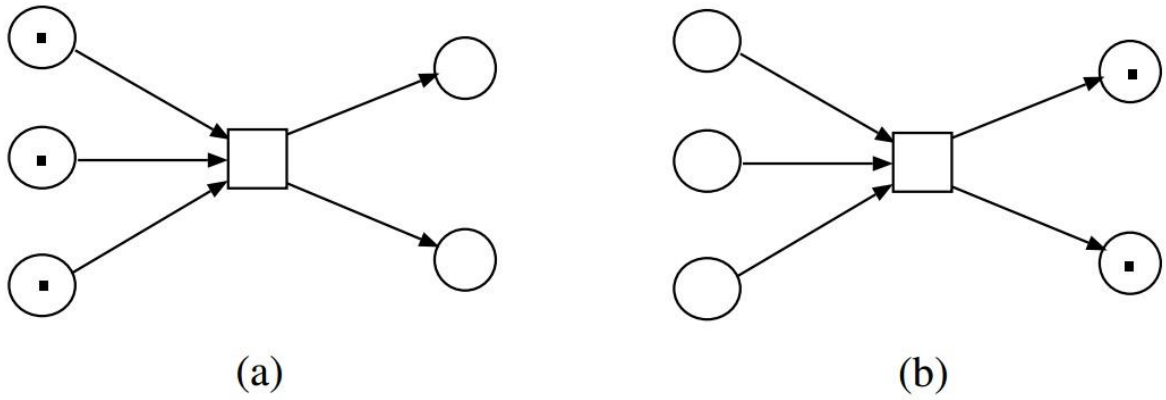
Vyvíjený model systému, který je v průběhu vývoje simulován, jak s virtuálními prvky, tak s reálnými, je potřeba formálně verifikovat. Výzkumná skupina Modelování a Simulace se proto snaží nahradit tradiční programovací jazyky formálními modely. Zaměřili se na formalismy jako *vysokoúrovňové objektově orientované Petriho sítě* (OOPN), *DEVS* a stavové diagramy (angl. state charts). Pro první dva formalismy implementovali nástroje, které kombinují programování a formální modelování, jde o *programovaní modely*, které umožňuje jednodušší práci s modely. (4)

Oba tyto formalismy a k nim vytvořené nástroje blíže proberu v následujících podkapitolách.

4.1 Objektově orientované Petriho sítě

Petriho sítě představují vhodný formalismus pro modelování diskrétních systémů, s možností grafického znázornění, analýzy a simulace. Populárnost Petriho sítí je dána jejich jednoduchostí. Model je sestaven z *míst*, které obsahují stavovou informaci ve formě *značek* (tokenů), dále *přechody*,

vyjadřující možné změny stavu a *hranami*, které propojují *místa* a *přechody*. Hrany mohou být ohodnocené, kde ohodnocení hrany určuje počet značek přenášených najednou. Provedení přechodu, též zvané *výskyt události*, u základních Petriho sítí je závislé na aktivaci přechodu. Přechod se stane aktivním, když má ve všech vstupních místech (místa napojená přes vstupní hrany přechodu) uloženou značku, která je následně ze vstupních míst odebrána a generována do všech výstupních míst, kde pro ni musí být místo (některá místa mohou být kapacitně omezena). Provedení přechodu je vidět na následujícím Obrázek 4.1. (9)



Obrázek 4.1 Příklad přechodu a) => b) jednoduché Petriho sítě. Malé čtverečky představují tokeny, které jsou přechodem (čtverec) odebrány ze vstupních míst (kružnice) a generovány do výstupních. Přechod je aktivován pouze je-li ve všech vstupních místech token a ve výstupních místech je prostor pro vygenerování tokenu. Převzato z (9).

Petriho sítě se dělí na různé typy, které vznikaly snahou zvýšit modelovací schopnosti tohoto formalismu při zachování koncepční jednoduchosti (9). Různé typy tedy jsou: C-E (Condition-Evets) sítě, P/T sítě, dále vysokoúrovňové (High-Level) sítě, jako například predikátové (Predicate-Transition), nebo barvené (Coloured) sítě, které umožňují pohodlně modelovat i zpracování a tok dat (9).

