VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

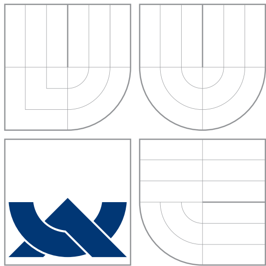
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÝ EDITOR SIMULAČNÍCH MODELŮ

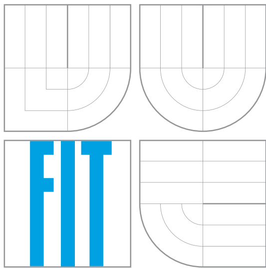
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAN HOŘÁK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÝ EDITOR SIMULAČNÍCH MODELŮ

GRAPHICAL EDITOR OF SIMULATION MODELS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN HOŘÁK
AUTHOR

VEDOUCÍ PRÁCE — Dr. Ing. PETR PERINGER
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2007/2008

Zadání diplomové práce

Řešitel: **Hořák Jan, Bc.**
Obor: Inteligentní systémy
Téma: **Grafický editor simulačních modelů**
Kategorie: Modelování a simulace

Pokyny:

1. Prostudujte problematiku modelování a simulace se zaměřením na formalismus DEVS. Prostudujte problematiku grafických editorů a jazyk XML.
2. Navrhněte grafický editor modelů který umožní interaktivní zadání DEVS modelu a jeho ukládání v XML. Navrhněte rozhraní pro moduly umožňující převod XML reprezentace do cílového simulačního jazyka a moduly pro řízení simulace.
3. Implementujte editor modelů v C++ a ověřte jeho funkčnost na vhodných příkladech.
4. Zhodnoťte výsledky práce a navrhněte možná vylepšení.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění prvních dvou bodů zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVR-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Peringer Petr, Dr. Ing.**, UITS FIT VUT
Datum zadání: 5. února 2007
Datum odevzdání: 19. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Jan Hořák**
Id studenta: 29684
Bytem: Otakara Ševčíka 64, 636 00 Brno
Narozen: 19. 09. 1982, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Grafický editor simulačních modelů
Vedoucí/školitel VŠKP: Peringer Petr, Dr. Ing.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

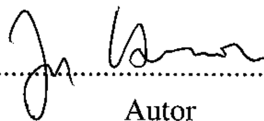
1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....



....	....
Nabyvatel	Autor

Abstrakt

Tato práce obsahuje úvod do problematiky modelování a simulace pomocí formalismu DEVS. Definuje základní modely (atomický DEVS a složený DEVS), určuje způsob simulace a obsahuje příklady formalismů, které jsou na DEVS založeny (ParallelDEVS, DESS). Dále je popsán způsob zadávání DEVS modelu pomocí grafického editoru, stručný přehled existujících nástrojů a jejich výhody a nevýhody. Zvláštní část je určena k využití jazyka XML pro ukládání modelů a přibližuje formy validace a transformace XML dokumentů. Poté práce obsahuje návrh grafického editoru simulačních modelů, který specifikuje třídy použité pro reprezentaci modelu, pracovní plátno, rozhraní pro export modelu a hlavní aplikaci, inspirované návrhovými vzory. Rovněž je uveden formát XML dokumentu určeného k uložení DEVS modelu a jednoduchý DEVS simulátor. Implementační část se zabývá použitými knihovnami a prací s nimi. Vymezuje jejich klady a zápory. Dokument je uzavřen příkladem návrhu a implementace jednoduchého DEVS modelu ve vytvořeném grafickém editoru určeného pro DEVS modely.

Klíčová slova

modelování, simulace, DEVS formalismus, atomický DEVS, složený DEVS, simulace DEVS modelu, paralelní DEVS, uživatelské rozhraní, plátno, grafický editor simulačního modelu, XML, DTD, XSL, XSLT, návrhový vzor, dynamický modul

Abstract

This paper contains brief introduction into modeling and simulation using Discrete Event Specified System (DEVS) formalism. It defines basic models (atomic and coupled DEVS) and shows how they are simulated. Examples of derived DEVS formalism like parallel DEVS or DESS are also presented. It is descibed how to create DEVS models using graphic modeling software and advantages and disadvantages of this approach. A short summary of known programs are also covered. Storing models in the XML language, validation of XML document and transformation capabilities by XSLT are discused. The main section is dedicated to the design of a graphic editor for simulation models inspired by design patterns including classes for canvas, model representation, export module interface and main application. The XML document used for storing DEVS models and simple DEVS simulator are also described. Implementation section presents used programming libraries, reasons why they have been used and their advantages and disadvantages. Paper ends with an example of a simple DEVS model created by implemented graphic editor for simulation DEVS models.

Keywords

modeling, simulation, DEVS formalism, atomic DEVS, coupled DEVS, simulation of DEVS, parallel DEVS, user interface, canvas, modeling using graphical editor, XML, DTD, XSL, XSLT, design pattern, dynamic module

Citace

Jan Hořák: Grafický editor simulačních modelů, diplomová práce, Brno, FIT VUT v Brně, 2008

Grafický editor simulačních modelů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Jan Hořák
17. května 2008

Poděkování

Rád bych poděkoval panu Dr. Ing. Petru Peringerovi za vedení, rady a připomínky, kterými mně během práce pomáhal. Děkuji rovněž rodičům a přátelům za materiální a morální podporu během psaní této práce.

© Jan Hořák, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Modelování a simulace s využitím grafických editorů **5**
2.1 Problematika modelování a simulace . . . 5
2.2 Modely a jejich reprezentace . . . 6
2.3 DEVS formalismus . . . 7
2.3.1 Discrete Event System Specification (DEVS) . . . 7
2.3.2 Discrete Event Specified Network (DEVN) . . . 10
2.3.3 Paralelní DEVS . . . 13
2.3.4 Differential Equation System Specification (DESS) . . . 13
2.4 Simulace DEVS a DESS formalismů . . . 15
2.4.1 Simulátory DEVS/DEVN . . . 15
2.4.2 Výhody a nevýhody použití DEVS formalismu . . . 20
2.5 Grafické editory pro zadávání modelů . . . 20
2.5.1 Historie vývoje grafického uživatelského rozhraní . . . 21
2.5.2 Uživatelské rozhraní aplikací . . . 21
2.5.3 Pozice a způsob ovládání prvků uživatelského rozhraní . . . 22
2.5.4 Požadavky . . . 22
2.5.5 Přehled nástrojů . . . 22
2.5.6 Možná vylepšení . . . 24
2.5.7 Výhody a nevýhody použití . . . 25
2.6 XML a jeho použití v grafických editorech modelů . . . 25
2.6.1 Stavební bloky XML dokumentu a jejich uspořádání . . . 26
2.6.2 Možnosti definice struktury XML dokumentu . . . 27
2.6.3 Zpracování XML dokumentu . . . 29
2.6.4 Použití XML v grafických editorech . . . 31

3 Návrh grafického editoru simulačních modelů **32**
3.1 Model . . . 32
3.1.1 Třídy pro reprezentaci modelu . . . 32
3.2 Pracovní plátno . . . 34
3.2.1 Třídy objektů plátna . . . 34
3.2.2 Odvozené třídy pro potřeby DEVS modelu . . . 36
3.2.3 Chování pracovního plátna . . . 36
3.2.4 Třídy pro vlastní plátno . . . 38
3.2.5 Třídy pro podporu vracení změn . . . 39
3.2.6 Možnosti využití pro jiné druhy modelů . . . 40
3.3 Návrh formátu souboru . . . 40

3.3.1 Knihovna modelů . . . 42
3.4 Hlavní aplikace . . . 42
3.4.1 Operace se soubory . . . 42
3.4.2 Export modelu . . . 43
3.4.3 Správa nastavení . . . 44
3.4.4 Grafické uživatelské rozhraní . . . 44
3.5 Rozhraní pro řízení běhu simulace . . . 44
3.6 DEVS simulátor . . . 45

4 Implementace editoru simulačních modelů 47
4.1 Knihovna pro prvky uživatelského rozhraní . . . 47
4.2 Výběr knihovny pro implementaci pracovního plátna . . . 47
4.3 Knihovna pro implementaci uživatelského rozhraní . . . 48
4.4 Čtení a zápis DEVS modelů pomocí knihovny . . . 48
4.5 Moduly pro export modelu . . . 49
4.6 Transformace modelu pomocí XSLT . . . 50
4.7 Vzhled uživatelského rozhraní . . . 50
4.8 Příklad řešení problému . . . 51
4.8.1 Generátor . . . 51
4.8.2 Zařízení s frontou . . . 52
4.8.3 Monitorovací blok . . . 54
4.8.4 Propojení bloků a simulace . . . 55
4.8.5 Výsledky . . . 56

5 Závěr 58

Kapitola 1

Úvod

Pod pojmem počítačová simulace chápeme počítačový program, který se pokouší simulovat chování zkoumaného systému. Nachází dnes uplatnění napříč spektrem oborů, například ve fyzice, chemii, biologii, strojírenství, ekonomice nebo sociologii. Používá se tam, kde poznatky není možné získat analyticky nebo reálnými experimenty, kvůli časové náročnosti nebo finančním důvodům. Poznatky získané simulací můžeme využít k srovnávání, optimalizaci, odhadu možných rizik, předvídání událostí nebo objevení nových souvislostí.

Cílem této práce je vytvořit obecné prostředí pro zadávání a simulaci modelů definovaných blokovými schématy nebo jinou podobnou hierarchickou grafickou reprezentací. Výsledný program by pak bylo možné používat například jako prostředí pro zadávání DEVS modelů a pro převod do některého simulačního jazyka.

První část této práce se zabývá především teorií, která s modelováním a simulací pomocí DEVS formalismu souvisí. Úvodem je zmíněna problematika, základní rozdělení, reprezentace a postupy v modelování a simulaci. Kapitola pokračuje definicí atomického DEVS, popisem jeho struktur a demonstruje důsledky jeho chování. Příklad modelu pak slouží k ujasnění souvislostí a významu přechodů stavu během simulace. Pro podporu skládání DEVS modelů je uvedena definice sítě složeného DEVS a význam portů spolu s vysvětleným příkladem modelu. K rozšíření znalostí jsou uvedeny dva formalismy, které modifikují základní definici. Parallel DEVS slouží k simulaci systémů s paralelním výskytem událostí a DESS slouží k podpoře simulací systémů specifikovaných diferenciálními rovnicemi. Následuje část, která pojednává o simulaci DEVS modelů, nahrazení atomických DEVS modelů simulátory a složených DEVS modelů koordinátory. Jsou popsány druhy zpráv, kterými koordinátory a simulátory mezi sebou komunikují a reakce na ně. Kapitolu uzavírá zhodnocení DEVS formalismu z hlediska výkonu, vytváření a modifikace modelů.

V následující kapitole je shrnuta historie vzniku uživatelských rozhraní a požadavky na grafické editory simulačních modelů. Rovněž je obsaženo srovnání několika existujících editorů určených pro DEVS modely včetně jejich výhod a nedostatků. Kapitola je uzavřena navrženými vylepšeními a úvahou kdy zvolit grafický editor simulačních modelů a kdy ne.

Teoretickou část práce uzavírá pojednání o využití jazyka XML pro ukládání datových struktur použitých v grafických editorech modelů. Obsahuje stručný úvod do jazyka XML a jeho stavebních bloků. Rozebírá možnosti pro stanovení struktury dokumentu pro potřeby validace dokumentu jazykem XSL nebo souboru s DTD a způsob jakým lze dokument zpracovat pomocí XSLT šablon.

Návrh samotné aplikace je uveden v další kapitole. Nejdříve je popsána definice modelu, jeho složek a návrhového vzoru použitého pro jeho konstrukci. Třídy určené pro pracovní plátno (*canvas*), které je ústředním objektem celé aplikace, zahrnují objekty umístitelné na

plátno, chování při výskytu vnějších událostí a vlastního plátna s podporou funkce *Zpět*. Je uvedena možnost použít pracovní plátno pro jiné druhy modelů. Popis tříd je proložen použitými návrhovými vzory, které implementaci usnadňují. Dále je uveden návrh vhodného formátu pro uložení modelu do souboru při využití výhod jazyka XML, knihovna modelů, kterou aplikace využívá a její funkce, třídy popisující hlavní chování, moduly pro export a návrh DEVS simulátoru použitého pro jejich testování.

Část věnovaná implementaci popisuje zvolené prostředí a knihovny, výhody jejich použití a nevýhody, obtíže při vytváření aplikace a popis vytváření vlastních exportních modulů s využitím XSLT šablon nebo jazyka C++. Práci uzavírá příklad jednoduchého DEVS modelu vytvořeného pomocí vzniklé aplikace a simulovaného některou simulační knihovnou.

Kapitola 2

Modelování a simulace s využitím grafických editorů

Chápeme-li grafický editor jako prostředek pro snadné zobrazení, vytváření a modifikaci struktury podobné grafu, je možné jej použít pro produkci simulačních modelů [9]. Pomocí grafického editoru však nemusí být výhodné definovat všechny známé typy modelů. Proto je důležité rozlišit, pro které modely jsou grafické editory vhodné a pro které nikoliv. V následující části probereme známé reprezentace modelů a určíme u které je vhodné grafickými editory konstruovat. Dále popíšeme z čeho by se měl správný grafický editor modelů skládat. Určíme pracovní postupy pro vytvoření a modifikaci modelu, způsob uložení modelu a technologie, které slouží pro převod grafické reprezentace do simulačních jazyků.

2.1 Problematika modelování a simulace

Základním principem modelování a simulace je vytvořit takový systém, ke kterému bude původní systém homomorfní. Systém $S_1 = (U_1, R_1)$ je homomorfní se systémem $S_2 = (U_2, R_2)$ pokud je prvkům univerza U_1 možné přiřadit jednoznačně prvky z univerza U_2 (opačně to platit nemusí – jedná se o násobnost vztahu N:1). Dále pak je možné prvkům charakteristiky R_1 jednoznačně přiřadit prvky charakteristiky R_2 tak, že prvku charakteristiky R_1 je vždy přiřazen ten prvek charakteristiky R_2, který vyjadřuje stejně orientovaný vztah mezi odpovídající dvojicí prvků univerza U_2.

Neformálně řečeno – jde nám o to, abychom systém, který hodláme simulovat, dostatečně zjednodušili pro účely, ke kterým bude výsledek simulace sloužit. Samotný proces vytváření simulovaného systému a jeho simulace lze rozdělit do pěti kroků:

Získávání znalostí: Nejdříve se zabýváme studiem systému, který budeme simulovat. Experimentujeme a pozorujeme, zaměříme se především na parametry, které chování systému ovlivňují nejvíce, vliv méně podstatných veličin zanedbáme.

Modelování: Ze získaných znalostí vytváříme abstraktní model pomocí některého známého postupu. Vztah mezi pozorovaným systémem a jeho modelem je většinou *homomorfní* (násobnost N:1).

Vytváření simulačního modelu: Abstraktní model transformujeme na simulační model. Vztah mezi abstraktním a simulačním modelem je *izomorfní* (násobnost 1:1). Z praktického hlediska jde o přepis modelu do některého programovacího jazyku nebo

přenesení modelu do simulačního prostředí. Možnosti řešitele simulační úlohy jsou proto omezeny jeho praktickými zkušenostmi s prostředím, v němž model vytváří. Po přenesení modelu následuje *verifikace*, což je proces, kterým zjišťujeme, zda se simulační model chová stejně jako jeho abstraktní předloha.

Experimentování: V další fázi přednastavíme v modelu počáteční podmínky, vystavujeme simulační model různým vstupním podmínkám, porovnáváme výsledky simulačního procesu, sledujeme výstupní data a docházíme k novým závěrům. Tento krok je nutné několikrát opakovat a volit vhodné vstupní hodnoty.

Interpretace výsledků: Na závěr zhodnocujeme výsledky a vyvozujeme z nich nové závěry. V neposlední řadě zjišťujeme, zda je model *validní*, tj. zda výsledky simulace odpovídají skutečnosti. Pokud tomu tak není, musíme celý postup opakovat, výsledky jsou totiž chybné.

2.2 Modely a jejich reprezentace

Model je prostředek, kterým popisujeme zjednodušený systém. Jedním z hledisek, jak můžeme na rozdělení modelů do kategorií pohlížet, je podoba výsledného návrhu [6]:

Konceptuální modely jsou nejvolnějším vyjádřením používaným k popisu systémů, u kterých prozatím nedošlo z hlediska teorie k přesnému popisu. Jsou vhodné především v rané fázi návrhu k ujasnění základních souvislostí. Vyjadřovacím prostředkem bývá text nebo obrázek.

Deklarativní modely se skládají ze stavů a přechodů mezi nimi. Přechody definují nutné podmínky k uskutečnění události a nový stav po jejich vzniku. Proto se hodí pro diskrétní modely, ve kterých dochází ke změnám skokově. K zápisu se používají konečné automaty, Petriho sítě, soupis událostí řízených kalendářem nebo DEVS formalismus.

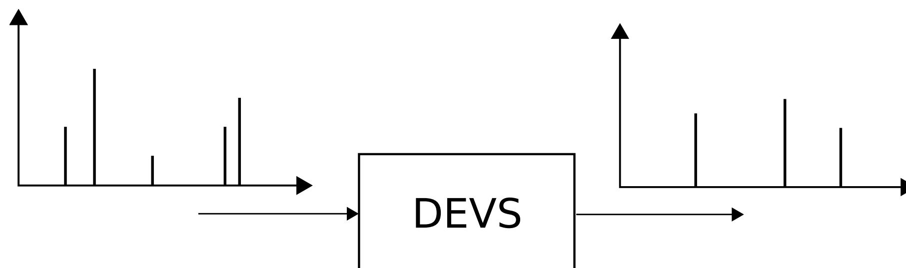
Funkcionální modely používají k reprezentaci orientovaný graf. Uzly chápeme jako funkce nebo proměnné, hrany pak tvoří síť, která určuje vzájemné propojení. Funkcionální modely jsou výhodné pro systémy hromadné obsluhy, spojitou simulaci znázorněnou pomocí blokových schémat a rovněž DEVS modely.

Modely popsané rovnicemi algebraickými, diferenciálními nebo diferenčními lze poměrně snadno převést na blokové schéma funkcionálního modelu nebo pro jejich simulaci slouží speciální metody.

Prostorové modely dělí prostor na objemově menší podsystémy, které se řeší odděleně, případně distribuovaně.

Multimodely vznikají kombinací výše uvedených modelů. Časté je spojení diskrétního a spojitého modelu v situacích, kdy potřebujeme skokovou změnu některého z parametrů. V praxi jsou multimodely nejrozšířenější, protože málokdy narazíme na složitější systém, který lze popsat pouze některým z modelů.

Z kategorií je patrné, že pomocí grafických editorů můžeme popsat poměrně široké spektrum modelů. Deklarativní modely mezi ně určitě patří. Petriho síť, konečné automaty i funkcionální modely lze zobrazit pomocí orientovaného grafu. Modely popsané rovnicemi



Obrázek 2.1: Chování DEVS modelu, vlevo vstup, vpravo výstup

je možné rovněž převést do grafické reprezentace (je ovšem dobré zvážit, zda je to výhodné). DEVS formalismus se týká všech těchto vhodných modelů pro grafickou reprezentaci a tato práce je na něj zaměřena. Proto je vhodné shrnout jeho definici.

2.3 DEVS formalismus

Vytváříme-li modely systémů, můžeme zjistit, že byť slouží svému účelu dobře, nejsou příliš vhodné pro to, abychom je mohli využít znovu. Navíc komplexní modely není snadné ladit ani udržovat, vynaložíme řadu prostředků a času pro jejich výrobu. K efektivnějšímu využití předchozích modelů může sloužit dekompozice modelu na podmodely a ty pak využít například v jiných modelech. Právě pro tyto účely vznikl DEVS formalismus, který definuje jednotné rozhraní podmodelů (bloků) a umožňuje hierarchicky skládat jednotlivé DEVS modely do sebe a vzájemně je propojovat. To přináší různou úroveň abstrakce, která je rovněž výhodná. DEVS (Discrete EVent System specification) je formalismus pro popis komplexních diskrétních, spojitých nebo kombinovaných systémů [1]. Byl představen Bernardem Zeiglerem v 70. letech [18].

2.3.1 Discrete Event System Specification (DEVS)

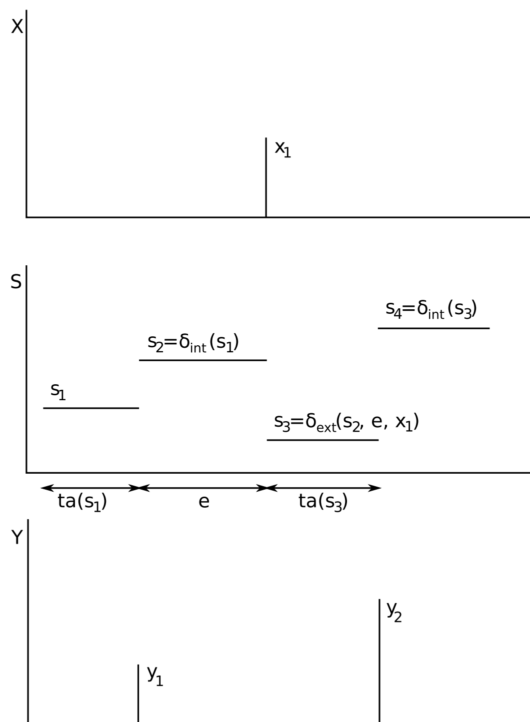
Pomocí DEVS můžeme popsat všechny systémy jejichž vstupní a výstupní chování lze popsat sekvencí událostí a stav prochází konečným počtem změn za konečný čas. Událost reprezentuje okamžitou změnu nějaké části systému. Je dána hodnotou, která může být libovolným datovým typem (číslo, vektor nebo obecně prvek množiny), a časem výskytu. DEVS model zpracuje vstupní události. Podle nich a vnitřního stavu systému vygeneruje výstupní události (viz obrázek 2.1). Chování DEVS modelu se tímto značně podobá chování konečného automatu.

Definice

DEVS model definujeme jako následující strukturu:

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

kde
- X je množina vstupních událostí
- Y je množina výstupních událostí
- S je množina vnitřních stavů



Obrázek 2.2: Chování DEVS modelu

δ_{int}, δ_{ext}, λ a ta určují chování systému (příklad chování demonstruje obrázek 2.2)

Pro stav $s \in S$ existuje posun času (*time advance*) definovaný funkcí posuvu času ta:

$$ta(s) : S \rightarrow \mathbb{R}^{+}_{0,\infty}.$$

Toto kladné číslo určuje, jak dlouho systém setrvá v daném stavu v případě, že nedojde k vnější události. Pokud tedy systém změní stav na s_1 v čase t_1, tak v čase $t_1 + ta(s_1)$ systém provede interní přechod a přejde do stavu s_2. Nový stav je vypočítán funkcí

$$s_2 = \delta_{int}(s_1).$$

Funkce

$$\delta_{int} : S \rightarrow S$$

se nazývá interní přechodovou funkcí. Pokud dojde ke změně události s_1 na s_2 vznikne výstupní událost $y_1 = \lambda(s_1)$. Funkce

$$\lambda : S \rightarrow Y)$$

se nazývá výstupní funkcí. Těmito funkcemi $(ta, \delta_{int}, \lambda)$ je definováno autonomní chování DEVS modelu.

V případě, že dojde k výskytu vnější události, stav systému se změní okamžitě. Nový stav nezávisí pouze na předchozím stavu systému a příchozí události, ale také na době, která uplynula od výskytu poslední události. Pokud je systém ve stavu s_2 v čase t_2 a dorazí-li vstupní událost v čase $t_2 + e < ta(s_2)$ s hodnotou x_1, tak nový stav systému je dán vztahem

$$s_3 = \delta_{ext}(s_2, e, x_1).$$

Funkce

$$\delta_{ext} : S \times \mathbb{R}_0^+ \times X \rightarrow S$$

se nazývá externí přechodovou funkcí a pokud v systému dojde k externí události, která změní stav systému, došlo k externímu přechodu.

Tento model se obvykle označuje jako atomický DEVS model. Atomické DEVS modely se skládají dohromady pomocí DEVN (viz 2.3.2).

Příklad modelu

Uvažujme osmibitový čítač s vlastním hodinovým signálem s periodou t_{clk} a možností resetu (který lze aktivovat asynchronně a který zároveň resetuje i hodinový signál). DEVS model popisující tento integrovaný obvod bude následující:

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

kde

X je jednoprvková množina ($X = [1]$) reprezentující aktivní signál reset

Y je množina výstupních událostí obsahující celá čísla v intervalu $(0, 255)$ S je množina stavů systému obsahující celá čísla v intervalu $(0, 255)$

$\delta_{int}(s) = (s + 1)\ mod\ 256$

$\delta_{ext}(s, e, 1) = 0$

$\lambda(s) = s$

$ta(s) = t_{clk}$

Uvažujme počáteční podmínky $s = 0$, $t = 0$, $t_{clk} = 3$ a dále, že v čase $t = 7$ dojde k resetu obvodu.

V čase $t = 0$
- $s = 0$
- $e = 0$
- $ta(s) = ta(0) = t_{clk} = 3$

V čase $t = 3^-$
- $s = 0$
- $e = 3$

V čase $t = 3$
- výstupní událost o hodnotě $\lambda(s) = \lambda(0) = 0$
- $s = \delta_{int}(0) = (0 + 1)\ mod\ 256 = 1$

V čase $t = 3^+$
- $s = 1$
- $e = 0$
- $ta(s) = ta(1) = 3$

V čase $t = 6^-$
- $s = 1$
- $e = 3$

V čase $t = 6$
- výstupní událost o hodnotě $\lambda(s) = \lambda(1) = 1$
- $s = \delta_{int}(1) = (1 + 1)\ mod\ 256 = 2$

V čase $t = 6^+$
- $s = 2$
- $e = 0$

$ta(s) = ta(2) = 3$
V čase $t = 7^-$
$s = 2$
$e = 1$
V čase $t = 7$
$s = \delta_{ext}(s, e, x) = \delta_{ext}(2, 1, 1) = 0$
V čase $t = 7^+$
$s = 0$
$e = 0$
$ta(s) = ta(0) = 3$
V čase $t = 10^-$
$s = 0$
$e = 3$
V čase $t = 10$
výstupní událost o hodnotě $\lambda(s) = \lambda(0) = 0$
$s = \delta_{int}(0) = (0 + 1)\ mod\ 256 = 1$
V čase $t = 10^+$
$s = 1$
$e = 0$
$ta(s) = ta(1) = 3$
...

Z běhu simulace vyplývá zpožděná odezva čítače na signál reset o t_{clk}. Rovněž k první výstupní události dojde až v čase $t = 3$.

2.3.2 Discrete Event Specified Network (DEVN)

V předchozím textu jsme si definovali atomický DEVS, nyní se seznámíme s možností propojování jednotlivých atomických DEVS modelů do hierarchických struktur tak, abychom mohli modelovat komplexní systémy. Propojení DEVS modelu s jiným DEVS modelem provedeme přeposláním požadovaných výstupních událostí prvního modelu na vstupní události druhého modelu. DEVS formalismus nám zaručuje, že složením dvou atomických DEVS modelů vzniká rovněž DEVS model (DEVS je tedy uzavřený na operaci skládání).

Pro spojování jednotlivé DEVS modely vybavíme vstupními a výstupními porty p.

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

kde

$$X = \{(p, v) \mid p \in IPort, v \in X_p\}$$

je množina vstupních portů a hodnot kterých mohou nabývat a

$$Y = \{(p, v) \mid p \in OPort, v \in Y_p\}$$

je množina výstupních portů a hodnot kterých mohou nabývat. Množina X_p (Y_p) obsahuje hodnoty všech možných vstupů (výstupů) pro daný port

Tyto porty pak mezi sebou dle požadavků spojujeme. Propojíme-li vstupní a výstupní porty podsystémů jistého systému, jedná se o *vnitřní propojení*. Naopak pokud existuje spojení z výstupního portu podsystému do výstupního portu systému, nazýváme jej *vnější výstupní propojení*. *Vnější vstupní propojení* je propojení mezi vstupním portem systému a vstupním portem z jeho podsystémů.

Definice

Strukturu N nazýváme DEVN systémem pokud:

$$N = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select)$$

kde

$X = \{(p, v) \mid p \in IPort, v \in X_p\}$ je množina vstupních portů a hodnot kterých mohou nabývat.

$Y = \{(p, v) \mid p \in OPort, v \in Y_p\}$ je množina výstupních portů a hodnot kterých mohou nabývat.

D je množina názvů komponent, které systém obsahuje.

Komponenty jsou DEVS modely, které jsou vybaveny porty, tj. pro každé $d \in D$

$$M_d = (X_d, Y_d, S, \delta_{ext}, \delta int, \lambda, ta)$$

je DEVS s

$$X_d = \{(p, v) \mid p \in IPorts_d, v \in X_p\}$$

$$Y_d = \{(p, v) \mid p \in OPorts_d, v \in Y_p\}$$

Pro umožnění vytváření složených modelů jsou nutné tyto podmínky:

- Vnější spárování slouží k připojení vnějších vstupů ke vstupům komponent:

$$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$$

- Vnější párování slouží k připojení vnějších výstupů k výstupům komponent:

$$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, ip_d \in OPorts_d\}$$

- Vnitřní propojení slouží k připojení vstupů a výstupů jednotlivých komponent:

$$IC \subseteq \{((a, op_a), (b, ip_b)) \mid op_a \in OPorts_a, a, b \in D, ip_b \in IPorts_b\}$$

  Zároveň však nejsou dovoleny žádné přímé zpětné vazby (tzv. rychlé smyčky).

- A existuje funkce *Select*, která slouží pro rozhodování při současném výskytu více událostí:

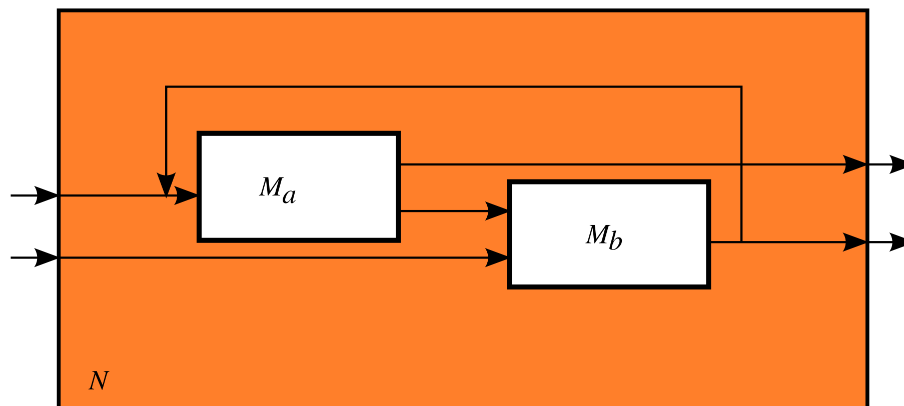
$$Select : 2^D - \{\} \to D$$

Porty složeného DEVS modelu N, jeho podsystémů M_a, M_b a jejich propojení demonstruje obrázek 2.3. Druhý výstupní port modelu M_a je připojen k prvnímu vstupnímu portu M_b, což lze zapsat dvojicí $[(M_a, op_2), (M_b, ip_1)]$. Ostatní propojení pak definujeme jako

$$[(M_a, op_1), (N, op_1)], [(M_b, op_1), (N, op_2)], [(M_b, op_1), (M_a, ip_1)],$$

$$[(N, ip_1), (M_a, ip_1)], [(N, ip_2), (M_b, ip_2)].$$

Vzhledem k vlastnostem DEVS modelů můžeme model N použít stejným způsobem jako jakýkoliv jiný atomický DEVS model a můžeme ho tedy složit s jinými atomickými nebo složenými modely.



Obrázek 2.3: Model složeného DEVSu

Příklad složeného modelu

Uvažujme obvod složený z AND hradel, který z tříbitového vstupu vypočítá jejich logický součin. Například pokud $X = (0, 1, 1)$ pak $Y = 0$. Hradlo se dvěma vstupy a jedním výstupem realizující funkci $AND(x_1, x_2)$ popíšeme následujícím DEVS modelem (uvažujme, že hradlo má nulové zpoždění):

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

kde

$X = (0, 1) \times (0, 1)$

$Y = (0, 1) \times (0)$

$S = (0, 1) \times (0, 1) \times \mathbb{R}^+$

$\delta_{int}(s) = \delta_{int}(b_0, b_1, \sigma) = \delta_{int}(b_0, b_1, \infty)$

$\delta_{ext}(s, e, x) = \delta_{ext}(b_0, b_2, \sigma, e, x_v, p) = \begin{cases} (x_v, b_1, 0) & \text{pro } p = 0 \\ (b_0, x_v, 0) & \text{pro } p \text{ jiné} \end{cases}$

$\lambda(s) = \lambda(b_0, b_1, \sigma) = (b_0 \wedge b_1, 0)$

$ta(s) = ta(b_1, b_2, \sigma) = \sigma$

Vstupní událost X kartézský součin (x, p), kde x je hodnota bitu a p určuje port na který tato hodnota dorazila. Výstupní událost Y je kartézský součin $(y, 0)$, kde y je hodnota výstupu (obvod logického součinu obsahuje jeden výstupní port). Stav systému určuje $S = (x, y, \sigma)$, kde x (y) je hodnota prvního (druhého) bitu na vstupu AND obvodu a σ je čas další naplánované události.

Pokud na vstupní porty dorazí událost ve formě dvojice (x, p), kde x je hodnota bitu a p označuje vstup obvodu AND, pak dojde k externí události, která změní stav systému externí přechodovou funkcí δ_{ext} a pomocí funkce posuvu času (ta) naplánuje výstupní událost o hodnotě λ. K výstupu tedy dojde okamžitě po výskytu externí události.

Složením dvou modelů M_{AND} získáme model, který vypočítá $AND[AND(x_1, x_2), x_3]$, tj. logický součin tří bitů. Označíme-li modely M_{AND} jako A a B, které budou tvořit podsystémy systému N, pak porty spojíme následujícím způsobem: $[(A, 0), (B, 0)]$, $[(B, 0), (N, 0)]$, $[(N, 0), (A, 0)]$, $[(N, 0), (B, 0)]$, $[(N, 1), (A, 1)]$, $[(N, 1), (B, 1)]$.

Skládáním DEVS modelů můžeme vytvořit opět DEVS model. Tento hierarchický přístup umožňuje programátorovi vytvářet modely zdola nahoru. Skládáním jednotlivých DEVS modelů může vytvořit složitý model, ale i současně vytvářet jistou úroveň abstrakce. Tento

kompozitní přístup přináší problémy v případě současného výskytu více událostí, které se řeší pomocí funkce priority nebo paralelním DEVS.

2.3.3 Paralelní DEVS

Paralelní DEVS se liší od původního DEVN tím, že umožňuje všem komponentám současně reagovat na události a posílat je jiným komponentám. O tom jak budou kolizní události zpracovány rozhoduje sama komponenta, které jsou doručeny. Zprávy jsou komponentě předány jako seznam všech současných vstupních dvojic port-hodnota. Základní paralelní DEVS je podobný atomickému DEVS.

Definice

Základní paralelní DEVS je struktura:

$$DEVS = (X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

kde
$X_M = \{(p, v) \mid p \in IPorts, v \in X_p\}$ je množina vstupních portů a hodnot
$Y_M = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ je množina výstupních portů a hodnot
S je množina sekvenčních stavů
$\delta_{ext} : Q \times X_M^b \to S$ je vnější přechodová funkce
$\delta_{int} : S \to S$ je vnitřní přechodová funkce
$\delta_{con} : Q \times X_M^b \to S$ rozlišovací funkce
$\lambda : S \to Y^b$ je výstupní funkce
$ta : S \to \mathbb{R}^+_{0,\infty}$ je funkce posunu času
$Q := \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ je množina totálních stavů

Z definice je zřejmé, že místo jediné vstupní události může mít paralelní DEVS množinu vstupních událostí, přičemž platí, že prvky v této množině se mohou opakovat. Definice je rovněž rozšířena o rozlišovací funkci δ_{con}, která slouží k řešení kolizí při současném výskytu vnější a vnitřní události.

Složený paralelní DEVS

Složený paralelní DEVS se od původního liší absencí funkce *Select*. Význam této změny je ze sémantického hlediska značný a týká se především zpracovávání současného výskytu události v několika komponentách. Jsou generovány všechny současné výstupní události, které jsou dále přenášeny pomocí sítě vytvořené pomocí propojených portů.

2.3.4 Differential Equation System Specification (DESS)

Systém specifikovaný diferenciálními rovnicemi je struktura:

$$DESS = (X, Y, Q, f, \lambda)$$

kde
X je množina vstupů
Y je množina výstupů
Q je množina stavů
$f : Q \times X \to Q$ je funkce udávající rychlost změny stavu

Výstupní funkce λ je
$\lambda : Q \to Y$ (Moorova typu)
nebo
$\lambda : Q + X \to Y$ (Mealyho typu)
Struktura, která s DESS souvisí je

$$S = (T, X, \Omega, Y, Q, \Delta, \Lambda)$$

musí splňovat následující podmínky

- Časová základna T je množina reálných čísel $\mathbb{R}$
- X, Y a Q jsou vektory z prostoru reálných čísel
- Množina Ω všech připustitelných vstupních segmentů je množina ohraničených, po částech spojitých trajektorií.
- Stavové a výstupní trajektorie jsou ohraničené a po částech spojité.
- Pro daný vstupní ohraničený segment $\omega :< t_1, t_2 >\to X$ a počáteční stav q v čase t_1 je požadováno, aby stavová trajektorie $STRAJ_{q,\omega}$ dynamického systému pro všechny body v intervalu $< t_1, t_2 >$ splňovala

$$STRAJ_{q,\omega}(t_1) = q$$

$$d\ STRAJ_{q,\omega}(t)/dt = d(STRAJ_{q,\omega}(t), \omega(t))$$

- Výstupní funkce Λ dynamického systému pro systémy Moorova typu je určena

$$\lambda(q, x) = \lambda(q)$$

Pro systémy Mealyho typu pak

$$\lambda(q, x) = \lambda(q, x)$$

Správně definovaný systém (tj. s jednoznačným řešením) musí splňovat Lipschnitzovu podmínku. Více v [19].

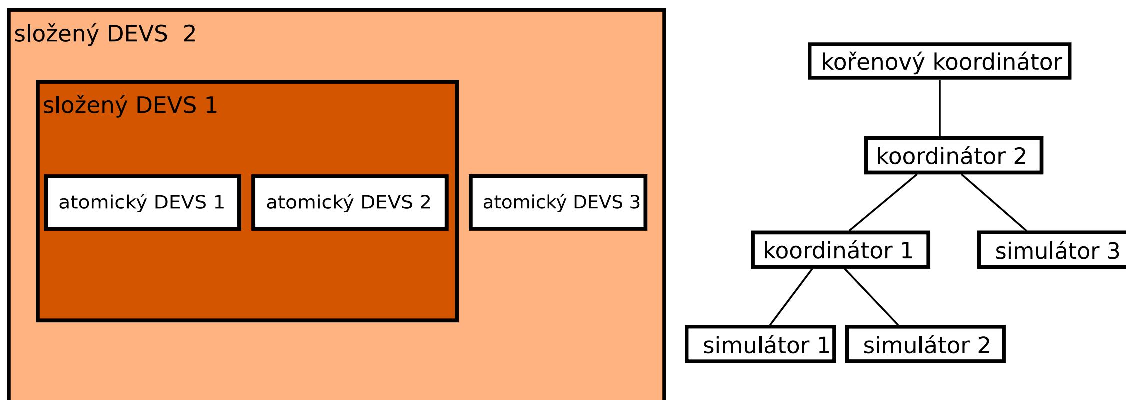
Složený DESS

Pro atomický DEVS existuje DEVN (složený DEVS), který umožňuje propojit libovolný počet atomických či složených DEVS modelů vybavených porty. Velmi podobným způsobem lze definovat systém specifikovaný diferenciálními rovnicemi:

$$N =< X, Y, D, \{M_d\}, \{I_d\}, \{Z_d\} >$$

pro který platí

- X, Y jsou vektory ve vektorovém prostoru nad $\mathbb{R}$
- D je množina komponent, komponenta $d \in D$ musí být DESS nebo jiná síť typu N.
- V systému neexistují žádné algebraické cykly. To znamená, že ve zpětné vazbě musí být alespoň jedna komponenta schopná vypočítat svůj výstup bez toho, aby musela znát vstup.