Protože Petriho sítě ve své původní podobě neposkytují strukturovací mechanismy známé z programovacích jazyků omezovalo se použití Petriho sítí pouze na nepříliš rozsáhlé systémy. Podíváme-li se na rozvoj programovacích jazyků a hlavně objektově orientovaného přístupu a postavíme jej vedle jednoduchých Petriho sítí, zjistíme, že tyto přístupy se zdají být ve svých výhodách a nevýhodách komplementární (9). Není divu, že lidé začali vymýšlet jak tyto přístupy propojit a získat tak od každého výhody a odstranit nevýhody, čili získat dobře strukturovaný, paralelní a hlavně formální (de facto matematický) programovací nástroj. Jeden z možných způsobů zavedení objektové orientace do Petriho sítí je popsán panem Janouškem v jeho disertační práci (9) publikované v roce 1998. Kromě zavedení objektového paradigmatu do Petriho sítí, které označujeme zkratkou OOPN, se pan Janoušek ve své práci zaměřuje na vytvoření nástroje *PNtalk* pro práci s těmito OOPN. Spojením Petriho sítí s objektově orientovaným přístupem bylo dosaženo modelování aktivity objektu v objektu samotném, nikoliv nadřazenou strukturou, jak je tomu u klasických OO programovacích jazycích jako třeba Java, nebo C++, které často umožňují kvaziparalelizmus. Tímto se objekty stávají samostatnějšími a opravdu paralelními entitami, které spolu komunikují předáváním zpráv.

Objektové modelování pomocí Petriho sítí je umožněno zapouzdřením funkčnosti objektu do třídy. Třída obsahuje definici objektu v podobě objektové sítě, která modeluje činnost objektu samotného. Místa v objektové sítí reprezentují objektové atributy a mohou na ně být navázány predikáty, které testují místa, případně synchronní porty, které umožňují provedení synchronní akce více objektů najednou. Metody objektu jsou modelovány vlastní Petriho sítí, ale mají přístup k objektovým atributům a dalším metodám. Instance objektu je potom kopie vzorové struktury popisující třídu, přesněji tedy objektové sítě. Komunikace objektů probíhá formou zasílání zpráv, která musí mít specifikovaného adresáta a předmět zprávy, kterým je nejčastěji volání metody. Při zavolání metody objektu dojde k instanciaci sítě metody, na vstupní místa jsou navázány parametry metody a jako výstup slouží místo zvané *return*. Metoda je ukončena a síť zničena v okamžiku, kdy se do místa *return* dostane jakákoliv značka, respektive objekt. (9)

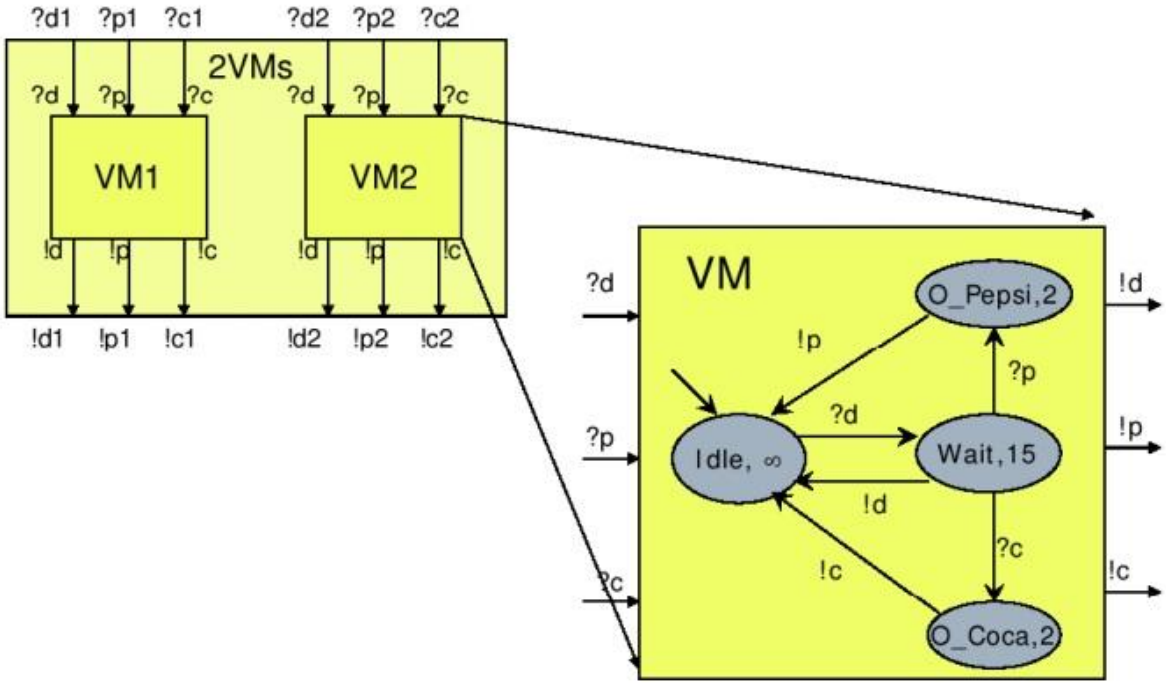
***** bude ukázka OOPN modelu ****

Petriho sítě se moc nehodí pro modelování rozsáhlých systémů, proto se užívají spíše pro modelování jeho určitých ucelených částí, které jsou dále spojovány dohromady, jak si ukážeme v následující podkapitole o formalismu DEVS.

Pro bližší seznámení s aplikací Petriho sítí v systémovém inženýrství doporučuji publikaci (10).

4.2 DEVS

DEVS (Discrete EVent System specification) je formalismus pro specifikaci modelů založený na teorii systémů. DEVS formalismus rozlišuje atomické a složené komponenty (viz. Obrázek 4.2), kde složené komponenty mohou obsahovat další složené, nebo atomické komponenty. Atomické komponenty reprezentují jeden modul s rozhraním tvořeným vstupními a výstupními *porty*. Na DEVS komponenty mohou být mapovány i jiné formalismy (např. konečné automaty, Petriho sítě, atp.), vzniká tak hierarchicky definovaný systém popsaný různými formalismy, dle vhodnosti aplikace. (4)



Obrázek 4.2 Příklad systému popsaného DEVS formalismem. Vlevo je spojovaný (hierarchický) model a vpravo je atomický model, který představuje jednu komponentu ve spojovaném modelu. Převzato z (11).

Systém (atomická komponenta také může být samostatný systém) modeluje vlastní aktivitu v podobě reakce na externí (vstupní) události externí přechodovou funkcí δ^{ext} a na interní (časově plánované) události interní přechodovou funkcí δ^{int}. Výstup systému (resp. komponenty) je závislý pouze na stavu systému (resp. komponenty). Formální popis atomického DEVS dle (11) je struktura

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, s_0)$$

kde

- X - množina vstupních hodnot (událostí)
- S - množina všech stavů systému
- Y - množina výstupních hodnot (událostí)
- $\delta_{int} : S \to S$ je interní přechodová funkce
- $\delta_{ext} : Q \times X \to S$ je externí přechodová funkce, kde $Q = \{(s, t_e) | s \in S, 0 \leq t_e \leq ta(s)\}$ je množina totálních stavů a t_e (elapsed time) je uplynulý čas od poslední události (vnitřní i vnější)
- $\lambda : S \to Y \cup \{e\}$ je interní výstupní funkce, kde e je prázdná událost
- $ta: S \to R_0^{+\infty}$ je funkce posuvu času (time advance function), kterou je při přechodu do nového stavu nastavena interní událost za daný čas
- $s_0 \in S$ - počáteční stav systému

Interpretace formalismu je poté následující: Systém se nachází ve stavu s. Pokud nenastane žádná externí událost $x \in X$, systém zůstává ve stavu s po dobu definovanou $ta(s)$. Při uplynutí času $t_e = ta(s)$ nastane vnitřní událost a provede se vnitřní přechodová funkce $\delta_{int}(s)$ a současně se vygeneruje výstupní hodnota pomocí výstupní funkce $\lambda(s)$. Pokud nastane externí událost $x \in X$ před uplynutím doby $ta(s)$, systém okamžitě přejde do stavu $s_{new} = \delta_{ext}(s, t_e, x)$. Při kolizi interní a externí události, čili v uplynulém čase $t_e = ta(s)$, má vždy přednost externí událost a interní událost se ignoruje. (11)