Obrázek 2.4: Převod složeného DEVS modelu (vlevo) na jeho simulační model (vpravo)

2.4 Simulace DEVS a DESS formalismů

Simulátor má za úkol z dodané specifikace systému, počátečního stavu všech stavových proměnných a časových segmentů pro všechny vstupní porty rozhraní modelu vytvořit odpovídající stav a výstupní segmenty modelu. K tomu je třeba jej navrhnout tak, aby implementoval daný formalismus, ve kterém je model specifikován.

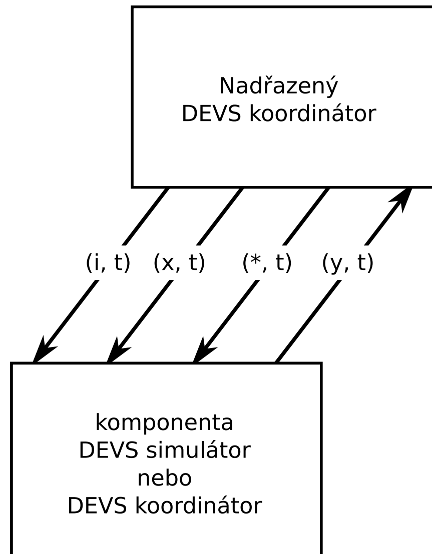
Například simulátor, který řeší diferenciální rovnice, musí být schopen přijmout po částech spojité vstupní trajektorie a vytvořit je pro stavové a výstupní proměnné. Nyní si ukážeme, jak navrhnout a implementovat simulátory, které řeší modely atomického a složeného DEVS a DESS.

K základnímu (atomickému) modelu můžeme vytvořit simulátor. Simulaci složeného modelu, který obsahuje více atomických či jiných složených modelů, provádíme koordinátorem. Simulátor a koordinátor komunikuje s okolím pomocí několika druhů zpráv. Přiřazením koordinátoru složenému modelu a simulátoru atomickému modelu vznikne simulační strom v jehož kořenu je kořenový koordinátor a v listech pak simulátory. Převod DEVS modelu na jeho simulační model demonstruje obrázek 2.4.

2.4.1 Simulátory DEVS/DEVN

Diskrétní simulátory pracují s udržovaným kalendářem událostí seřazených tak, že časově nejbližší událost se nachází na začátku kalendáře a nejvzdálenější na jeho konci. K událostem dojde až tehdy, jsou-li odebrány ze začátku kalendáře. Poté je událost spuštěna. To může vést k přidání nových událostí do kalendáře nebo naopak jejich zrušení tak, že je z kalendáře odebere.

Podobný přístup se budeme snažit aplikovat i na atomické a složené DEVS modely. Definujeme chování simulátoru pro atomické DEVS modely a koordinátoru pro složené DEVS modely. Použijeme modifikovaný kalendář pro efektivní správu času celého modelu. Nakonec zjistíme, jak mohou různé formalismy spolupracovat. Je totiž možné, abychom různé formalismy použili v jednom modelu právě pomocí obecného významu zpráv, které řídí průběh simulace. Tento koncept se nazývá *DEVS-bus*.



Obrázek 2.5: Zprávy používané pro simulaci DEVS modelů a směr jejich zasílání

Zprávy používané pro simulaci

Hierarchický simulátor pro složené DEVS modely, který se skládá z DEVS simulátoru a koordinátoru, používá čtyři druhy zpráv (zdroj a cíl jednotlivých zpráv je na obrázku 2.5):

Inicializační zpráva (i, t) je rozeslána nadřazeným koordinátorem všem komponentám, které obsahuje.

Interní přechodová zpráva (*, t) je odeslána koordinátorem té podřízené komponentě, které je určena.

Výstupní zpráva (y, t) je odeslána komponentou svému nadřízenému koordinátoru a slouží k upozornění že došlo k výstupní zprávě.

Vstupní zpráva (x, t) je odeslána koordinátorem odpovídajícím komponentám pro provedení vnější události.

K tomu, abychom byli schopni simulovat modely složeného DEVSu je nutné specifikovat algoritmus, který řídí chování simulátoru atomického DEVSu, koordinátoru složeného DEVSu a kořenového koordinátoru, který se nachází na vrcholu stromu (viz obrázek 2.4).

Simulátor atomického DEVSu

Simulátor používá dvě proměnné tl a tn. Proměnná tl obsahuje simulační čas kdy došlo k poslední události, tn pak plánovaný čas výskytu nejbližší další události. Z definice funkce posuvu času (ta) v DEVS modelu se

$$tn = tl + ta(s)$$

Pomocí aktuálního globálního simulačního času t je možné vypočítat uplynulý čas od poslední události

$$e = t - tl$$

a čas zbývající do další události

$$\sigma = tn - t = ta(s) - e$$

Čas výskytu další události tn se posílá nadřazenému koordinátoru a slouží ke správné synchronizaci událostí.

```
DEVS simulátor
   proměnné:
     parent // nadřazený koordinátor
     tl     // čas poslední události
     tn     // čas následující události
     DEVS   // DEVS model = (X, Y, S, δ_int, δ_ext, λ, ta )
     y      // aktuální výstup
   když dorazí inicializační zpráva (i, t) v čase t:
     tl = t - e
     tn = tl + ta(s)
   když dorazí interní přechodová zpráva (*, t) v čase t:
     y = λ(s)
     pošli výstupní zprávu (y, t) nadřazenému koordinátorovi
     s = δ_int(s)
     tl = t
     tn = tl + ta(s)
   když dorazí vstupní zpráva (x, t) v čase t se vstupem x
     e = t - tl
     s = δ_ext(s, e, x)
     t = tl + ta(s)
```

Z algoritmu vyplývá, že je nutné na začátku simulace přijmout inicializační zprávu (i, t). Jakmile DEVS simulátor zprávu obdrží, nastaví čas výskytu poslední události tn tak, že odečte čas e od času t, ve kterém byla zpráva doručena. Čas výskytu další události tn se vypočítá přičtením výsledku funkce posuvu času $ta(s)$ k času výskytu poslední události tl. Tento čas tn je odeslán nadřazenému koordinátorovi, aby znal čas výskytu nejbližší vnitřní události.

Interní přechodová zpráva $(*, t)$ zaručuje provedení interní přechodové funkce $s = \delta_{int}(s)$. Zároveň je vypočítána výstupní funkce $y = \lambda(s)$ a vytvořena výstupní zpráva (y, t), která je odeslána nadřízenému koordinátorovi. Nakonec je přenastaven čas poslední události tl a čas výskytu následující události tn je opět aktualizován přičtením výsledku funkce posuvu času $ta(s)$ k aktuálnímu času.

Vstupní zpráva (x, t) informuje simulátor o výskytu vnější události x v čase t. Simulátor na základě současného stavu s, doby uběhnuté od poslední události e a události x vypočítá nový stav $s = \delta_{ext}(s, e, x)$. Stejně jako u zpracování interní přechodové zprávy se čas uplynulý od poslední události tl nastaví na aktuální čas t a čas výskytu další události tn vypočte jako součet aktuálního času t a výsledku funkce posuvu času $ta(s)$.

Simulátor složeného DEVSu

Komponenty obsažené v koordinátoru, který simuluje složený DEVS, jsou všechny obsluhovány vlastním simulátorem. Úlohou koordinátoru je provádět správnou synchronizaci

komponent a zpracovávat výstupní zprávy pocházející od jeho vlastních komponent nebo jeho nadřazeného koordinátora. K tomu využívá výše uvedených společných zpráv.

Koordinátor obsahuje seznam událostí, který obsahuje události všech svých komponent seřazené od nejbližší po nejvzdálenější události. K určení výskytu události konkrétní komponenty využívá její proměnné tn. Pro případ současného výskytu více komponent je koordinátor vybaven funkcí $Select$, která je definována odpovídajícím složeným DEVS modelem. První položka seznamu událostí označuje výskyt nejbližšího interního přechodu určité komponenty.

Koordinátor obsahuje stejně jako simulátor atomického DEVSu dva časové údaje. Čas výskytu nejbližší události

$$tn = min\{tn_d \mid d \in D\}$$

a čas výskytu poslední události

$$tl = max\{tl_d \mid d \in D\}.$$

Proměnná tn slouží nadřazenému koordinátorovi k určení času nejbližšího výskytu události, tl pak k určení času posledního výskytu události.

```
DEVS koordinátor
   proměnné:
     parent     // nadřazený koordinátor
     tl         // čas poslední události
     tn         // čas následující události
     N          // DEVN = (X, Y, D,{ M_d | d ∈ D }, EIC, EOC, IC, Select)
     event-list // seznam seřazených dvojic (d, tn_d) podle tn_d,
                // kde d ∈ D je komponenta a tn_d čas výskytu
                // její interní události
   když dorazí inicializační zpráva (i, t) v čase t:
     pro všechny komponenty d ∈ D:
        pošli komponentě d inicializační zprávu (i, t)
     seřad’ event − list podle tn_d
     tl = max{ tl_d | d ∈ D }
     tn = min{ tn_d | d ∈ D }
   když dorazí interní přechodová zpráva (∗, t) v čase t:
     pošli interní přechodovou zprávu (∗, t) první komponentě v event − list
     seřad’ event − list podle tn_d
     tl = t
     tn = min{ tn_d | d ∈ D}
   když dorazí vstupní zpráva (x, t) v čase t se vstupem x = (value, port):
     // zjisti, se kterou komponentou je koordinátor propojen
     connected = {(c, p) | c ∈ D, ((N, x.port), (c, p)) ∈ EIC}
     pro všechny (c, p) z connected:
       pošli vstupní zprávu {(x_d, t) | x_d = (x.value, p)} komponentě c
     seřad’ event − list podle tn_d
     tl = t
     tn = min{ tn_d | d ∈ D}
```

```
když dorazí výstupní zpráva (y_d, t) v čase t s výstupem y_d od komponenty d:
  // zjisti, zda existuje propojení d s výstupními porty koordinátoru
  když {p | ((d, y_d.port), (N, p)) ∈ EOC}
    pošli výstupní zprávu {(y_N, t) | y_N = (y.value, p)} pro parent
  // zjisti, zda existuje propojení uvnitř koordinátoru
  connected = {(c, p) | c ∈ D, ((d, x.port), (c, p)) ∈ IC}
  pro všechny (c, p) z connected:
    pošli vstupní zprávu {(x_c, t) | x_c = (y_d.value, p)} komponentě c
```

Když koordinátor obdrží inicializační zprávu, přepošle ji všem svým komponentám. Po jejich inicializaci nastaví koordinátor svůj čas výskytu nejbližší události tn na čas nejbližšího výskytu události všech komponent. Svůj čas výskytu nejvzdálenější události tl pak určí podle času nejvzdálenější události všech komponent.

Jakmile koordinátor příjme interní přechodová zpráva $(*, t)$ přepošle ji té komponentě, která je v seznamu událostí seřazeném podle času výskytu jednotlivých událostí na prvním místě. Komponenta provede změnu stavu (pokud jde o simulátor) a eventuálně odešle výstupní zprávu (y, t) zpět koordinátoru. Po provedení všech interních přechodů a externích událostí způsobených výstupními zprávami těchto komponent se nastaví čas výskytu další události na čas nejbližšího výskytu události všech komponent, které koordinátor obsahuje.

Po obdržení výstupní zprávy (y, t), která obsahuje údaje o zdrojovém portu a výstupní hodnotě komponenty, je zjištěno, zda port není připojen na výstupní port koordinátoru. V takovém případě je výstupní zpráva přeposlána nadřazenému koordinátoru. Pokud je port připojen na vstupní port jiné komponenty, je zpráva převedena na vstupní zprávu

$$(x_r, t) = ((y.value, p), t) \mid ((d_s, y.port), (d_d, p)) \in IC$$

kde

d_s je komponenta, která výstupní zprávu odeslala

d_t je komponenta, která je připojena k portu $y.port$ komponenty d_s.

Po odeslání případných výstupních a vstupních zpráv se podle obsahu seznamu událostí aktualizuje čas tn a tl.

Při výskytu vstupní zprávy (x, t) obdržené od nadřazeného koordinátoru se zjistí které komponenty jsou ke vstupnímu portu připojeny a je jim zaslána vstupní zpráva

$$(x_r, t) = ((x.value, p), t) \mid ((N, x.port), (d_r, p)) \in EIC$$

kde

d_r je komponenta připojená ke vstupnímu portu koordinátoru

Po odeslání případných výstupních a vstupních zpráv se podle obsahu seznamu událostí aktualizuje čas tn a tl.

Kořenový koordinátor

Kořenový koordinátor řídí hlavní simulační smyčku tak, že posílá zprávy svému jedinému přímému potomkovi (komponentě). Nejdříve rozešle inicializační zprávu díky které se inicializuje celý model (inicializační zpráva je propagována všemi koordinátory). Poté zjistí čas výskytu nejbližší události tn a vyšle interní přechodovou zprávu (*, t), kde $t = tn$. To opakuje dokud není splněna podmínka ukončení simulace.

```
DEVS kořenový koordinátor
  proměnné:
    t             // aktuální simulační čas
    potomek       // DEVS simulátor nebo koordinátor
  t = t0
  pošli inicializační zprávu (i, t) potomkovi
  t = potomek.tn  // nastaví aktuální čas na čas výskytu události potomka
  opakuj:
    pošli interní přechodovou zprávu (*, t) potomkovi
    t = potomek.tn
  dokud není splněna podmínka ukončení simulace
```

2.4.2 Výhody a nevýhody použití DEVS formalismu

Hlavní výhodou DEVS formalismu je možnost vytvářet hierarchické modely a vytvořené odladěné modely znovu využívat. Rovněž pevně daná struktura modelu usnadňuje orientaci a čitelnost modelu pro ty, kteří model nevytvářeli. Tvorba nového modelu může být využitím existujících podmodelů rychlejší a bezpečnější. Modely se rovněž snadněji udržují a existují postupy pro validaci a verifikaci.

Na druhou stranu není jednoduché určit, jakým způsobem komplexní model rozdělit na komponenty. Může být zároveň více časově náročné vytvořit komplexní model dekompozicí na více podmodelů. Otázkou je i znovupoužitelnost vytvořeným modelů. Proto je vždy na tvůrci modelu, zda DEVS formalismus využije a měl by přitom přihlédnout ke všem hlediskům, která s vytvářením DEVS modelů souvisí.

2.5 Grafické editory pro zadávání modelů

Uživatelské rozhraní je pojem popisující interakci člověka se strojem. Dělí se na vstupní a výstupní část. Vstupní část umožňuje kontrolu systému uživatelem, výstupní mu o systému zpřístupňuje informace.

Pojem uživatelské rozhraní nalezneme ve všech vztazích člověk-stroj, v počítačové terminologii vstup chápeme jako používání klávesnice nebo polohovacího zařízení. Výstup v textové a/nebo grafické podobě bývá zobrazen na monitoru, tiskárně nebo jiném zobrazovacím zařízení. V současnosti rozeznáváme dva nejrozšířenější druhy uživatelského rozhraní:

Příkazový řádek (CLI[1]) na který uživatel píše pomocí klávesnice příkazy o jejichž provádění je informován pomocí textu.

Grafické uživatelské rozhraní (GUI[2]) umožňuje řízení aplikace pomocí klávesnice a polohovacího zařízení. O nabízených možnostech je informován textovou a grafickou formou.

Využívání grafické reprezentace dat se již v několika letech dostává do popředí zájmu uživatelů. Dodávaná informace se totiž rozšiřuje oproti textu o jeden rozměr a je možné na ni pohlížet z více perspektiv.

Propojení simulačních nástrojů a grafických editorů se přímo nabízí. Reprezentace modelu, jakou nám nabízí orientovaný graf – funkcionální model – blokové schéma, je do dvourozměrného prostoru přenositelná velmi snadno. V poslední době je navíc „programování“ pomocí pouhého propojování bloků stále populárnější.

[1] Command-line Interface
[2] Graphical User Interface

2.5.1 Historie vývoje grafického uživatelského rozhraní

Koncepce grafického uživatelského rozhraní jak ho známe dnes byla představena již v prosinci roku 1968 panem Engelbartem [16] a jeho týmem. Ve své devadesátiminutové prezentaci předvedl hypertextový systém *NLS/Augment* s několika okny, ovládaný třítlačítkovou myší. Ve své době šlo o převratný objev s důsledky až do současnosti.

Na uvedení do praxe však tento koncept musel čekat pět let. Tehdy v dílnách XEROX PARC vzniklo WIMP[3] paradigma uvedené do komerční praxe až v roce 1981 systémem XEROX 8010.

Výzkum týmů Lisa a Macintosh firmy Apple Computer v roce 1984 vedl k rozšíření prvního komerčně úspěšného produktu založeného na grafickém uživatelském rozhraní rozšířeného o rolovací menu, překrývatelnými okny a manipulací s ikonami.

Situace na dnešní nejrozšířenější platformě *x86* se začala vyvíjet příchodem produktu GEM Desktop od Digital Research, který běžel jako nadstavba MS-DOSu, zaznamenal však minimální úspěch. Společnost Microsoft uvedla v roce 1985 jako grafickou nadstavbu nad MS-DOS systém Windows, který vycházel z Mac OS. Do verze 3.0 však nezaznamenal větší úspěch. Později si získal oblibu integrací systému správy souborů, konfigurace pomocí průvodců (anglicky *Wizard*) a v neposlední řadě diskutovanou integrací prohlížeče internetových stránek.

Potřeba grafického uživatelského prostředí v Unixových systémech vedla na začátku 80. let na MIT[4] k vývoji X Window System (zkráceně X11 nebo X). Původním cílem projektu bylo umožnit vzdálený přístup ke grafickému rozhraní bez ohledu na operační systém nebo hardware. Vzhledem k velkému množství dostupných zdrojových kódů projektu se ale X Window System stal standardní vrstvou pro správu grafických, vstupních a výstupních zařízení pro vytvoření vzdáleného i lokálního uživatelského rozhraní na všech unixových systémech.

Během vývoje grafických uživatelských rozhraní byla snaha upozadit příkazový řádek (CLI), což se projevilo jako nerozumné. Dnes se situace ve prospěch příkazové řádky mění v řadě operačních systémů a aplikacích.

2.5.2 Uživatelské rozhraní aplikací

K vytváření uživatelského rozhraní se nejčastěji používají vývojové knihovny (anglicky *toolkit*). Jsou to prostředí, která zjednodušují vytváření a sjednocují vzhled prvků uživatelského rozhraní jako jsou tlačítka, menu, posuvníky, dialogy, atp.

Programování pro samotné jádro grafického uživatelského rozhraní by bylo komplikované a zdlouhavé. Navíc bychom se tím ochuzovali o možnost běhu naší aplikace na jádře jiném. Z tohoto důvodu vznikly vývojové knihovny, z nichž rozeznáváme uniplatformní (jako je například WINAPI) a multiplatformní (GTK, QT, FLTK, Swing, wxWidgets).

Aplikace používající WINAPI nepřeložíme a nespustíme na žádném jiném operačním systému než Windows a platformě x86, zatímco aplikaci napsanou například pomocí QT přeložíme pro Windows, X11, Mac OS X a dokonce pro mobilní zařízení využívající OS Linux, přičemž jediná věc, která se na různých platformách liší, není zdrojový kód naší aplikace, ale implementace knihovny QT (kterou my řešit nemusíme). Výběr vývojové knihovny by se proto neměl nikdy podceňovat a vývojář by měl dbát na možnosti multiplatformního programování.

[3] Windows, Icons, Menus, Pointers – okna, ikony, menu, kurzor myši
[4] Massachusetts Institute of Technology

2.5.3 Pozice a způsob ovládání prvků uživatelského rozhraní

Umisťování ovládacích prvků se postupem času dostalo do nepsané standardní formy, kdy ovládání stejně zaměřených aplikací se liší jen velmi málo. Rovněž pořadí a názvy položek menu a klávesové zkratky se ustálily. Při návrhu aplikace je nutné tyto zvyklosti respektovat, protože jejich ignorací zhoršujeme uživatelův komfort při užívání naší aplikace. Dialogová okna, pořadí jejich tlačítek, klávesové zkratky, výchozí volba a zpráva by se, pokud možno, měla co nejvíce blížit nejrozšířenějšímu rozložení dialogových oken.

2.5.4 Požadavky

Pro zefektivnění tvorby DEVS modelů lze použít grafický editor. Ten se většinou skládá z plátna, knihovny modelů a stromu obsahující model, který je aktuálně otevřen.

Vytváření modelu a jeho modifikace by měla být jednoduchá a intuitivní. Model zobrazený na plátně by měl být dostatečně popisný tak, aby bylo na první pohled zřejmé, k čemu je model určen, z čeho se skládá a jak je propojen. Jednotlivé bloky (DEVS modely) a jejich porty by měly být pojmenované textem s přiměřenou délkou a případným rychle dostupným detailním popisem (například pomocí kontextové nápovědy). Navigaci by mělo zajišťovat stromové zobrazení modelu s jasným rozlišením atomických a složených modelů a se zobrazením aktuální pozice modelu korespondující situaci na plátně. Použití knihovny modelů by mělo být přímočaré například použitím principu *Drag&drop*.

2.5.5 Přehled nástrojů

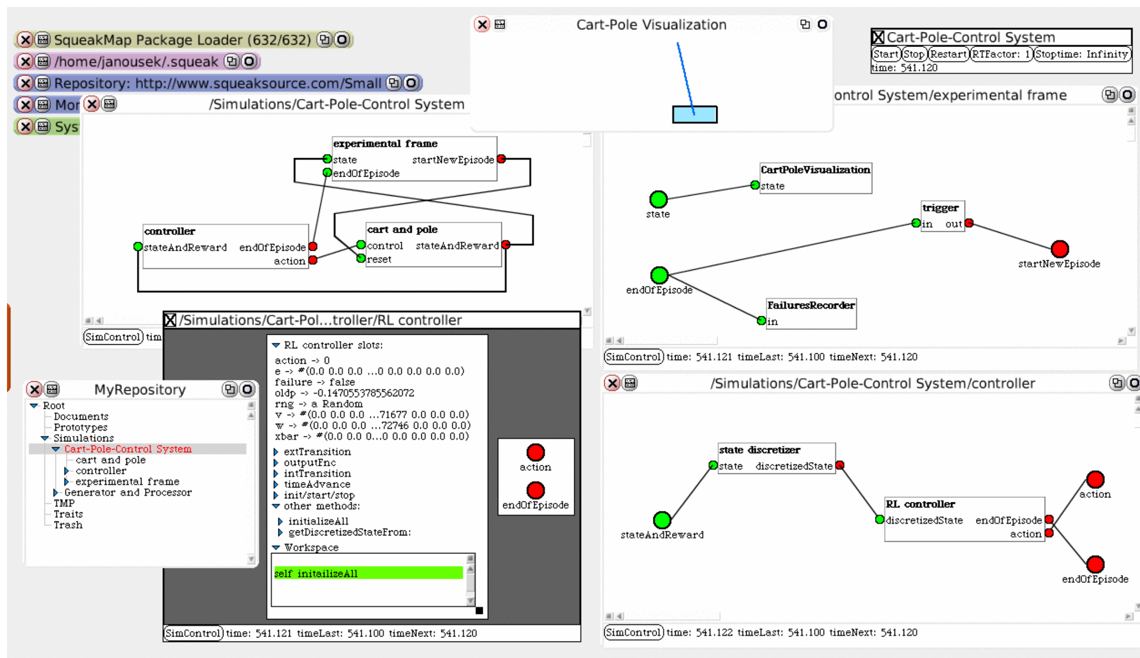
Nástrojů pro tvorbu DEVS modelů pomocí grafického editoru příliš velké množství neexistuje. Většina je zatížena restriktivní licencí a přístup k některým nástrojům je komplikován vyplňováním formulářů, které jsou schvalovány a až poté je přístup k získání aplikace dovolen.

SmallDEVS

SmallDEVS je komplexní systém pro vytváření, modifikaci a simulaci DEVS formalismu napsaný v jazyce *Smalltalk/Squeak*, díky tomu je multiplatformní. Je určen pro výzkumné a výukové účely. Umožňuje dynamickou modifikaci modelu za běhu (Dynamic DEVS). Obsahuje prostředky pro modifikaci atomických a složených DEVS modelů a knihovnu modelů, která obsahuje všechny vytvořené modely. Jakékoliv jejich podmodely lze libovolně přidávat do vlastních modelů. Rozhraní je specifické pro aplikace napsané v prostředí *Squeak*, což může začátečníkům v jazyce Smalltalk činit problémy. Prostředí může být značně produktivní, ale není snadné se v něm zorientovat tak rychle, jako například v aplikacích s běžným rozhraním. SmallDEVS je na jednu stranu tím pádem silným nástrojem a na druhou komplikovanějším pro uživatele prostředí *Squeak* neznalé.

PowerDEVS

PowerDEVS je prostředí pro tvorbu a simulaci DEVS modelů. Skládá se z editoru atomického a složeného modelu a preprocesoru pro generování zdrojového kódu výsledného modelu v jazyce C++. Uživatelské rozhraní je napsáno v jazyce Visual Basic a program je možné provozovat v operačních systémech Microsoft Windows. Vygenerované zdrojové kódy lze přeložit v libovolném překladači jazyka C++ (tj. i na jiných platformách).



Obrázek 2.6: Grafické uživatelské rozhraní prostředí SmallDEVS

Plátno modelu a knihovna modelů se nachází v oddělených oknech, přidávání modelu z knihovny na plátno je řešeno pomocí *Drag&drop*. Knihovna modelů je rozdělena do několika kategorií:

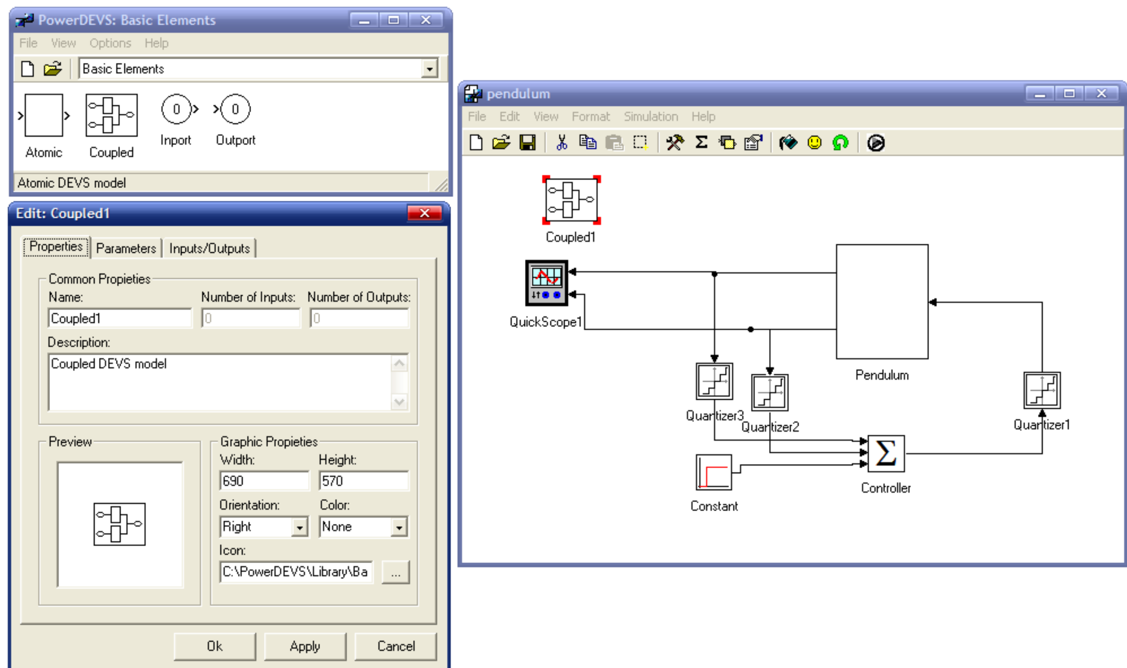
- Základní prvky - nový atomický, složený DEVS a vstupní, výstupní porty
- Spojité bloky - integrátor, nelineární funkce, aritmetické a goniometrické bloky, atp.
- Hybridní bloky - kvantizer, přepínač, komparátor a další
- Generátory signálů - pulzní, konstantní, trojúhelníkový, sinusový, krokový, atp.
- Výstupní prvky - uložení do souboru, zobrazení v grafu, ...

Definice nových bloků je poněkud méně přehledná, nachází se ve formuláři s několika záložkami (viz 2.7) a ke specifikaci chování bloku je určen další formulář. Při editaci je nutné rovněž často používat kontextové menu bloků. Vytvořený model je možné exportovat do knihovny modelů. Během editace není možné vracet provedené změny pomocí funkce *Zpět*. I přesto lze v programu navrhnout poměrně rychle libovolný DEVS model.

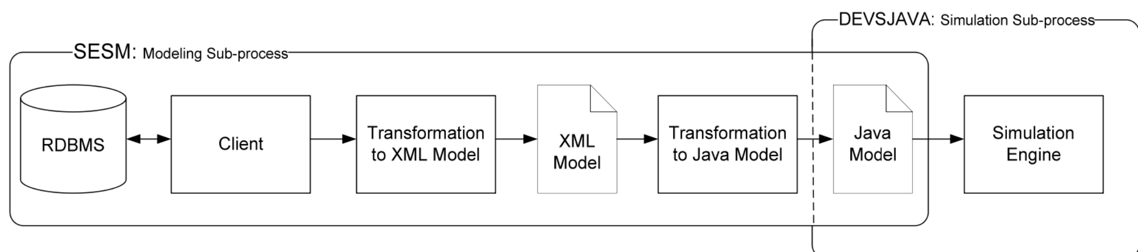
Scaleable Entity Structure Modeler with Complexity Measures (SESM/CM)

SESM/CM je grafický editor schémat vhodných pro hierarchické modely. Umožňuje definovat dva druhy objektů: atomický a složený. Nové objekty lze rovněž vytvářet ze šablon třech druhů:

- Šablona modelu - může obsahovat pouze jiné šablony modelu
- Šablona instance modelu - může obsahovat pouze jiné instance šablony modelu, vzniká vytvořením instance šablony modelu



Obrázek 2.7: Grafické uživatelské rozhraní programu PowerDEVS



Obrázek 2.8: Převod modelu zadaného v SESM/CM do simulačního kódu v DEVSJAVA

- Instance modelu - vzniká vytvořením instance šablony instance modelu

Uživatel nejdříve vytvoří šablonu modelu, poté její instanci a nakonec instanci modelu. To podle autorů umožní vytvářet alternativní modely v závislosti na požadavcích.

Program využívá k uložení modelu databázi JDBC-ODBC, což umožňuje sdílení modelů prostřednictvím řízeného přístupu k databázi modelů. Převod modelu zadaného pomocí SESM/CM demonstruje obrázek 2.8. Model je nejdříve uložen v jazyce XML, poté převeden do zdrojového kódu v programovacím jazyce Java, který je simulován pomocí knihovny DEVSJAVA.

2.5.6 Možná vylepšení

Cílem této práce je vytvořit obecný grafický editor simulačních schémat zaměřený na problematiku vytváření DEVS modelů. Není zaměřen na žádnou konkrétní simulační knihovnu, která DEVS formalismus implementuje, ale definuje rozhraní pro vytváření exportních filtrů

sloužících k převodu modelu uloženém v XML do libovolné simulační knihovny. Zároveň je kladen důraz na intuitivní a rychlé ovládání samotné aplikace a na dostatečnou seberepre-zentaci zobrazených bloků pomocí pojmenovaných bloků a portů s možností dodatečného zobrazení detailního popisu požadované komponenty. Důležitá je rovněž snadná a přehledná navigace ve stromu, který zobrazuje strukturu modelu. Rovněž práce s knihovnou modelů musí být jasná a přehledná.

2.5.7 Výhody a nevýhody použití

Použití grafických editorů je výhodné především pro usnadnění navigace ve složitějších modelech a manipulací s nimi. Například vytváření komplexních propojení není ve zdrojovém kódu dostatečně přehledné, nehledě na komplikované hledání špatně spojených bloků.

Naopak za zvážení stojí komfort psaní zdrojového kódu oproti grafickému rozhraní založeném na vyplňování formulářů. Zdrojový kód oplývá větší svobodou datových typů, vlastních struktur a zdrojový kód může být čitelnější, než vygenerovaný pomocí filtru. Použití simulačních knihoven rovněž usnadňuje a urychluje implementaci DEVS simulátorů.

2.6 XML a jeho použití v grafických editorech modelů

XML (eXtensible Markup Language) je konzorciem W3C [2] doporučený, všeobecně použitelný otevřený značkovací jazyk [17]. Jedná se zjednodušenou podmnožinu SGML (Standard Generalized Markup Language) určenou k popisu dat různého druhu. Jeho hlavním účelem je umožnění sdílení strukturovaného textu a informací v síti Internet a standardizovaný způsob ukládání datových struktur. K hlavním výhodám dokumentů uložených v XML patří:

- Obsah souboru je strojově i lidmi (relativně) čitelný. Podporuje více kódování znaků různých abeced, mimo jiné *Unicode*, což umožňuje vyjadřovat se téměř v jakémkoliv jazyce.
- Je vhodný k ukládání frekventovaných datových struktur (záznamy, seznamy, stromy, aj.). Jeho hierarchická struktura je vhodná pro většinu dokumentů. Používá se (spolu se SGML) už více než deset let, vývojáři s ním mají řadu zkušeností a využívá se v mnoha programech
- Dokument dokáže popsat svou strukturu a názvy polí (data spolu s metadaty). Přísná pravidla syntaxe umožňují vytvářet jednoduché algoritmy, které dokumenty vytvářejí nebo zpracovávají.
- Formát je v prostém textu, platformně nezávislý, je nezatížený licenčními omezeními a mezinárodně standardizován
- Pomocí známých postupů je možné XML soubor transformovat do zcela jiného tvaru.

V určitých aplikacích však XML vykazuje některé nevýhody [5]:

- Syntaxe je z datového hlediska poměrně obsáhlá, což znesnadňuje efektivitu čtení a zabírá větší prostor. To je především nevhodné pro multimediální aplikace běžící na zařízeních s omezenými zdroji (mobilní telefony, PDA, atp.) nebo při transportech dat prostřednictvím služeb, u kterých se platí za počet přenesených dat (například GPRS[5]).

[5]General Packet Radio Service

Znak	Kódová reference
<	<
>	>
”	"
’	'
&	&

Obrázek 2.9: Předdefinované reference znaků použitých v syntaxi jazyka XML

- Syntaxe obsahuje některé nepotřebné vlastnosti dané dědictvím SGML (mnoho dat popisuje poměrně malé množství dat, která jsou pro nás podstatná).
- Základní zpracování přímo nepodporuje spektrum standardních datových typů.
- Překrývání – nehierarchické datové struktury, vyžaduje další práci.
- Pro zpracování dat existuje řada prostředků, jejichž zvládnutí není jednoduché.
- Není zcela jasné, kdy použít atribut elementu nebo element.
- Syntaktický analyzátor (parser), který dovede zpracovat XML dokument, není snadné vytvořit. Existuje pouze málo parserů, které dovedou zpracovat dokument správně včetně validace a přitom efektivně.

2.6.1 Stavební bloky XML dokumentu a jejich uspořádání

XML dokument se skládá z elementů, které mohou obsahovat atributy a vlastní text. Každý element musí být ukončen a elementy mohou být pouze vnořené se nesmí křížit:

```
<bold><underlined>toto je nedovolené</bold>křížení elementů</underlined>
<bold>vnořování <underlined>elementy</underlined> je v pořádku</bold>
```

Názvy elementů jsou citlivé na velikost písmen. K tomu, aby dokument byl korektní (*well-formed*), je nutné, aby obsahoval pouze jeden kořenový element (XML dokument má stejnou strukturu jako strom, kde v uzlech jsou elementy). Text se nemůže nacházet mimo elementy a hodnoty atributů musí být v uvozovkách nebo apostrofech:

```
<element atribut="hodnota">vlastní text</element>
```

Kořenový element je zpravidla předcházen nepovinnou XML deklarací. To je speciální element, jehož atributy určují použitou verzi XML a kódování, ve kterém je dokument uložen, dokument nesmí obsahovat žádné znaky, které kódování neobsahuje:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Do dokumentu je možné psát komentáře pomocí sekvence znaků:

```
<!-- tento text se nebude interpretovat,
     protože  se jedná o komentář -->
```

Všechny elementy v dokumentu však musí být uzavřené. K reprezentaci elementů, které neobsahují žádný text lze použít místo:

```
<element atribut="hodnota"></element>
```

zkrácenou ekvivalentní verzi:

```
<element atribut="hodnota" />
```

Znaky, obsažené v syntaxi elementů a atributů (<, >, ", ') nebo ty, které nelze napsat pomocí klávesnice, se v textu nahrazují speciální sekvencí `&označení_znaku;`, pro znaky použité pro ohraničení elementů a obsahu atributů se používají předdefinované reference znaků uvedené v tabulce 2.9.

2.6.2 Možnosti definice struktury XML dokumentu

Uživatelem definovaná struktura dokumentu se uvádí ve formátu *Document Type Definition* (DTD) nebo *XML Schema Language* (XSL). Oba se používají při validaci dokumentu. Validní XML dokument je dokument, který splňuje následující podmínky:

- strom reprezentovaný XML dokumentem odpovídá popisu DTD či XSL
- pořadí elementů je v souladu s definicí DTD či XSL
- zadané atributy a jejich hodnoty jsou daný dokument dovolené

Document Type Definition (DTD)

Nejstarší schéma pro definici struktury dokumentu je *Document Type Definition* (DTD) a je původně určeno pro SGML. Z toho důvodu má několik nedostatků. Nemá například podporu pro jmenné prostory, nemá schopnosti zcela popsat strukturu XML dokumentu a nepoužívá syntaxi XML. Dodnes se používá především z důvodu čitelnosti a proto, že se poměrně snadno píše. Strukturu DTD dokumentu přibližuje následující příklad:

```
<!ELEMENT model (block*)>
<!ELEMENT block (name, id, type?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT type (#PCDATA)>
```

Element `model` může obsahovat libovolný počet elementů `block`. K určení četnosti $< 0, n)$ se používá znak `*`. Element `block` obsahuje povinné elementy `name`, `id` a nepovinný element `type`. K určení volitelných atributů slouží znak `?`. Obsah zbylých elementů, které mohou obsahovat pouze text, je dán označením `#PCDATA`.

Pojmenujeme-li výše uvedený DTD dokument jako `model.dtd`, pak XML dokument s obsahem:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model SYSTEM "model.dtd">
<model>
  <block>
    <name>Integrator</name>
    <id>11</id>
    <type>atomic</type>
  </block>
</model>
```

je validním.

XML Schema Language (XSL)

Jedním z novějších jazyků určených pro specifikaci struktury dokumentu v jazyce XML je *XML Schema Language* (XSL). Byl publikován v roce 2001 organizací W3C a je doporučený pro popis struktury XML dokumentů.

Jazyk XSL popisuje strukturu dokumentu, obsahuje názvy elementů, jejich atributů a pro lepší možnosti validace dokumentu datové typy, které určují jejich obsah. Datové typy jsou vestavěné (celé a reálné číslo, řetězec, datum, atp.) a uživatelské. Dokument, který obsahuje popis struktury v jazyce XSL, se nazývá *XML Schema Definition* (XSD). Konkrétní XSD dokument demonstruje následující příklad:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="blockType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="atomic" />
      <xs:enumeration value="coupled" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="model" type="Model"/>
    <xs:complexType name="Model">
        <xs:sequence>
            <xs:element name="name" type="xs:string"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="id" type="xs:decimal"
                        minOccurs="1" maxOccurs="1"/>
            <xs:element name="type" type="xs:blockType"
                        minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Na začátku dokumentu je po deklaraci jmenného prostoru definice uživatelského typu. Hodnoty, které může typ nabývat, jsou určeny výčtem řetězců.

Poté následuje definice komplexního typu, který se skládá z několika elementů. Množství elementů, které XML dokument podle tohoto schématu může obsahovat, je určeno atributy `minOccurs` (minimální počet výskytů) a `maxOccurs` (maximální počet výskytů). V příkladu je nepovinný pouze element `<type>`.

Pojmenujeme-li výše uvedený XSD dokument jako `mode.xsd`, pak XML dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="model.xsd">
  <block>
    <name>Integrator</name>
    <id>11</id>
    <type>atomic</type>
  </block>
```

```
</model>
```

je validní.

Samotný jazyk je poměrně obsáhlý, podrobnou specifikaci lze získat například v [3].