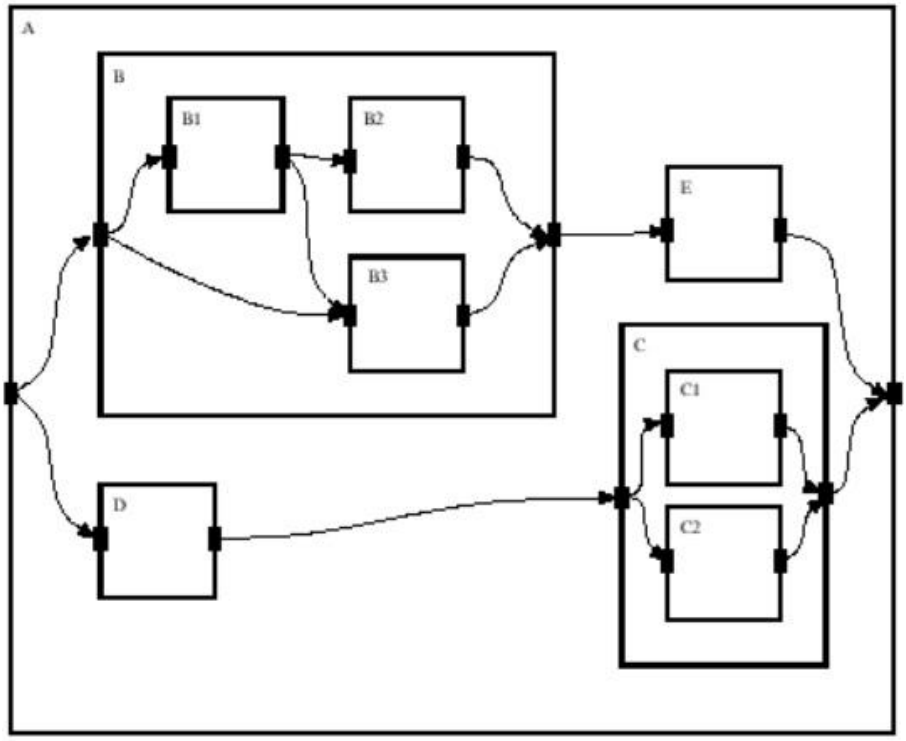
To byla definice atomického DEVS. Systémy jsou obvykle složeny z více subsystémů (komponent), které spolu interagují pomocí vstupních a výstupních portů (viz. Obrázek 4.3), přičemž stav celého systému je dán stavem jednotlivých subsystémů. Při implementaci těchto složených systémů často množinu stavů a množiny vstupních a výstupních událostí obvykle specifikujeme jako strukturované množiny, což nám umožní používat libovolný konečný počet vstupních, výstupních portů a stavových proměnných. Strukturovaná množina obsahuje n-tice složek stavu nebo události, kde složka odpovídá dílčímu stavu (resp. události) v dílčí komponentě (resp. portu). Předcházející definice základního DEVS formalismu (atomické komponenty) je potom obohacena o množinu vstupních a výstupních portů. Spojovaný model je definován propojením submodelů (atomické nebo spojované modely) a vzniká nám tak možnost *hierarchického modelování*, z čehož plyne, že množina DEVS je uzavřena vůči skládání. Spojovaný model je dle (11) formálně definován strukturou

$$CM = (X, Y, D, \{M_i\}\, EIC, EOC, IC, select)$$

kde

- X a Y jsou množiny vstupních a výstupních hodnot (událostí),
- D je neprázdná konečná množina jmen komponent,
- $\{M_i\}$ je množina komponent, pro každé $i \in D$ je M_i atomický nebo spojovaný model,
- $EIC \subseteq X \times (\bigcup_{i \in D} M_i.X)$ množina připojení externího vstupu
- $EOC \subseteq (\bigcup_{i \in D} M_i.Y) \times Y$ množina připojení externího výstupu
- $IC \subseteq (\bigcup_{i \in D} M_i.Y) \times (\bigcup_{i \in D} M_i.X)$ množina interních propojení komponent
- $select: 2^D \to D$ je funkce výběru z konfliktních komponent (při výskytu události ve stejný čas)

Dle definice je propojení komponent (vstupů a výstupů) realizováno propojením událostí, avšak při použití portů jsou množiny strukturované a propojení je možné definovat mezi porty, kdy všechny události na port p_1 jsou mapovány na stejné události na portu p_2. (11) (12)



Obrázek 4.3 Hierarchický spojovaný model DEVS. Je zde naznačeno propojení komponent pomocí vstupních a výstupních portů komponent. Dále je zde vidět možnost hierarchického skládání komponent. Převzato z (11).

DEVS umožňuje simulaci modelu formou nadřazení simulátoru samotnému modelu v hierarchii komponent a propojením s jeho rozhraním. Lze tak simulovat celý model najednou, nebo pouze jeho části. Využívá se tzv. *Experimentální rámec*, který představuje zdrojový systém dat (reálné prostředí) a je napojen na vstupy a výstupy simulovaného modelu.

Při porovnání DEVS s OO přístupem lze dle (11) zjistit následující vlastnosti:

- OO zvládá dynamický vznik/zánik objektů, DEVS to řeší komplikovaně
- OO čitelně modeluje systém tříd, DEVS naopak pracuje na úrovní instancí
- OO komunikace objektů vyžaduje referenci na adresáta, DEVS definuje komunikační vazby jako relaci nad komponentami
- Některé komponenty systému mají dlouhou životnost, jiné nikoli. Komponenty s delší životností je vhodné modelovat komponentami DEVS, zbytek potom objekty uvnitř atomických komponent.