2.6.3 Zpracování XML dokumentu

XML dokument lze zpracovat třemi způsoby:

- programovacím jazykem pomocí SAX
- programovacím jazykem pomocí DOM
- transformací pomocí filtru – XSLT

Simple API for XML (SAX)

SAX pracuje tak, že čte vstupní dokument a jakmile narazí na začátek či konec elementu zavolá uživatelsky definovanou funkci. Ta element podle jeho názvu, případně podle atributů zpracuje. Tento přístup je výhodný především pro svou rychlost a paměťovou nenáročnost. Je nevhodný při potřebě přístupu k náhodně umístěným elementům. Pro programátora je problematické rozeznat v jakém kontextu se nalezený element nachází. Proto je vhodné použít SAX tam, kde se element zpracovává stejným způsobem, ať je umístěn kdekoliv.

Document Object Model (DOM)

Z XML dokumentu se vytvoří stromová struktura, která jej reprezentuje. Poté lze k jednotlivým elementům, jejich atributům a vlastnímu obsahu libovolně přistupovat. Rovněž je možné přidávat a modifikovat elementy, atributy a jejich obsah. Modifikovaný XML dokument lze poté opět uložit. Nevýhodou je především paměťová náročnost, protože před vlastní prací je nutné celý dokument načíst do paměti a vytvořit stromovou strukturu, která mu odpovídá.

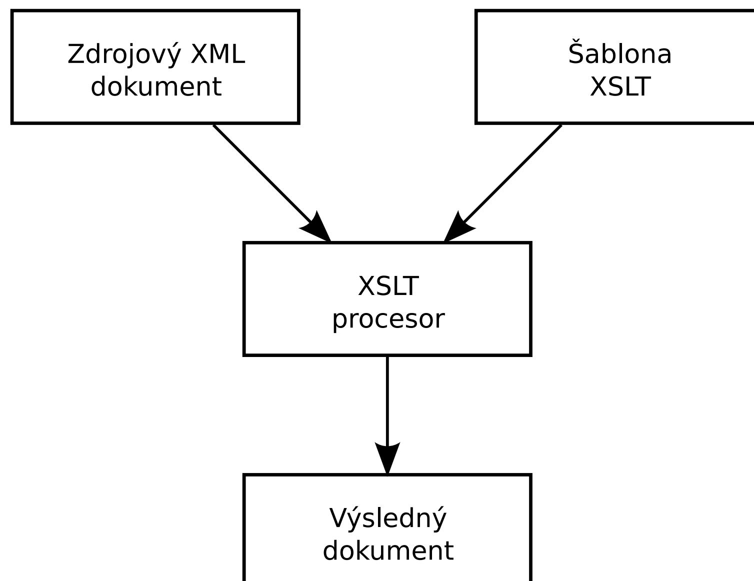
Transformace XML dokumentu pomocí XSLT

Pro transformaci dokumentů XML slouží XSLT (*Extensible Stylesheet Language Transformations*), což je jazyk založený na syntaxi XML. Jeho nejbližším předchůdcem byl *Document Style Semantics and Specification Language* (DSSSL), který sloužil k transformaci SGML dokumentů.

Do procesu XSLT se zapojuje (viz obrázek 2.10:

- jeden či více zdrojových XML dokumentů
- jedna či více XSLT šablon
- XSLT procesor
- výsledný dokument

XSLT procesor použije šablonu a zdrojový XML dokument k vytvoření výsledného dokumentu. Šablona určuje podobu výsledného dokumentu, který může být v jazyce XML nebo i v jiné formě (například zdrojový kód), pomocí pravidel, která obsahuje.



Obrázek 2.10: Transformace dokumentu XML pomocí XSLT

Jazyk XSLT je deklarativní, specifikuje, co se má udělat s jednotlivými uzly. K tomu se využívá jazyka *XPath*, který slouží k výběru elementů či atributů v XML dokumentech [4]. Pokud XSLT uzel nalezne, aplikuje na něj odpovídající šablonu. To spočívá v nahrazení specifikovaných částí údaji ze zdrojového XML dokumentu.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/model">
   // This file is generated
   #include "simulator.h"
  <xsl::apply=templates select=block[*] />
</xsl:template>

<xsl:template match="block">
  <xsl:if test="type = 'coupled'">
class Coupled_<xsl:value-of select="generate-id(.)"/>: Coupled {
  </xsl::if>
  <xsl:if test="type = 'atomic'">
class Atomic_<xsl:value-of select="generate-id(.)"/>: Atomic {
  </xsl::if>
  char *name = "<xsl:value-of select="@name" />";
}
</xsl:stylesheet>
```

První šablona (`<xsl:template>`) vygeneruje úvodní komentář a deklaraci hlavičkového souboru *simulator.h* místo kořenového elementu `model`. Poté na všechny elementy s názvem

`block` použije ostatní šablony. Následuje šablona pro element `block`. Ta vygeneruje v závislosti na obsahu elementu `type` odpovídající kód pro atomický, resp. složený blok.

Použijeme-li tuto šablonu na XML dokument uvedený jako příklad na straně 27, tak nám XSLT procesor vytvoří dokument:

```
<?xml version="1.0" encoding="utf-8"?>
   // This file is generated
   #include "simulator.h"
class Atomic_id235d3: Atomic {
  char *name = "Integrator";
}
```

Kromě vestavěných elementů obsahuje jazyk XSLT velké množství funkcí, které slouží například k navigaci (odkaz na aktuálně zpracovávaný uzel či kořenový uzel) nebo k vygenerování klíče určeného k jednoznačné identifikaci uzlu v rámci dokumentu. Například pro odlišení názvu třídy je použita vestavěná funkce s názvem `generate-id(uzel)`.

Značná nevýhoda jazyka XSLT tkví ve složitosti šablon. Pokud provádíme pouze jednoduché transformace, je XSLT velmi účinným nástrojem. V komplexnějších případech je třeba zvážit použití programovacího jazyka spolu s parserem SAX či DOM (viz část 2.6.3).

2.6.4 Použití XML v grafických editorech

Dokument XML je dostatečně silný pro ukládání složených struktur. Není problém, aby blok mohl obsahovat další blok (po úpravě XSL schematu). To nám zaručí vytvářet XML dokumenty, které budou obsahovat libovolné množství zanořených objektů podobně jako je tomu ve složených DEVS modelech. Poté nezbývá než doplnit XSL schema tak, aby dokument obsahoval všechna potřebná data pro jeho zobrazení (typ bloku, pozici na plátně, atp.), definici jeho chování (atomický DEVS) a propojení (složený DEVS).

Pro převod modelu uloženého v prostředí grafického editoru do cílového simulačního jazyka lze využít XSLT. Pro některé simulační jazyky vytvoření šablony bude jednoduché, pro jiné příliš komplikované.

Dokument XML má především výhodu své čitelnosti. Jeho obsah lze poměrně snadno interpretovat. Ve srovnání s binárním souborem je použití XML mnohem otevřenější k manipulaci mimo software, pro který byl původně určen.

Kapitola 3

Návrh grafického editoru simulačních modelů

Uživatelské rozhraní grafického editoru simulačních modelů se skládá z plátna, knihovny modelů, panelu nástrojů, aplikačního menu a prohlížeče aktuálního modelu. Kromě toho aplikace uchovává vytvářený model a parser, který slouží k načtení a uložení výsledného modelu do XML.

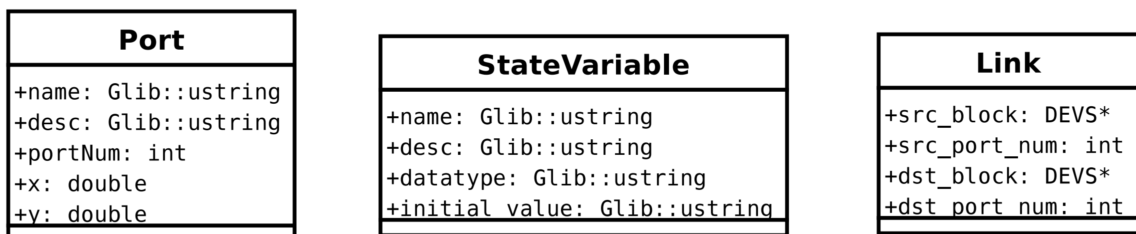
Na plátno se umisťují nové či předdefinované bloky a porty, navzájem se propojují a vytváří pomocí grafické reprezentace modelu vlastní model. Bloky, jejich spoje a porty lze mazat a případně přesouvat.

Nejdříve je nutné definovat třídy a objekty, které budou uchovávat samotný model.

3.1 Model

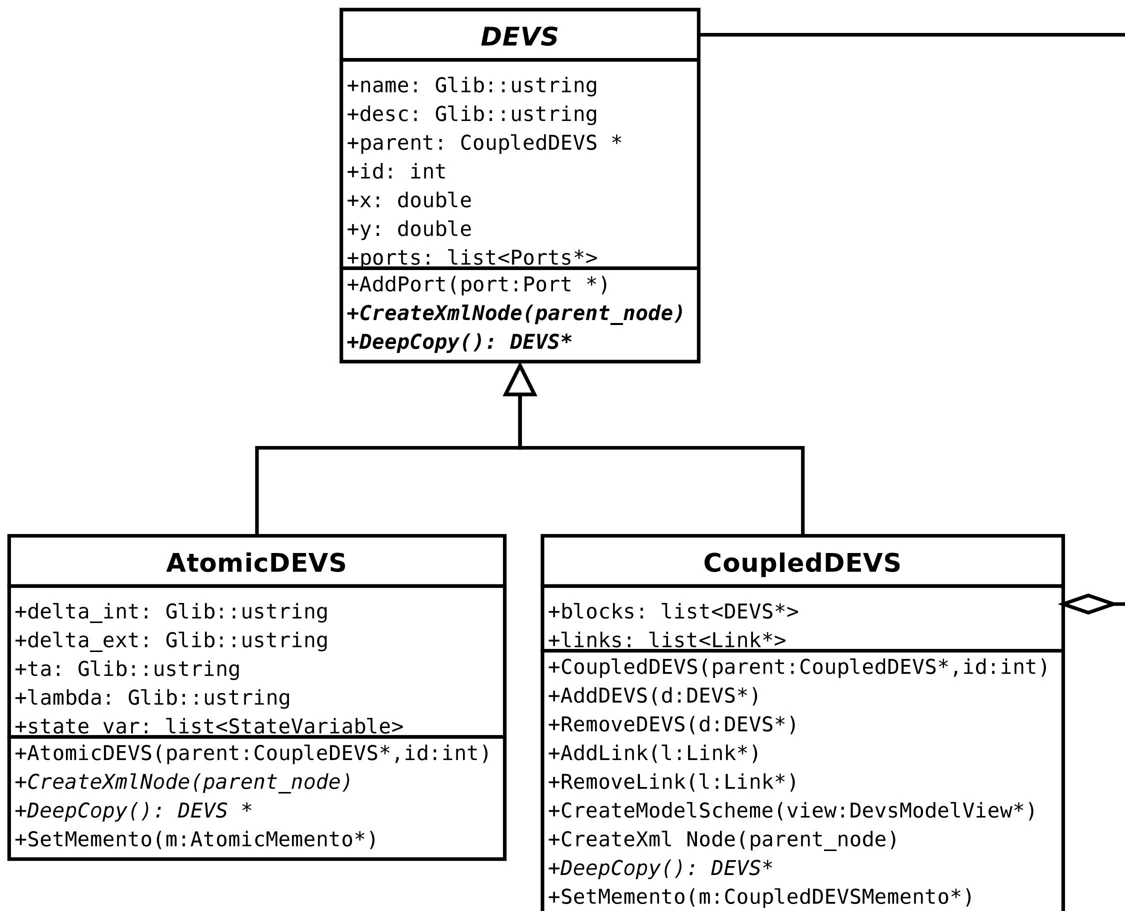
DEVS model je možné si na úrovni několika propojených atomických modelů představit jako orientovaný graf, kde uzly jsou atomické DEVS modely vybavené vstupními a výstupními porty, které hranami mezi sebou spojujeme. Z hlediska složených DEVS modelů pak jako strom, kde listy jsou atomické modely a ostatní uzly včetně kořenu složenými modely. Pro uložení této struktury proto použijeme strukturální návrhový vzor *Composite* [7] (viz obrázek 3.2).

3.1.1 Třídy pro reprezentaci modelu



Obrázek 3.1: Třídy použité v DEVS modelu

Abstraktní třída `DEVS` definuje rozhraní pro složený i atomický DEVS model, implementuje standardní chování společné pro oba typy modelů a definuje rozhraní pro přístup k



Obrázek 3.2: Diagram návrhového vzoru *Composite* pro uložení atomických a složených DEVS modelů

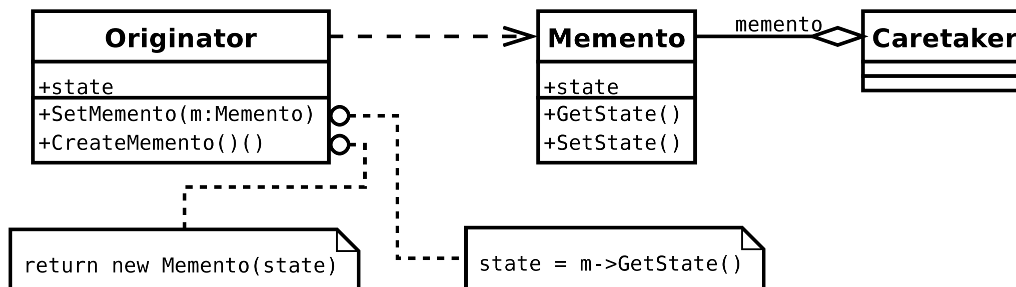
podmodelům. Abstraktní metoda `CreateXmlNode` slouží ke generování XML jazyka, který atomický či složený DEVS model popisuje. Používá se při ukládání modelu do souboru. Metoda `DeepCopy` slouží k vytvoření úplné kopie objektu včetně kopie všech vnořených objektů. Využívá se při kopírování objektu na plátno. Pro propojení je abstraktní třída `DEVS` vybavena porty (seznam `ports`). Každý port je definován názvem, pozicí a typem (vstupní–výstupní), diagram třídy určené pro reprezentaci portu se nachází na obrázku 3.1.

Třída `AtomicDEVS` nemůže mít žádný podmodel a obsahuje pouze chování specifické pro atomické DEVS modely (přechodové funkce, stavové proměnné, funkci posuvu času a výstupní funkci). Třída `StateVariable` (viz obrázek 3.1) je určena pro ukládání stavových proměnných modelu a obsahuje název proměnné, její popis, datový typ a počáteční hodnotu určenou pro inicializaci. Všechny stavové proměnné daného atomického modelu jsou uloženy v seznamu `state_vars`.

Třída `CoupledDEVS` definuje chování pro modely, které obsahují atomické či složené podmodely (seznam `blocks`). Objekty třídy `CoupledDEVS` tedy tvoří hierarchii – strom – na jehož listech se nacházejí atomické DEVS modely. Pro propojení jsou odvozené třídy od třídy `DEVS` vybaveny porty (seznam `ports`). Spojení jednotlivých portů (resp. DEVS modelů) je realizováno seznamem `links` objektů třídy `Link`. Ty obsahují ukazatel na zdrojový a cílový DEVS model a čísla portů, na který je spoj připojen.

Obě třídy obsahují metodu `SetMemento()`. Ta slouží k vytvoření objektu, který obsahuje taková data o objektech, aby pomocí nich bylo možné objekty znovu rekonstruovat. K tomu se využívá návrhový vzor memento (obrázek 3.3). Použijeme ho v případě, že potřebujeme uložit stav objektu k pozdější obnově bez toho, aby bylo nutné porušit zapouzdření objektu. Objekt třídy `Memento` ukládá vnitřní stav objektu `Originator`. Pro mementa existují dva typy rozhraní. Objekt `Caretaker` má přístupné úzké rozhraní, které mu umožňuje memento pouze předat jinému objektu. `Originator` má naopak přístup k celému mementu tak, aby mohl svůj stav pomocí něj obnovit. Objekt třídy `Originator` vytváří memento, které obsahuje jeho současný stav. Až je k tomu vyzván, využije memento k tomu, aby svůj stav obnovil. Objekt třídy `Caretaker` si vyžádá memento od objektu `Originator`, uloží jej a v případě potřeby zašle memento zpět objektu `Originator`.

Tento návrhový vzor se používá v pro implementaci funkce *Zpět/Znovu* a spolupracuje s návrhovým vzorem `Command`. V implementované aplikaci u objektů třídy `CoupledDEVS` obsahuje seznam `block` a `links`. Objekty `AtomicDEVS` vytváří memento s přechodovými funkcemi a stavovými proměnnými.



Obrázek 3.3: Třídy návrhového vzoru *Memento*

3.2 Pracovní plátno

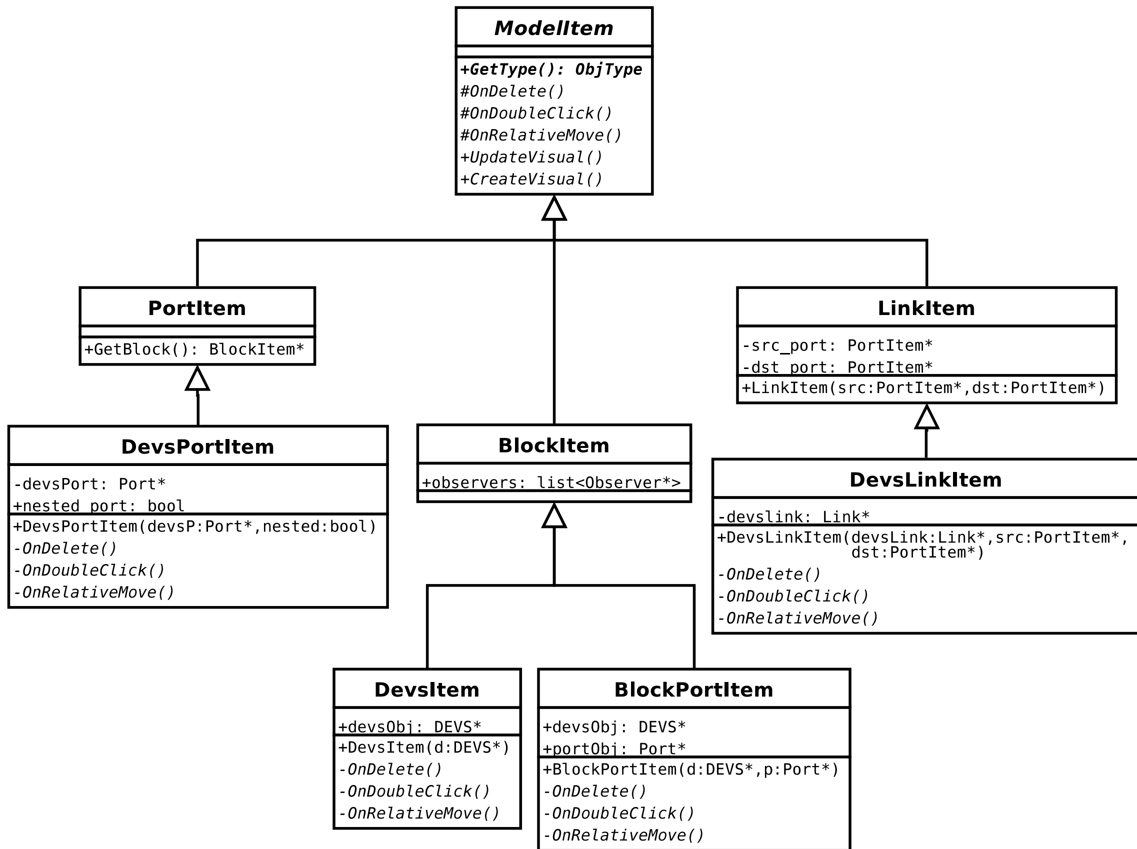
Pracovní plátno grafických editorů blokových schemat je plocha určená k vytváření, propojování a modifikaci bloků. Reaguje na události vzniklé vstupem klávesnice nebo polohovacího zařízení a obsahuje různé druhy elementů. Obecně lze plátno a práci s ním rozdělit na dvě části:

- objekty, které je možné na plátno umístit
- chování plátna při vstupech a událostech

3.2.1 Třídy objektů plátna

Na plátno je možné umístit bloky, vybaveny porty, které se propojují pomocí spojů. Tyto tři prostředky nám umožňují vytvářet síť atomických DEVS modelů. Pro vytváření složených DEVS modelů zavedeme ještě jeden speciální blok, který bude sloužit jako vstupní či výstupní port modelu, který vytváříme (ve složených DEVS modelech jsou to ty vnější vstupní a výstupní porty a spoje, které patří do množiny vnějšího spárování, viz 2.3.2).

Třídy pro reprezentaci objektů umístitelných na plátno se nachází na obrázku 3.4. Abstraktní třída `ModelItem` deklaruje základní rozhraní, pomocí kterého mohou odvozené objekty implementovat své chování při výskytu události mazání – `OnDelete()`, dvojklik –

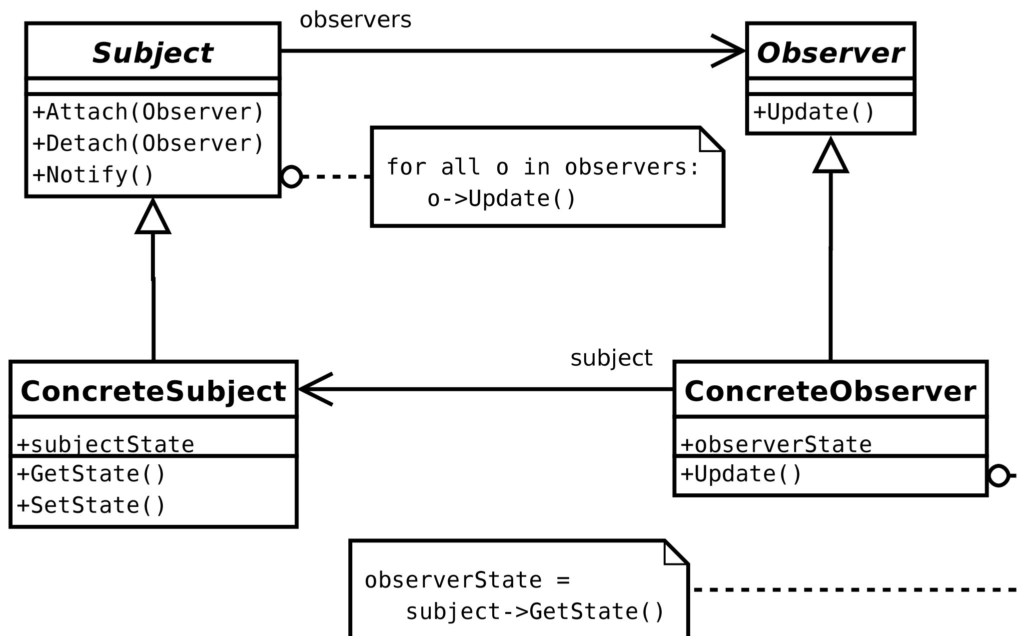


Obrázek 3.4: Diagram tříd určených k umístění na plátno

`OnDoubleClick()` a posun objektu – `OnRelativeMove()`. Abstraktní metoda `GetType()` slouží k určení typu objektu, rozeznáváme blok, spoj a port. Typy objektů se používají při vstupech k určování chování. Virtuální metody `CreateVisual()` a `UpdateVisual()` vytváří, resp. obnovují grafickou reprezentaci objektů, které lze umístit na plátno.

Odvozené třídy `BlockItem`, `LinkItem` a `PortItem` jsou obecnými třídami pro reprezentaci libovolných bloků, spojů a portů. Objekt třídy `PortItem` obsahuje odkaz na blok do kterého port patří. Spoje realizované pomocí objektu `LinkItem` určuje odkaz na zdrojový a cílový port. `BlockItem` je třída, která obsahuje porty.

K zajištění obnovování grafické reprezentace připojených spojů je použit návrhový vzor `Observer` – pozorovatel [7] (viz obrázek 3.5). Ten se používá v případě, že změna stavu jednoho objektu vyžaduje změnu stavu jiných objektů (jejich počet není předem znám). Rovněž lze vzoru použít v případě, kdy nechceme, aby objekty typu `Observer` nebyly příliš svázány s objektem, na kterém závisí. Objekt třídy `Subject` obsahuje rozhraní pro připojování a odpojování pozorovatelů (metody `Attach()`, `Detach()`). Třída typu `Observer` definuje rozhraní, které slouží k obnovování stavu objektem `Subject`. Objekty odvozené třídy `ConcreteSubject` obsahují libovolně dlouhý seznam připojených pozorovatelů a v případě požadavku jim odesílají zprávu o změně stavu (pomocí metody `Notify()`). `ConcreteObserver` obsahuje odkaz na `ConcreteSubject`, obsahuje stav, který má odpovídat stavu pozorovaného objektu a obsahuje implementaci funkce, která toto zajišťuje (`Update()`).



Obrázek 3.5: Návrhový vzor pozorovatel (*Observer*) a spolupracující třídy

Spolupráce mezi třídami návrhového vzoru pozorovatel vypadá následovně:

- Objekt `ConcreteSubject` upozorní všechny svoje pozorovatele pokaždé, když dojde ke změně stavu, který pozorovatele zajímá.
- Jakmile pozorovatel je upozorněn na změnu stavu, může požádat o informace objekt `ConcreteSubject` a využít je k obnovení vlastního stavu.

Návrhový vzor pozorovatel lze rozšířit o typ zprávy, která je pozorovatelům rozesílána. Pokud totiž máme různé odvozené objekty od třídy `Observer`, které jsou svázány s různými proměnnými či stavy objektu `ConcreteSubject`, pak by při volání metody `Notify()` byli obnoveni i ti pozorovatelé, kterých se změna stavu netýká. Pokud však je metoda `Update` volána s parametrem, který vymezí cílovou skupinu pozorovatelů, provedou obnovení stavu pouze ti pozorovatelé, kterých se cílová skupina týká. To je využito ve třídě `DevsModelView` popsané v sekci 3.2.4.

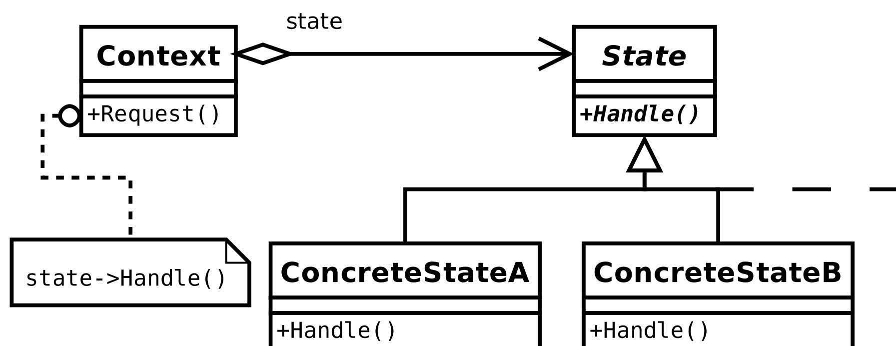
3.2.2 Odvozené třídy pro potřeby DEVS modelu

Pro připojení dat souvisejících s modelováním DEVS systému jsou odvozeny třídy `DevsItem`, `BlockPortItem`, `DevsLinkItem` a `DevsPortItem`. Každá třída obsahuje odkaz na odpovídající objekt v DEVS modelu (viz 3.1). Ten slouží k získávání atributů objektu a jejich zpětnému ukládání. Kromě toho objekty implementují abstraktní metody nadřazené třídy `ModelItem` (`OnDelete()`, `OnDoubleClick()` a `OnRelativeMove()`), které interpretují události vzniklé editací na plátně vlastnímu modelu.

3.2.3 Chování pracovního plátna

Chování pracovního plátna je dáno interpretací vstupů, které plátno přijímá pomocí polohovacího zařízení. Během editace se plátno dostává do několika stavů, během kterých na

stejné události reaguje odlišným způsobem. Například během přesouvání objektu dojde při přerušení stisku levého tlačítka myši k aktualizaci pozice v objektu, zatímco u označování výběrem k selekci odpovídajících bloků. K tomu je možné použít návrhový vzor stav (*State*) [7].



Obrázek 3.6: Návrhový vzor stav (*State*) a spolupracující třídy

Ten se používá v případech kdy chování objektu závisí na jeho stavu definovaném během spuštění programu. Rovněž je vhodný pro operace, které jsou řešeny rozsáhlými podmíněnými výrazy kde je stav reprezentován jedním či více výčtovými typy.

Diagram tříd návrhového vzoru stav se nachází na obrázku 3.6. Kontext (třída `Context`) obsahuje současný stav jako instanci některé odvozené třídy `State` a definuje rozhraní pro klienta. Třída `State` definuje rozhraní pro zapouzdření chování určené stavem ve třídě `Context`. Její odvozené třídy `ConcreteState` implementují samotné chování určené stavem třídy `Context`.

Třídy spolupracují tak, že `Context` požádá instanci `ConcreteState` o zpracování současného stavu. Kontext může stavovému objektu předat odkaz na sám sebe, aby stavový objekt mohl v případě požadavku přistupovat přímo k jeho metodám a proměnným. Vzhledem k tomu, že kontext je primárním rozhraním pro klienta, tak by klient za běhu neměl se stavovými objekty manipulovat. Třídy `Context` a `ConcreteState` mohou určovat jaký stav následuje a za jakých podmínek ke změně dojde.

Použití návrhového vzoru stav je výhodné především pro odstranění implementace chování pomocí dlouhých podmíněných výrazů, které je těžší modifikovat a rozšiřovat. Rovněž eliminuje potřebu pomocných proměnných, které se využívají pouze v určitých stavech. Rozšíření kontextu o další stavy je poměrně jednoduché. Stačí vytvořit novou třídu odvozenou od `State` a určit, kdy ke změně na nový stav dojde. Nevýhodná může být nutnost vytvoření stavových tříd, v případě velkého počtu stavů to však eliminuje dlouhé rozhodovací bloky, které znesnadňují čitelnost programu.

Diagram tříd stavů pracovního plátna se nachází na obrázku 3.7. Abstraktní třída `ModelViewState` definuje rozhraní, kde abstraktní funkce budou definovat chování při stisku a uvolnění tlačítka, při výskytu události upuštění během *Drag&Drop* a při pohybu myši nad plátnem. Ve stavu `DefaultState` je možné označovat objekty a přejít do všech ostatních stavů v závislosti na pohybu myši, stisknutým tlačítkům a objektu, který se nachází pod ukazatelem myši.

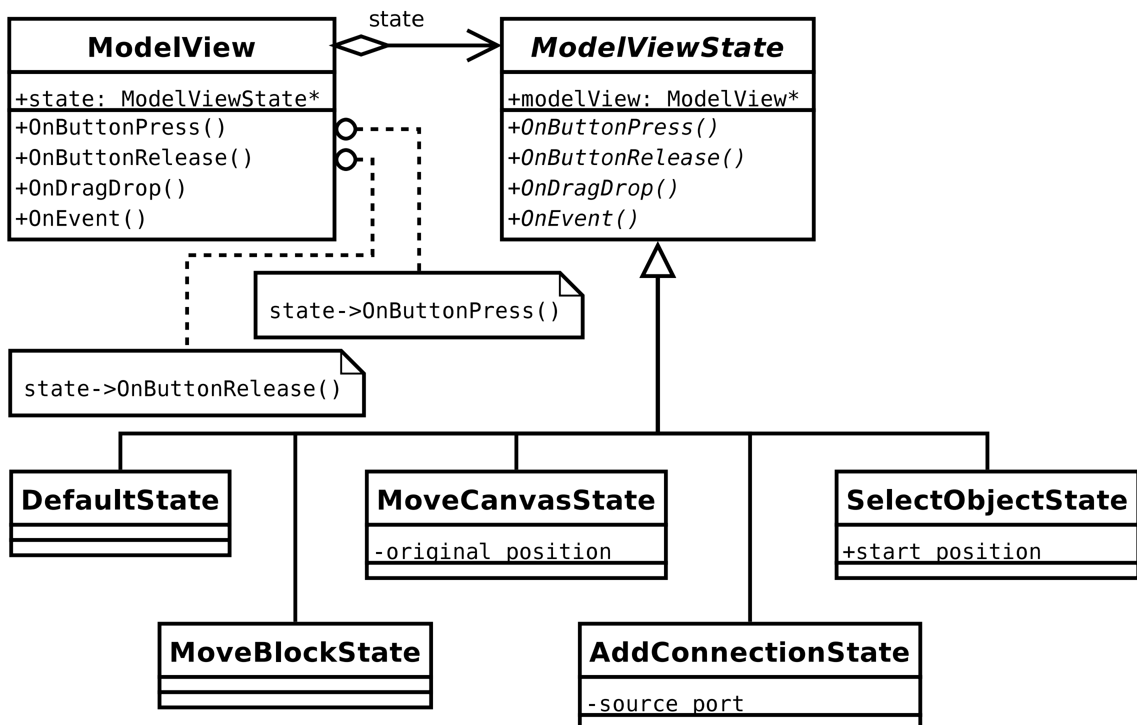
Stav `MoveBlockState` slouží k přesunu jednoho či více objektů a uchovává původní pozici objektu. Po dokončení přenosu (uvolněním tlačítka myši) se volá obsluha události `OnMove()` určená k aktualizaci DEVS modelu a uložení původní polohy pro funkci *Zpět*.

`MoveCanvasState` je jednoduchá stavová třída, která slouží k posunu pracovního plátna. Nový stav je vybrán proto, aby nedocházelo ke kolizím při výskytu jiné události (například stisku jiného tlačítka).

Třída `AddConnectionState` slouží k řízení tvorby spojů. Je vytvořena pokud dojde ke stisknutí levého tlačítka myši nad portem. Obsahuje proměnnou, která je odkazem na zdrojový port. Během vytváření zobrazuje na plátně úsečku reprezentující vytvářený spoj. Pokud je tlačítko uvolněno nad jiným portem, je zavolána funkce pro vytváření spojů.

`SelectObjectState` slouží k výběru objektů pomocí obdélníku, který je určen stisknutím, tahem a uvolněním levého tlačítka myši. Stisknutí musí proběhnout na pracovním plátně na místě, kde se nenachází žádný objekt a zaznamená se při něm počáteční pozice. Po uvolnění se vytvoří seznam objektů, které se nacházejí v definovaném obdélníku a z nich se vytvoří výběr.

Všechny stavy se po dokončení operace vracejí do původního stavu `DefaultState`.



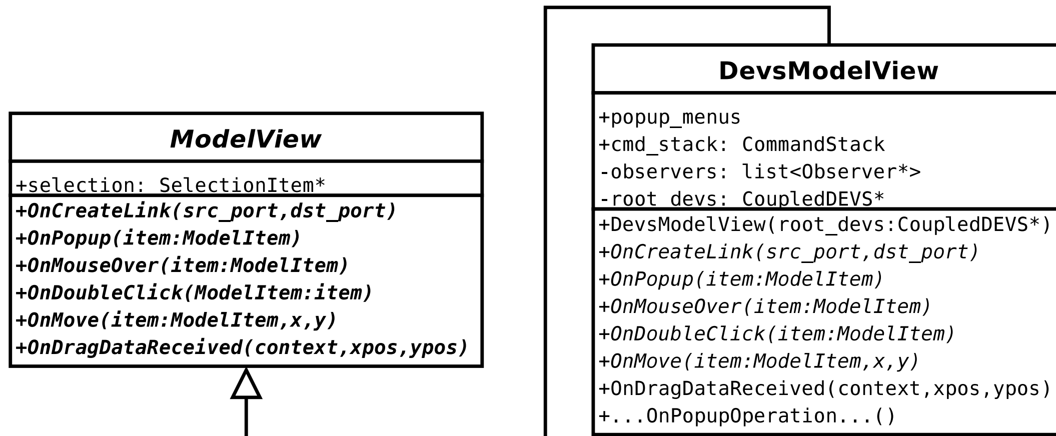
Obrázek 3.7: Diagram tříd určující chování podle návrhového vzoru stav

3.2.4 Třídy pro vlastní plátno

S využitím všech předchozích tříd lze definovat pracovní plátno (diagram tříd na obrázku 3.8). Abstraktní třída `ModelView` se skládá ze stavu plátna (odkaz na objekt třídy odvozené z `ModelViewState`) a definovaného rozhraní, které tvoří metody určené pro obsluhu vzniklých událostí. Rozeznáváme následující události:

- Požadavek na vytvoření spoje (metoda `OnCreateLink()`)
- Výskyt ukazatele myši nad objektem (`OnMouserover()`)

- Dvojité klepnutí tlačítka myši nad objektem (`OnDoubleClick()`)
- Po přesunu objektu (`OnMove()`)
- Objekt přijatý pomocí *Drag&Drop* (`OnDragDataReceived()`)
- Požadavek na zobrazení kontextové nabídky (`OnPopup()`)



Obrázek 3.8: Třídy *ModelView* a *DevsModelView* určené pro zobrazení modelu na kreslící plátno

Všechny tyto metody jsou implementovány v odvozené třídě `DevsModelView`, která slouží jako pracovní plátno pro vytváření a modifikaci DEVS modelu. Při výskytu požadavku na vytvoření spoje je nejdříve zjištěno, zda jej lze realizovat (zdrojový a cílový port jsou různého typu a nejsou oba v rámci jednoho bloku) a až poté je spoj vytvořen.

Pro snadnou orientaci v modelu při výskytu ukazatele myši nad objektem se o něm zobrazí detailní informace. Dvojité klepnutí myší na objekt slouží k vykonání akce nad objektem. Reprezentuje-li objekt atomický DEVS, zobrazí se formulář s názvem bloku, popisem, stavovými proměnnými a chováním. V případě, že jde o složený DEVS, zobrazí se jeho vnitřní bloky a spoje.

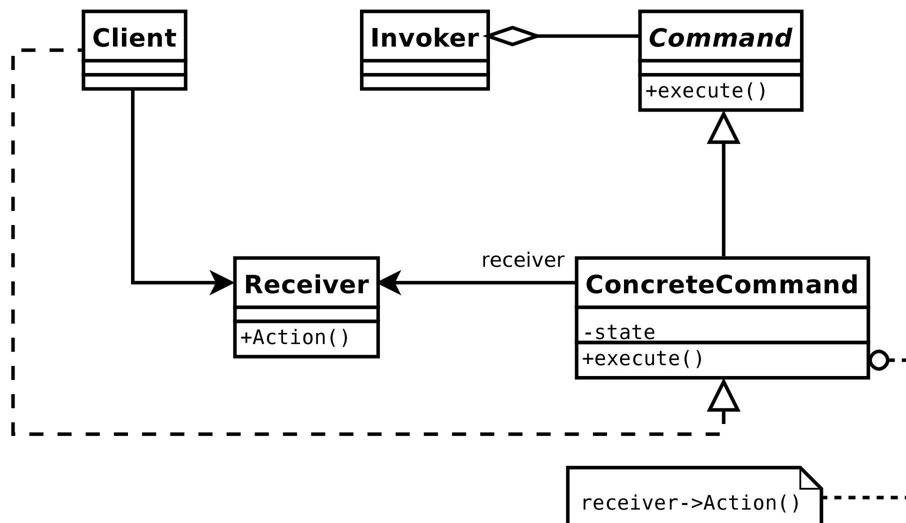
Kromě toho třída `DevsModelView` obsahuje prostředky pro vytváření kontextových nabídek a obsluhu jednotlivých jejich položek. Pomocí nich lze vytvářet nové atomické a složené DEVS modely, ukládat je do knihovny modelů, mazat, kopírovat, vkládat a měnit atributy bloků, portů a spojů.

Pro potřeby vytváření a navigace obsahuje třída ukazatel na složený DEVS, jehož podmodely jsou zobrazeny na plátně. Pracovní plátno rovněž sledují objekty typu pozorovatel (viz 3.2.1). Pokud například dojde ke změně struktury modelu, je obnoven strom zobrazující aktuální model a jeho podmodely.

3.2.5 Třídy pro podporu vracení změn

Pro implementaci funkce *Zpět* je třída `DevsModelView` rozšířena o objekt `CommandStack`. Obsahuje objekty podtřídy `Command`, která vychází z návrhového vzoru příkaz (*Command*) [7].

Třída `Command` deklaruje rozhraní pro provedení operace, odvozené třídy jako je například `ConcreteCommand` definují spojení mezi objektem příjemce (`Receiver`) a operací. Metoda



Obrázek 3.9: Diagram návrhového vzoru *Command*

`Execute()` obsahuje volání operace prováděné na příjemci. `Client` vytváří objekt typu `ConcreteCommand` a nastavuje atribut `receiver`. Objekty třídy `Invoker` požádají příkaz o vyřízení požadavku a příjemce `Receiver` implementuje operace spojené s jeho vyřízením.

Spolupráce probíhá tak, že klient vytvoří instanci třídy `ConcreteCommand` a určí jejího příjemce. Objekt `Invoker` jej uloží a zařídí požadavek na jeho spuštění. V případě, že důsledky příkazu lze vrátit zpět, předtím `ConcreteCommand` uloží původní stav objektu k pozdějšímu obnovení, a vyřídí samotný požadavek zavoláním metody `Execute()`.

K implementaci funkce *Zpět* je nutné rozšířit třídu `Command` o metodu `UnExecute()`, která provede za pomocí uloženého předchozího stavu objekt do předchozí podoby. Funkce *Zpět* může být jednoúrovňová (stačí ukládat instanci posledního příkazu) nebo víceúrovňová. Při té je nutné zavést seznam historie, kam se ukládají všechny volané příkazy. Zpětným (dopředným) průchodem se pak realizuje zrušení (obnovení) provedených akcí.

Návrhový vzor `Command` odděluje objekt, který požaduje vykonání nějaké operace od objektu, který tuto operaci provádí. Dále umožňuje vytvářet složené příkazy (makra) a v neposlední řadě usnadňuje přidávání nových příkazů, protože není třeba měnit existující třídy.

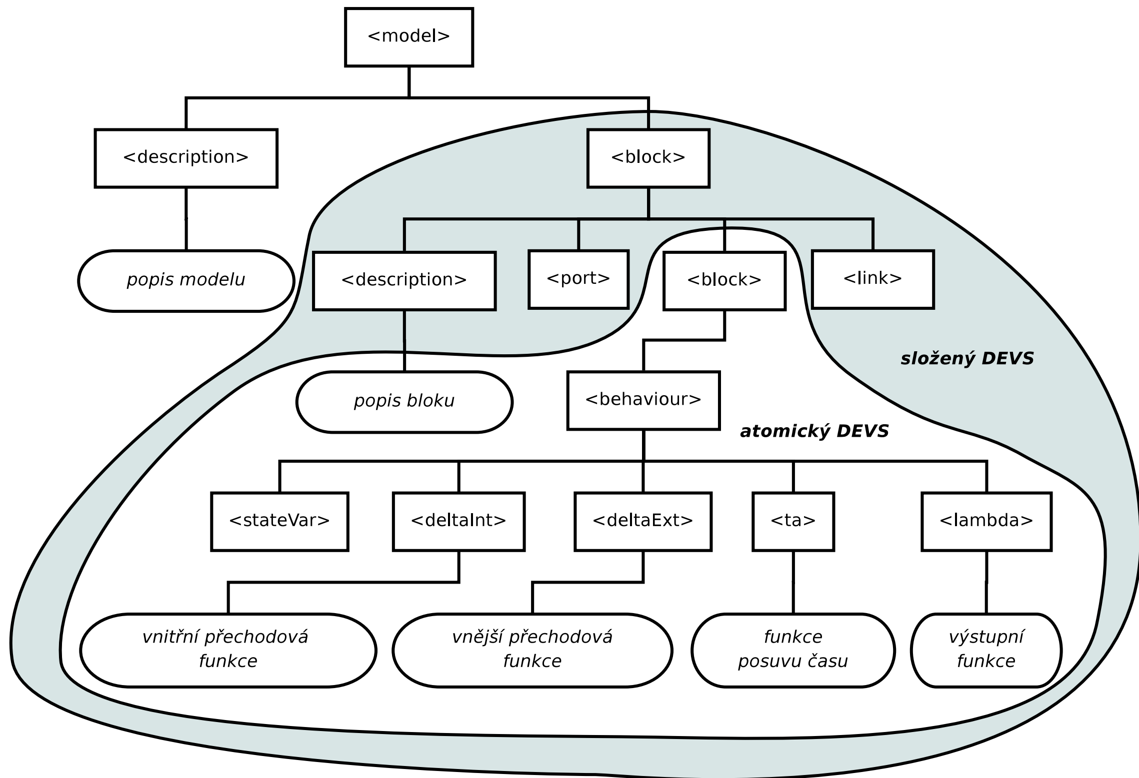
3.2.6 Možnosti využití pro jiné druhy modelů

Třídy byly navrženy tak, aby bylo možné vytvořit i jiné typy objektů než jsou DEVS modely. To je dáno oddělením implementace chování modelu pomocí třídy `ModelView`, která na zpracování stavů používá odvozené třídy `ModelViewState`. Odvozená třída `DevsModelView` obsahuje proměnné specifické pro DEVS modely. Rovněž jsou odděleny objekty plátna (`BlockItem`, `PortItem`, `LinkItem`) od odvozených tříd určených pro DEVS (`DevsBlockItem`, `DevsPortItem` a `DevsLinkItem`).

3.3 Návrh formátu souboru

Vytvoříme-li model, je vhodné jej uložit do souboru. Proto aby byl model snadněji čitelný a bylo možné ho zpracovávat pomocí XSLT šablon, použijeme formát XML. Nejdříve de-

finujeme popis formátu v jazyce XSL, který bude popisovat strukturu uloženého modelu. XML Schema se nebude příliš lišit od vlastní reprezentace modelu v paměti. Kompletní XML schema se nachází v příloze. Zde bude popsána pouze základní struktura souboru. Na obrázku 3.10 se nachází příklad objektového modelu dokumentu (*Document Object Model - DOM*).