DEVS je založen na komponentním přístupu a objekty jsou zde používány na jemnější úrovní , čili v atomických komponentách. (11)

4.3 Nástroj PNtalk/SmallDEVS

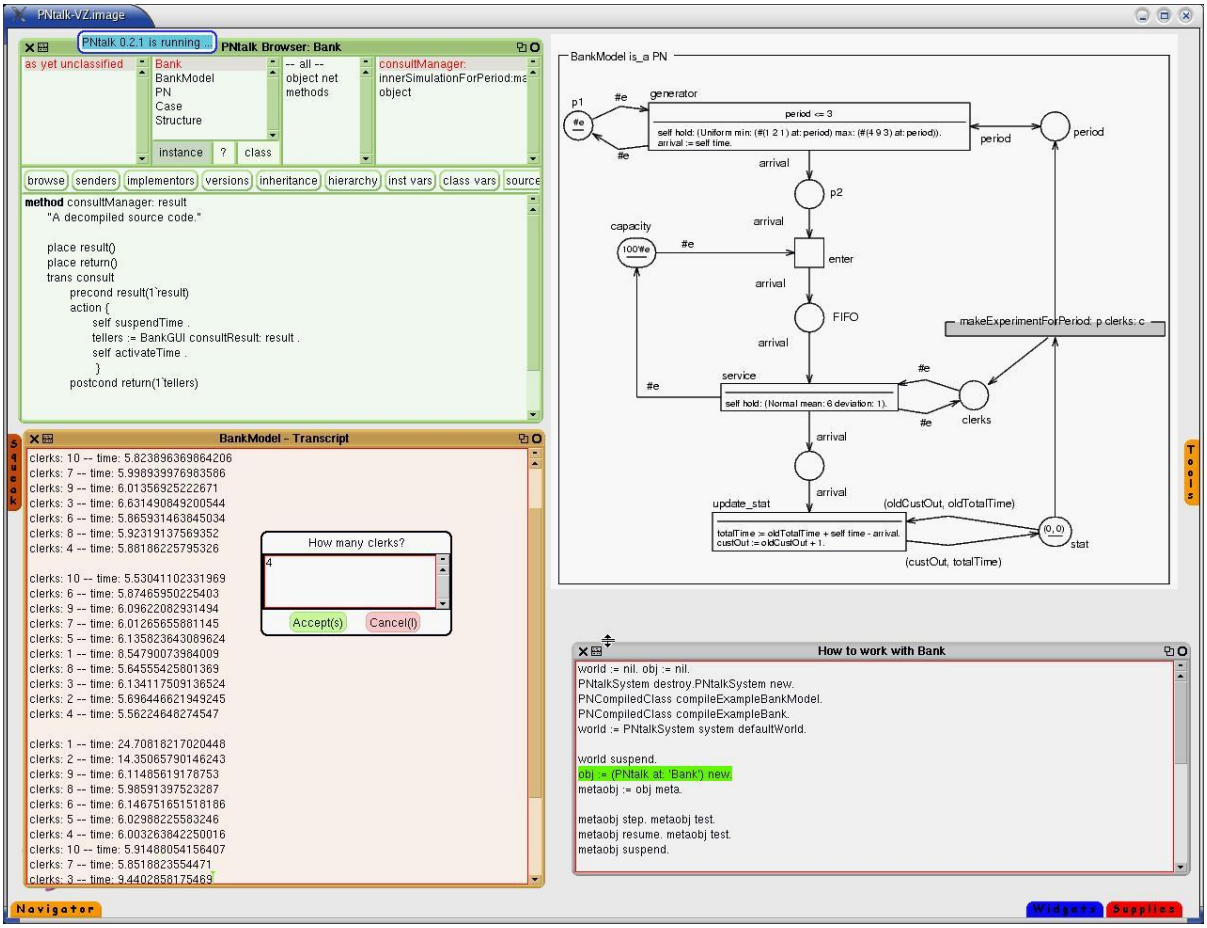
V úvodu této kapitoly jsem zmínil výzkumnou skupinu Modelování a Simulace na FIT VUT v Brně (2). Tato skupina se zabývá nejenom výzkumem v oblasti modelování a simulace, ale také vývojem

nových formalismů (viz. OOPN v kapitole 4.1) a nástrojů pro práci s nimi. Pro výše probrané formalismy to jsou nástroje *PNtalk* a *SmallDEVS*. Podobnost názvů těchto nástrojů s názvem programovacího jazyka *Smalltalk*, není náhodná, nýbrž opodstatněná, protože oba nástroje jsou založeny právě na tomto programovacím jazyku *Smalltalk*. Vývoj zmíněných nástrojů původně probíhal odděleně, nyní je však *PNtalk* integrovaný do *SmallDEVS*, respektive *SmallDEVS* slouží pro *PNtalk* jako operační prostředí. (4)

Pro stručné popsání nástrojů si dovolím doslovně citovat z (4).

"PNtalk je experimentální modelovací formalismus založený na objektově orientovaných vysokoúrovňových Petriho sítích (OOPN). Kombinuje vlastnosti Petriho sítí s výhodami objektově orientovaného návrhu systémů. Původním záměrem projektu PNtalk bylo prověřit možnost přímého použití matematického formalismu pro programování. Uvažuje se o aplikacích v oblasti workflow, plánování, logistiky a řízení."

PNtalk umožňuje efektivní práci s OOPN a kromě aplikace v modelování a simulace se zaměřuje hlavně na možnost začlenění modelu do reálného prostředí. Takto pojatý model může sloužit jako část prototypu, případně přímo aplikace. Díky struktury a principu vytváření objektů v OOPN popsaném v 4.1 je možné zavést dynamické chování modelu. Dynamičností se rozumí změna struktury, nebo chování modelu za běhu. Tím že objekty (instance) jsou kopií vzorového objektu (třídy), je možné měnit buď chování objektu samotného, nebo změnit vzor třídy, přičemž každá další vytvořená instance této třídy bude kopírovat novou strukturu vzoru, ale instance již vytvořené zůstanou nezměněny. *PNtalk* pro možnost dynamických změn využívá principy otevřené implementace, zejména metaúrovňových architektur (4). Základem otevřené implementace je umožnění aplikaci nahlédnout až za definované rozhraní objektu, tedy do vnitřních aspektů, které může ovlivňovat. Metaúroveň je potom chápána tak, že pro každý klasický objekt existuje nějaký jiný metaobjekt, který svým rozhraním umožňuje inspekci a změnu aspektů daného klasického objektu. Obecně je metaúrovňová architektura rozdělena do různých úrovní, kde každá úroveň je řízená jinou úrovní (metaúrovní z pohledu řízené úrovně). Z komplexnějšího hlediska je potom metaúroveň celého systému, nebo jeho podsystémů, nazývána *metasystém*. Nástroj *PNtalk* je založen na co největší otevřenosti s využitím reflektivních a metaúrovňových architektur (4).

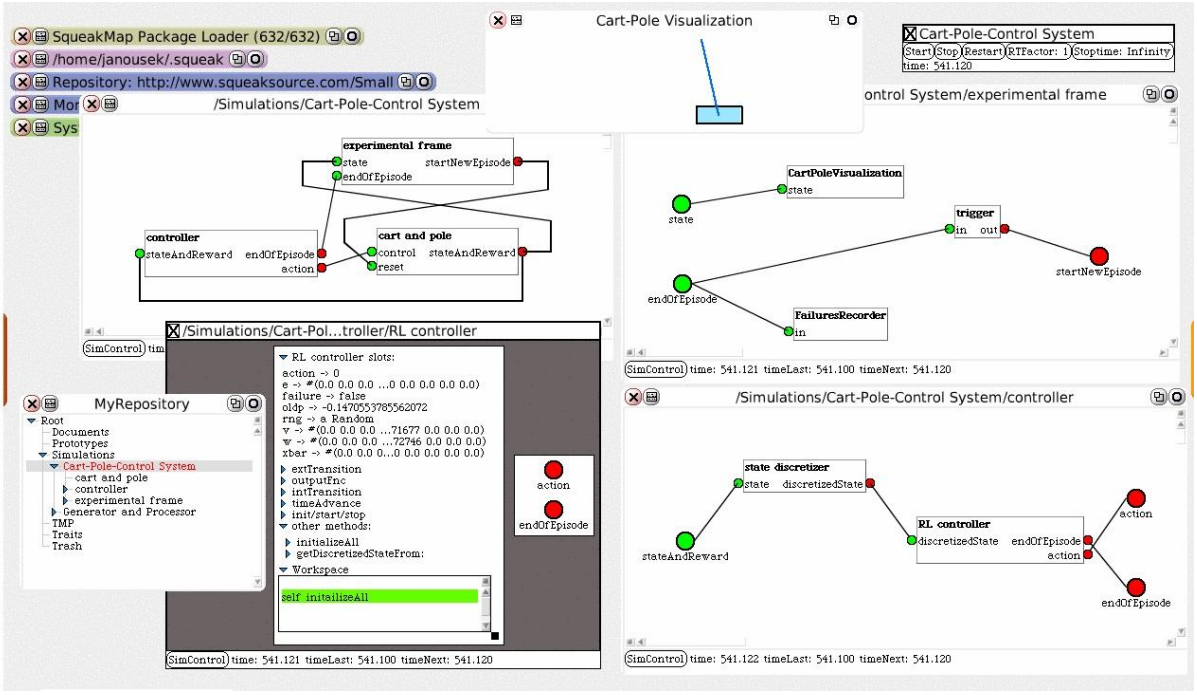


Obrázek 4.4 Ukázka prostředí nástroje *PNtalk*. Vpravo je graficky znázorněna programovaná Petriho síť, definující bankovní model. Převzato z (4).

Zatímco *PNtalk* je nástroj pro modelování OOPN, tak *SmallDEVS* je nástroj pro modelování a simulaci komplexních systému a je prezentován skupinou Modelování a Simulace následujícím odryvkem z (4):

"SmallDEVS je experimentální implementace formalismu DEVS, zaměřená obecně na modelování a prototypování, ze zvláštním důrazem na aplikace v oblasti adaptivních a inteligentních systémů. Umožňuje vývoj struktur i chování komponent za běhu, propojování simulovaných a reálných systémů a kombinaci s jinými formalismy, což je v těchto aplikacích nezbytné. V současné době SmallDEVS slouží jako operační systém pro PNtalk – poskytuje mu vývojové a prováděcí prostředí, uložiště modelů, vstupy-výstupy a komponentní zapouzdření."

SmallDEVS je tedy dynamický, objektově orientovaný jazyk s otevřenou implementací (popsáno výše v této podkapitole) založený na Zieglerově formalismu DEVS (8). Modelování pomocí *SmallDEVS* je možné dvěma způsoby, kdy první je využití standardních nástrojů jazyka Smalltalk a knihovny tříd pro podporu DEVS formalismu (obsahuje třídy jako AtomicDEVS nebo CompositeDEVS) a druhý je využití specializovaného GUI poskytovaného jazykem *SmallDEVS*, které je prozatím poněkud nestabilní, tudíž je tato možnost riskantnější. V obou případech je třeba ctít zásady DEVS (viz. 4.2).



Obrázek 4.5 Ukázka prostředí nástroje *SmallDEVS*. Vlevo lze vidět propojení DEVS komponent, které jsou vpravo specifikovány. Převzato z (4).

Vývoj těchto nástrojů je zaměřen na modelem a simulací řízený vývoj systémů, přičemž je kladen důraz na ponechání modelu a možnosti jeho simulace až do poslední chvíle, čili k aplikačnímu nasazení (*model continuity*). Z toho vyplývá další směr výzkumu a to optimalizace struktury modelů, transformace modelů do výkonnější podoby, případně možnost svázání s jiným programovacím jazykem. (4)

Vyvíjené nástroje jsou testovány na případových studiích jako třeba: konferenční systém řízený modelem toku dat (workflow), plánování v agentních systémech, řízení robotických systémů a modelování a simulace senzorových sítí. (4)

5. Realizace modelu konferenčního systému

5.1 Návrh tříd

5.2 Konverze diagramu tříd do objektově orientovaných Petriho sítí

6. Testování/simulace

6.1 Návrh testů

6.2 Výsledky testů

7. Napojení uživatelského rozhraní

8. Závěr

V této práci jsem se zabýval *modelem řízeným návrhem* počítačových systémů. Stručně jsem prošel dosavadní historií vývoje softwarového inženýrství, a poté jsem se detailněji zaměřil na techniku modelem řízeného návrh (*MBD*), přesněji metodiku *MDA* (Model Driven Architecture) specifikovanou skupinou OMG (viz. (1)). MBD odstraňuje hlavní nevýhody klasických návrhových metod, které jsou nekonzistentnost návrhových modelů s výslednou implementací a nemožnost testovat a validovat systém ve fázi návrhu, tedy tvorby modelu systému. Návrh technikou *MBD* je založen na formálně popsaných funkčních modelech, díky kterým je možné systém prototypovat a testovat již ve fázi návrhu. Překlad na odpovídající implementační kód je oproti klasických metod, kdy implementační tým přepisuje model manuálně řádově několik měsíců, prováděn automaticky pomocí mapovacích pravidel odpovídajících dané cílové platformě, nebo je místo transformace funkční model zachován v aplikaci i při reálném nasazení (*model continuity*). Konkrétně u *MDA* je návrhový model popsán jazykem *xUML*, který je vytvořen z notačního modelovacího jazyka *UML*, přičemž odstraňuje sémanticky slabé elementy a zbylým přidává striktně specifikovanou syntaxi a sémantiku.

V další kapitole jsem probral obecný význam využití modelování a simulací ve vývoji systémů a poukázal jsem na výzkum probíhající na fakultě informačních technologií na VUT v Brně (viz. stránky výzkumné skupiny (2)), který je zaměřen na využití formalismů jako *DEVS* a *OOPN* (Objektově orientované Petriho sítě), vývoj nástrojů pro práci s těmito formalismy a aplikaci dříve zmiňovaného *model continuity*. Podrobněji jsem se seznámil s formalismem *DEVS*, který umožňuje hierarchické skládání podsystémů (komponent) v jeden komplexní událostně řízený diskrétní systém. Komponenty zde mohou být opět spojované, nebo atomické, kde se mimo jiných formálních popisu chování, jako například konečné automaty, uplatní hlavně formalismus *OOPN*, který spojuje výhody objektového programování a formálního matematického aparátu Petriho sítí. Teoreticky jsem se seznámil s vyvíjenými nástroji *SmallDEVS/PNtalk*, které umožňují programování modelu pomocí zmíněných formalismů, s důrazem na dynamičnost systému a možnost reálného nasazení modelu, respektive propojení simulovaných a reálných subsystémů, a kombinaci s jinými formalismy.

Doposud jsem provedl teoretickou přípravu pro modelem řízený návrh konferenčního systému, který je hlavním cílem mé diplomové práce. Pomocí uvedených formalismů a nástrojů budu vytvářet konferenční systém v základní funkčnosti. Účelem mé diplomové práce je na statickém, deterministickém informačním systému demonstrovat techniky vývoje zkoumané výzkumnou skupinou Modelování a Simulace při fakultě FIT VUT v Brně. Tento proces vývoje poté zhodnotím a porovnám s klasickou metodou vývoje informačních systémů.

Bibliografie

1. **Object Management Group, Inc.** MDA. *Object Management Group.* [Online] 30. 10 2012. [Citace: 10. 12 2012.] http://www.omg.org/mda/.

2. **Kočí, R., a další.** Modelování a simulace. *Modelování a simulace.* [Online] [Citace: 11. 12 2012.] http://perchta.fit.vutbr.cz/research-modeling-and-simulation.

3. **Object Management Group, Inc.** UML Resource Page. *Object Management Group - UML.* [Online] 28. 11 2012. [Citace: 10. 12 2012.] http://www.uml.org/.

4. **Kočí, R. a Janoušek, V.** Simulace a vývoj systémů. *Modelování a simulace.* [Online] 17. 9 2010. [Citace: 11. 12 2012.] http://perchta.fit.vutbr.cz/research-modeling-and-simulation/3.

5. **Raistrick, Ch., a další.** *Model Driven Architecture with Executable UML.* Cambridge : Cambridge University Press, 2004. ISBN 0 521 53771 1.

6. **Zendulka, Jaroslav.** Projektování programových systémů. [Online] [Citace: 10. 12 2012.] http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/11_2.pdf.

7. **Patočka, Miroslav.** *Řešení IS pomocí Model Driven Architecture.* [pdf] Brno : Masarykova Univerzita, 2008. Diplomová práce.

8. **Ziegler, B.P., Preahofer, H. a Kim, T.G.** *Theory of Modeling and Simulation: Integrating discrete event and continuous complex dynamic systems.* Vyd. 2. San Diego : Academic Press, 2000. ISBN 01-277-8455-1.

9. **Janoušek, V.** Modelování objektů Petriho sítěmi. *Disertační práce.* Brno : Fakulta informačních technologií, Vysoké učení technické v Brně, 2008. ISBN 978-80-214-3747-4.

10. **Girault, C. a Valk, R.** *Petri Nets for Systems Engineering. A Guide to Modeling, Verification, and Applications.* Berlin : Springer, 2003. ISBN 3-560-41217-4.

11. **Janoušek, V.** Systémy s diskrétními událostmi, formalismus DEVS. *Prezentace k přednáškám.* [pdf].

12. **Peringer, P.** Simulační nástroje a techniky. *Prezentace k přednáškám.* [pdf]. Brno : Fakulta informačních techologií, Vysoké učení technické v Brně, 2011.

13. Objektová Petriho síť. *akela.mendelu.cz.* [Online] [Citace: 11. 12 2012.] https://akela.mendelu.cz/~petrj/opnml/opn.html.