Obrázek 3.10: Příklad objektového modelu dokumentu s jedním složeným a atomickým DEVS modelem

Kořenový element `model` může obsahovat pouze jeden element `block` a popis vlastního modelu prostřednictvím elementu `description`. Každý blok, který reprezentuje atomický či složený DEVS obsahuje atributy název (`name`), typ (`type`), identifikátor (`id`) a pozici na plátně (`x, y`). Typ určuje, který model blok popisuje, a může nabývat hodnot `atomic` (atomický DEVS model) nebo `coupled` (složený DEVS model). Blok rovněž obsahuje elementy, které určují jakými porty je vybaven. Element `port` má atributy určující název `name`, pozici `pos`, typ `type` a popis portu `desc`. Typ může nabývat hodnot vstupní nebo výstupní (`input, output`). K popisu funkce bloku slouží element `description`.

Bloky definující atomický DEVS obsahují popis modelu a jeho chování je umístěné v elementu `behaviour`. To se skládá z elementů stavových proměnných `stateVar` a elementů, které obsahují části kódu specifikující chování atomického DEVS modelu: interní a externí přechodové funkce `deltaInt`, `deltaExt`, funkce posuvu času `ta` a výstupní funkce `lambda`. Stavové proměnné obsahují atributy název (`name`) a popis (`desc`) proměnné, počáteční hodnotu (`initialValue`) a datový typ (`type`).

Popis složeného DEVS modelu se skládá z vnořených elementů `block`, které mohou být jak atomického, tak složeného typu. Pro propojení se využívá elementu link. Ten obsa-

huje atributy identifikátoru zdrojového (cílového bloku `fromId` (`toId`) a číslo zdrojového (cílového) portu `fromPortNum` (`toPortNum`).

3.3.1 Knihovna modelů

Abychom nemuseli vždy vytvářet úplně celý model, zavedeme knihovnu modelů. Ta bude obsahovat atomické či složené DEVS modely, které bude možné vkládat libovolně do modelu. Zároveň bude možné uložit vybraný DEVS model do knihovny a tím ji rozšiřovat.

Uložení modelu v knihovně se bude lišit pouze v umístění souboru, který model v XML reprezentaci obsahuje. Soubory s knihovními modely se nacházejí v uživatelsky definovaném adresáři. Do aplikace je načten rekurzivně celý adresář, takže je možné a doporučené používat podadresáře jako větvení modelů podle typu, případně podle simulačního jazyka či knihovny pro kterou jsou určeny.

K obnovování obsahu knihovny dojde na požádání zavoláním metody `Update()`. Knihovna je totiž pozorovatel připojený ke třídě hlavní aplikace. Ten metodu zavolá, pokud byl uložen nějaký objekt do knihovny.

Pro přidání nového bloku z knihovny modelů na plátno se použije funkce známá jako *Drag&Drop*. Pomocí levého tlačítka myši se vybere objekt z knihovny, tím se uchopí, přesune se na plátno a na místo, kde dojde k uvolnění tlačítka myši se objekt umístí. Umístění objektu spočívá v jeho načtení z XML souboru, který blok popisuje a přidání do seznamu bloků aktuálního modelu.

3.4 Hlavní aplikace

Hlavní aplikace se skládá z uživatelského rozhraní a metod, které implementují chování jeho jednotlivých událostí. Události vznikají výběrem položky v nabídce, kliknutím na tlačítka atp a slouží ke změně měřítka zobrazení modelu, mazání, vyřezávání, kopírování a vkládání bloků a jejich spojů.

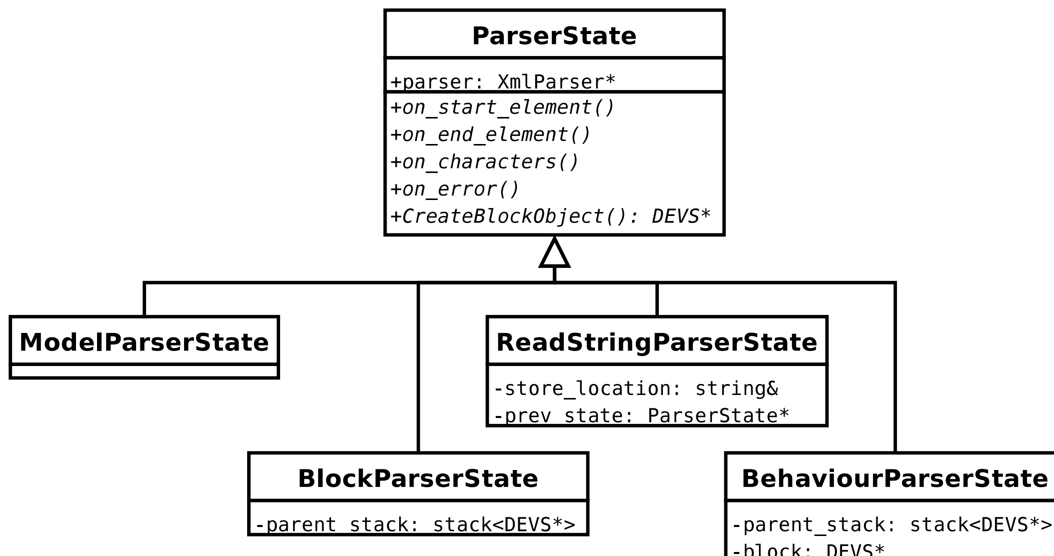
3.4.1 Operace se soubory

Soubor, který obsahuje validní model uložený pomocí jazyka XML, je možné otevírat a ukládat. Čte se pomocí XML parseru a zapisuje metodou `CreateXmlNode()`, která je deklarovaná v objektu `DEVS`.

Parser dokumentu uloženém v XML je řešen pomocí SAX (viz 2.6.3). Pro implementaci obslužných funkcí (*callbacků*) je použit návrhový vzor stav (3.2.3 a obrázek 3.11).

Třída `ParserState` definuje rozhraní pro SAX parser. Metoda `on_start_element()` se volá při výskytu libovolného elementu, `on_end_element()` při jeho ukončení. Pro zpracování obsahu elementu je volána metoda `on_character()`. V případě výskytu chyby SAX parser provede operace v metodě `on_error()`.

Počáteční stav při čtení souboru je `ModelParserState`. Ke čtení obsahu elementu slouží `ReadStringParserState`, který se používá k získání popisu modelu. Výskytem elementu `block` dojde ke změně stavu na `BlockParserState`. Podle hodnoty atributu `type` se rozezná, zda jde o atomický či složený DEVS model. V případě atomického se přečte pomocí stavu `BehaviourParserState` celý atomický blok. V opačném případě se přečtou rekurzivně za pomoci zásobníku `parent_stack` všechny podmodely, které zpracovávaný složený DEVS model obsahuje.



Obrázek 3.11: Aplikace návrhového vzoru *stav* pro potřeby parseru SAX

Ukazatel na objekt třídy `XmlParser` plní funkci objektu `Context` (viz obrázek 3.6) a využívá se k vytváření objektu `Document`, který po úspěšném načtení vstupního souboru obsahuje kompletní DEVS model. Kromě toho dokument obsahuje název souboru a metody pro jeho uložení.

3.4.2 Export modelu

Model je možné exportovat pomocí modulů vytvořených uživatelem. Moduly se nachází v předdefinovaném adresáři a načítají se při startu programu. Každý modul se skládá z funkce `get_module_info()`, která vrací strukturu `module_info`. Ta obsahuje název modulu, který je použit v menu programu a typu. Typ modulu určuje způsob výběru cílového objektu. Pokud modul generuje více souborů stanovuje se jeho typ tak, aby při vyvolání exportu byl uživatel vyzván k výběru složky. V případě, že generuje jeden soubor je typ nastaven tak, aby byl uživatel vyzván k zadání názvu souboru.

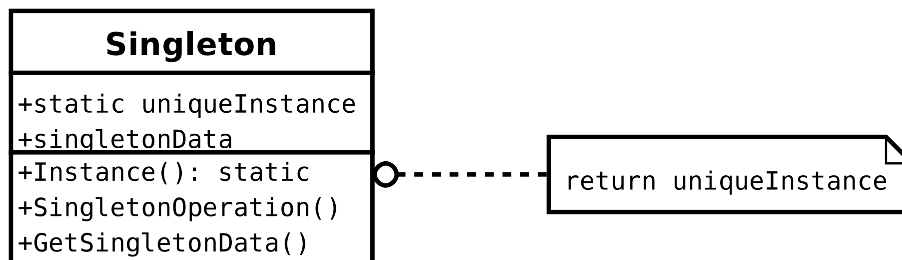
Při aktivaci exportního modelu je zavolána funkce `export_func` s parametry cesta či název výstupního souboru a ukazatel na kořenový objekt aktuálního modelu typu `DEVS`. Implementace této funkce pak vytváří výstupní data.

Z hlediska implementace exportní funkce je možné uvažovat dva přístupy. Lze použít XSLT transformaci tak, že z předaného objektu třídy DEVS vytvoříme XML soubor pomocí metody `CreateXmlNode()`, na který aplikujeme XSL transformaci s vlastní XSLT šablonou a díky tomu nám vznikne zdrojový kód určený pro specifikovanou simulační knihovnu či jazyk. Druhou možností je postupný rekurzivní průchod modelem, během kterého se generuje výsledný soubor či soubory například pomocí standardní funkce `fprintf()`.

Dalším z aspektů tvorby modulů je způsob jejich připojení k aplikaci. Je možné moduly přímo zakomponovat do výsledné aplikace nebo je nahrávat dynamicky. Zakomponované moduly vyžadují při každé změně (přidání nového modulu, úprava modulu) nutnost přeložit kompletní aplikaci. Dynamické moduly obsahují kód, který je možné spouštět za běhu aplikace a překládají se jako sdílené knihovny.

3.4.3 Správa nastavení

K uchování uživatelsky definovaných nastavení jako je vzhled bloku, velikosti písma na plátně, specifikace umístění knihovny modelů a exportních modulů slouží třída `Settings`. Proto, aby byla kdykoliv dostupná a aplikace obsahovala pouze její jednu instanci, je použit návrhový vzor jedináček (*Singleton*) [7] zobrazený na obrázku 3.12.



Obrázek 3.12: Návrhový vzor jedináček (*Singleton*)

Třída `Singleton` definuje metodu `Instance()`, která dovolí vytvoření a zpřístupnění pouze jedné instance v rámci celé aplikace. Je to metoda třídy, nikoliv objektu (v jazyce C++ se jedná o statickou metodu). Klient přistupuje k objektu pouze pomocí metody `Instance()`. Tento návrhový vzor řeší zahlcení programu globálními proměnnými. Proto aby nebylo možné vytvořit objekt `Singleton` jinak než pomocí metody `Instance()`, je konstruktor chráněný (*protected*).

Třída `Settings` je rovněž vybavena metodami k načtení (uložení) konfiguračních informací z (do) XML souboru a ve spolupráci s uživatelským rozhraním je schopna nabídnout možnosti nastavení uživateli.

3.4.4 Grafické uživatelské rozhraní

K vytváření uživatelského rozhraní, jako jsou formuláře, jejich vyplňování a vyhodnocování a ostatní prvky slouží třída `UI`. Vzhledem k tomu, že okno, ve kterém se pracuje, je pouze jedno, je k implementaci použit návrhový vzor jedináček (viz 3.4.3) a proto je třída snadno přístupná všem třídám, které ji ke své funkce vyžadují. Například dojde-li k výskytu události změny parametrů atomického DEVS modelu, je třídou `UI` vytvořen dialog obsahující současné parametry modelu (název, stavové proměnné a funkce s jeho chováním). Pokud je změna potvrzena, tak třída `UI` změní parametry modelu tak, aby odpovídaly nově zadaným hodnotám. Třída rovněž implementuje metody, pomocí kterých se přistupuje k jednotlivým prvkům grafického uživatelského rozhraní (například pro potřeby připojení obslužných metod k událostem vzniklých při aktivaci tlačítek).

Kromě toho obsahuje prostředky pro vytváření uživatelského rozhraní pomocí externího souboru, který rozhraní popisuje a zpřístupňuje prvky rozhraní jako jsou kontextové nabídky (*popup menu*).

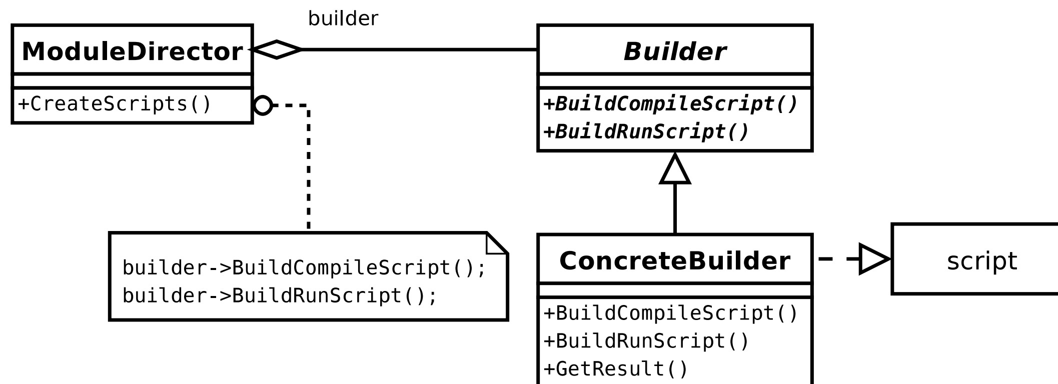
3.5 Rozhraní pro řízení běhu simulace

Pomocí výše uvedených modulů pro převod modelu do cílové simulační knihovny či jazyka jsme schopni vygenerovat zdrojový kód, který je pro ně určený. Abychom mohli samotnou

simulaci spustit, je nutné nejdříve zdrojový kód přeložit. Nyní navrhneme moduly pro překlad a spuštění simulace. Modul se skládá ze dvou částí:

- postupu pro přeložení zdrojového kódu modelu
- postupu pro spuštění simulace

Obě části reprezentuje skript, který požadovanou úlohu řeší. Aplikace tyto skripty od modulu získá a spustí. Toto může řešit návrhový vzor stavitel – *Builder* [7] (diagram na obrázku 3.13).



Obrázek 3.13: Diagram rozhraní pro řízen běhu simulace pomocí návrhového vzoru *Builder*

Abstraktní třída `Builder` definuje rozhraní pro vytváření výstupních objektů. Objekty třídy `ConcreteBuilder` toto rozhraní implementují a vytváří tak skripty potřebné pro kompilaci modelu a spuštění simulace. Mezi aplikací a stavitelem je třída, která vytváření výstupních objektů řídí – `ModuleDirector`.

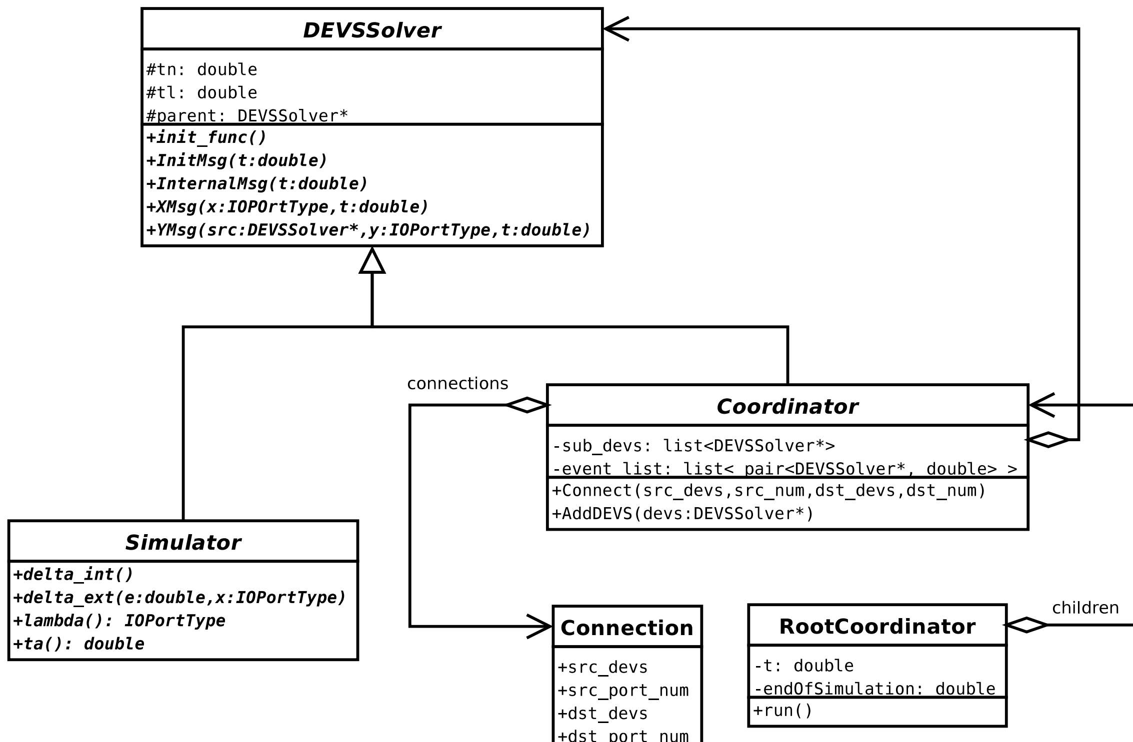
Aplikace nejdříve vytvoří instanci třídy `ModuleDirector` a předá mu instanci objektu třídy `ConcreteBuilder`, pomocí kterého se bude výstupní objekt – `Script` konstruovat. `ModuleDirector` požádá stavitele o vytvoření specifikované části produktu, ten žádost vyřídí a požadovanou část vytvoří. Poté aplikace může získat produkt pomocí metody `GetResult()`. Produktem je v tomto případě skript, který řeší požadovanou úlohu.

3.6 DEVS simulátor

Pro potřeby demonstrace výsledků a ověření funkčnosti byl zvolen přístup konstrukce vlastního DEVS simulátoru uvedeného na obrázku 3.14.

Abstraktní třída `DEVSSolver` obsahuje rozhraní společné pro koordinátor a simulátor. Proměnná *tn* (*tl*) označuje čas vzniku další události (poslední události), *parent* je ukazatel na nadřazený koordinátor. Metoda `init_func` slouží k počátečnímu nastavení stavů simulátorů a vytvoření propojení v koordinátorech. `InitMsg`, `InternalMsg`, `XMsg` a `YMsg` jsou metody pro obsluhu jednotlivých zpráv.

Odvozená, ale stále abstraktní třída `Simulator` obsahuje implementaci simulaci chování atomického DEVS modelu pomocí metod `delta_int`, `delta_ext`, `lambda`, `ta`. Abstraktní metody zděděné po `DEVSSolver` obsahují chování při výskytu jednotlivých zpráv. V případě potřeby volá přechodové funkce a s výstupní událostí kontaktuje nadřazený koordinátor.



Obrázek 3.14: Digram tříd pro simulátor DEVS modelu

Koordinátor obsahuje seznam komponentů `sub_devs`(simulátorů nebo koordinátorů) a jejich propojení `connections`. Kalendář událostí `event_list` je realizován seřazeným seznamem párů *komponenta–čas výskytu události*. Pro přidání nového simulátoru či koordinátoru do seznamu komponent slouží metoda `AddDEVS`. Propojování jednotlivých bloků mezi sebou a mezi vlastním koordinátorem se provádí pomocí `Connect`.

Kořenový koordinátor `RootCoordinator` obsahuje proměnnou `t`, která určuje aktuální simulační čas. Odkaz na koordinátor slouží k rozeslání inicializační zprávy, zprávy o provedení interního přechodu a k získání času nejbližší události pro potřebu aktualizace globálního simulačního času. Koordinátor rovněž rozhoduje, za jakých podmínek se simulace ukončí.

Vlastní model vytvoříme odvozením tříd `Simulator` a `Coordinator`. Definujeme jim chování podle našeho DEVS modelu, propojíme je podle sítě, ve které mají být zapojeny, a nastavíme koncový čas simulace v kořenovém koordinátoru. Poté je možné model spustit. Výsledky, které mají být později zpracovány, musí generovat některý ze simulátorů (nemusí být pouze jeden) například do souboru.

Kapitola 4

Implementace editoru simulačních modelů

Pro svou objektovou orientaci byl k implementaci grafického editoru simulačních modelů zvolen jazyk C++ a vhodné knihovny pro urychlení vývoje aplikace. Byl kladen důraz na výběr volně dostupných svobodných knihoven s vhodnou licencí. Zvolená hlavní vývojová platforma je operační systém Linux, řešení je však možné přeložit i na jiných operačních systémech. Při vývoji byl použit překladač jazyka C++ *GNU GCC* verze 4.1.3.

4.1 Knihovna pro prvky uživatelského rozhraní

Pro implementaci prvků uživatelského rozhraní byla použita knihovna *gtkmm* [12], která adaptuje rozhraní knihovny *GTK* [11] do jazyka C++. *GTK* je napsáno v jazyce C, ale je navrženo podle objektového paradigmatu a proto vytvoření wrapperu (adaptéru) pro libovolný jazyk není příliš složité. Existuje například *PyGTK* (wrapper knihovny *GTK* pro jazyk Python) nebo *php-gtk* (totéž pro jazyk PHP). Výbava knihovny *GTK* je poměrně dostačující, obsahuje všechny základní prvky uživatelského rozhraní jako jsou například okna, dialogy, tlačítka, formuláře, stromové seznamy, ale neobsahuje žádnou formu pracovního plátna (*canvas*)dostatečnou pro výrobu blokových schemat. K dispozici je plátno vybavené pouze základními operacemi a nedovoluje složitější operace jako je například změna měřítka. Tuto knihovnu jsem nakonec zvolil především z osobních důvodů, protože se v ní nejlépe orientuji. Kromě toho, že je postavená na kvalitním a odladěném *GTK*, usnadňuje definici uživatelského rozhraní pomocí knihovny *libglade* (více v 4.3).

4.2 Výběr knihovny pro implementaci pracovního plátna

K implementaci pracovního plátna bylo možné použít knihovnu *Cairo* s velice kvalitním výstupem. Její nevýhodou je však rozhraní na nižší úrovni abstrakce, což by implementaci plátna určeného pro editor zkomplikovalo. Další možností bylo použít zastaralejší *Gnome-Canvas*, který ovšem nezvládá zobrazení textu při změně měřítka. Nad *Cairo* jsou postaveny knihovny s abstraktnějším rozhraním *Goocanvas* a *Papyrus* [13]. *Goocanvas* je v jazyce C, což je z hlediska návrhu nevyhovující. Knihovnu *goocanvasmm* adaptující toto rozhraní do jazyka C++ se mně však nepodařilo přeložit. Proto padla volba na posledního kandidáta a to knihovnu *Papyrus* [13]. Ta se na první pohled zdála jako funkční a kvalitní alternativa předchozím knihovnám, nicméně během implementace jsem narazil na poměrně

velké množství problémů, kdy některé metody této knihovny vracely špatné hodnoty. Proto bylo třeba část knihovny přepsat. Díky návrhu samotné knihovny a síle jazyka C++ bylo přepracování těchto metod poměrně jednoduché. Tyto obtíže však implementaci značně zdržovaly a proto použitou knihovnu *Papyrus* není možné příliš doporučit. Chyby, na které jsem narazil, nejsou nejspíš dány dobou vývoje (knihovna vzniká od roku 2006), ale spíše ustrnutím vývoje z důvodu, že knihovna není příliš využívaná. Je to škoda, protože alternativ je málo.

4.3 Knihovna pro implementaci uživatelského rozhraní

Uživatelské rozhraní lze implementovat třemi způsoby:

Pevně ve zdrojovém kódu: pomocí znalosti knihovny *gtkmm* vytvářet instance tříd (hlavního okna, aplikační nabídky, dialogů, tlačítek, atd.) a nastavovat jejich atributy. To je nevýhodné především v případech značné rozsáhlosti vlastního uživatelského rozhraní a vede k dlouhým tělům funkcí či metod, ve kterých se těžko orientuje.

Generováním zdrojového kódu: uživatelské rozhraní se definuje v aplikaci, která slouží k návrhu a pomocí ní se vygeneruje zdrojový kód, který po spuštění vytvoří uživatelské rozhraní přesně tak, jak bylo navrženo. Problematické je udržování vygenerovaných zdrojových kódů při potřebě rozšíření nebo změny návrhu.

Vytvářením uživatelského rozhraní za běhu: pomocí speciální knihovny je za běhu aplikace načten soubor vytvořený aplikací pro návrh, který obsahuje definici uživatelského rozhraní a podle něj se uživatelské rozhraní vytvoří. Tento způsob je nejméně efektivní z hlediska nároků na zdroje, ale poskytuje nejsnadnější způsob, jak uživatelské rozhraní během vývoje aplikace libovolně rozšiřovat a měnit bez nutnosti zásahu do zdrojových kódů aplikace. Rovněž je usnadněn překlad uživatelského rozhraní a popis je otevřen i pro samotného uživatele aplikace, který není spokojen například s rozmístěním jednotlivých objektů.

Pro implementaci byl vybrán třetí způsob vytváření uživatelského rozhraní. Knihovna, která z popisu v jazyce XML dovede vytvořit hlavní okno aplikace a ostatní formuláře pomocí prvků uživatelského rozhraní GTK, se nazývá *libglade* (resp. jejího wrapperu *libglademm*, který je součástí *gtkmm*). Aplikace pro návrh grafického uživatelského rozhraní pak *Glade*. Soubor s definicí rozhraní neobsahuje pouze hlavní okno aplikace a jeho obsah (aplikační nabídka, lišta nástrojů, stromový seznam obsahující modely z knihovny a místo pro pracovní plátno), ale i kontextové nabídky určené pro objekty na plátně či formuláře pro specifikaci parametrů atomických DEVS modelů a portů.

4.4 Čtení a zápis DEVS modelů pomocí knihovny

Použitím knihovny *libxml++* [14] se provádí čtení vstupních souborů v jazyce XML, které popisují DEVS modely. Od základního SAX parseru je odvozena vlastní třída, která implementuje metody při výskytu různých částí XML dokumentu využitím návrhového vzoru *stav* (viz 3.4.1).

K zápisu se rovněž využívá knihovna *libxml++*. Třída `Element` obsahuje metody pro specifikaci atributů a jejich hodnot, obsahu elementu a umožňuje přidání nových elementů – potomků. Objekty typu `AtomicDEVS` a `CoupledDEVS` tyto metody využívají. Výsledný kořenový objekt `Element` je možné uložit do souboru.

4.5 Moduly pro export modelu

Moduly určené pro export modelu je možné pevně zakomponovat do programu tak, že jejich překlad proběhne při překladu samotné aplikace. Druhou možností je využití knihovny, která umožňuje načítat moduly za běhu. V prostředí operačních systémů splňujících standard POSIX (například Linux) k tomu slouží knihovna s názvem *libdl*. Ta umožňuje načtení libovolné sdílené knihovny a obsahuje funkce, které dovedou zavolat její funkci specifikovanou názvem.

Připravený modul se načte pomocí funkce `dlopen` s parametry název souboru a způsob načtení. Pokud neobsahuje lomítko, jsou knihovny hledány na obvyklém místě (adresáře `/lib/`, `/usr/lib/` nebo další lokace určené obsahem proměnné prostředí `LD_LIBRARY_PATH`). Způsob načtení určuje, zda budou závislosti na symbolech vyřešeny (*resolved symbols*) až dojde k volání kódu, který je vyžaduje, nebo ihned při volání funkce. Pokud je knihovna nalezena a otevřena, vrací funkce ukazatel, v opačném případě *NULL*. Ukazatel slouží ostatním funkcím pro přístup k interní struktuře reprezentující načtený modul. V případě úspěchu je možné volat funkce knihovny pomocí `dlsym()` s parametrem určujícím název požadované funkce. Návratová hodnota obsahuje ukazatel na funkci, kterou lze pak zavolat. Je proto nutné znát předem počet parametrů modulární funkce, jejich typ a návratovou hodnotu. Pro ukončení práce s modulem je volána funkce `dlclose()`, při výskytu chyby lze použít funkci `dlerror()`, která vrací řetězec obsahující popis vzniklé chyby.

Následuje příklad prázdného modulu určeného pro export:

```
#include "../src/module.hh"

extern "C" module_info
get_module_info() {
   module_info info;
   info.name = "Template export...";
   info.type = MODULE_EXPORT_TO_FILE;
   return info;
}

extern "C" void
export_func(DEVS* devs, Glib::ustring output_filename)
{
   // funkce, která model DEVS převede na požadovanou reprezentaci
}
```

Pro překladač jazyka C++ je nutné zdrojový kód doplnit o direktivu `extern "C"`, která slouží při překladu k označení funkce jazyka C [8]. Odpovídající *Makefile.am* určený pro nástroje *autoconf* a *automake* vypadá následovně:

```
file_LTLIBRARIES = module.la

module_la_SOURCES = module.cc

AM_CXXFLAGS = @GTKMM_CFLAGS@ @LIBXMLPP_CFLAGS@
AM_LDFLAGS = -avoid-version -module
```

Tím se vytvoří sdílená knihovna, kterou je možné pomocí knihovny *libdl* načíst. Parametr `avoid-version` zakáže vytvářet verzované moduly a parametr `module` vytvoří knihovnu, kterou lze otevřít pomocí `dlopen()`.

Pro potřebu demonstrace exportu modelu čistě pomocí jazyka C++ a knihovny *libdl* byl vytvořen exportní modul pro simulační knihovnu *adevs* [10] Simulační knihovna *adevs (A Discrete EVent System simulator)* je napsána v jazyce C++ a slouží ke konstrukci diskrétních simulátorů založených na bázi paralelního a dynamického DEVS formalismu.

4.6 Transformace modelu pomocí XSLT

V případě použití XSLT pro převod modelu uloženého v XML do některého simulačního nástroje lze využít některou z dostupných knihoven. Například *libxslt* je napsaná v jazyce C a implementuje transformaci XML dokumentů podle šablon [15]. Vychází z knihovny *libxml2*, která slouží k parsování XML dokumentů (wrapperem *libxml2* do jazyka C++ je dříve zmíněná *libxml++*). Pomocí *libxslt* lze model jednoduše transformovat následujícím kódem:

```
cur = xsltParseStylesheetFile(stylesheet_filename);
doc = xmlParseFile(source_filename);
res = xsltApplyStylesheet(cur, doc, params);
xsltSaveResultToFile(destination_filename, res, cur);
```

Vetším problémem může být vytvoření správné XSLT šablony. Specifikace jazyka XSLT je totiž poměrně obsáhlá, proto jeho použití klade větší nároky na znalosti tvůrce modulu. Aplikace obsahuje příklad řešení exportu pomocí XSLT šablony, která řeší export do simulačního nástroje navrženého v 3.6. Obsah šablony je uveřejněn v příloze.

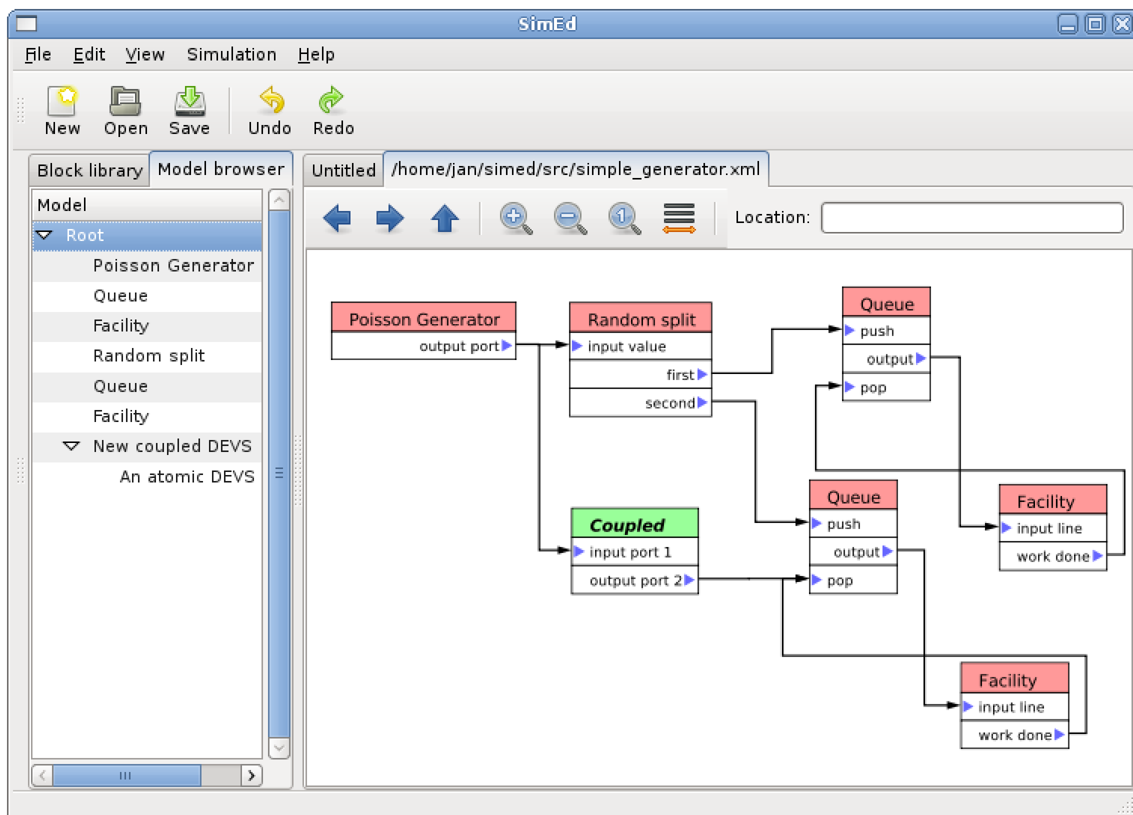
4.7 Vzhled uživatelského rozhraní

Při návrhu vzhledu uživatelského rozhraní byl kladen důraz na podobnost s existujícími grafickými editory blokových schémat. Na obrázku 4.1 se nachází snímek okna hlavního uživatelského rozhraní.

Vlevo je umístěna knihovna modelů spolu s náhledem na strukturu modelu (*model browser*). Nahoře je hlavní nabídka aplikace, která obsahuje práci se soubory (zakládání, otevírání, ukládání, zavírání), export modelu do některé simulační knihovny, editační funkce (funkce dopředu, zpět, výřez, kopírování, vkládání, mazání, výběr všeho a nastavení aplikace). Menu zobrazení slouží ke změně měřítka pracovního plátna.

Největší plochu zabírají záložky s otevřenými modely. Ty jsou zobrazeny na pracovním plátně pomocí stavebních bloků používaných pro vytváření DEVS modelů (atomický a složený DEVS model, porty, spoje).

Dialog pro nastavení parametrů a chování atomického DEVS modelu (na obrázku 4.2) obsahuje název a popis modelu, seznam stavových proměnných, které lze libovolně přidávat, mazat a měnit pomocí tlačítek a dvojitého kliknutí na jednotlivé položky v seznamu. Na dalších záložkách (*Behaviour*) se nachází vstupní textová pole pro určení chování atomického DEVS modelu pomocí vnitřní a vnější přechodové funkce – $\delta_{int}, \delta_{ext}$, funkce posuvu času ta a výstupní funkce λ. Pro rozšířené formalismy (například paralelní DEVS) lze definovat rozlišovací funkci δ_{int} a inicializační funkci.



Obrázek 4.1: Uživatelské rozhraní implementované aplikace

4.8 Příklad řešení problému

Pro demonstraci základních vlastností aplikace byl vybrán jednoduchý model systému hromadné obsluhy, který řeší simulaci a optimalizaci počtu zařízení pro obsluhu událostí vzniklých příchodem výrobků do systému. K simulaci byla použita knihovna *adevs*.

4.8.1 Generátor

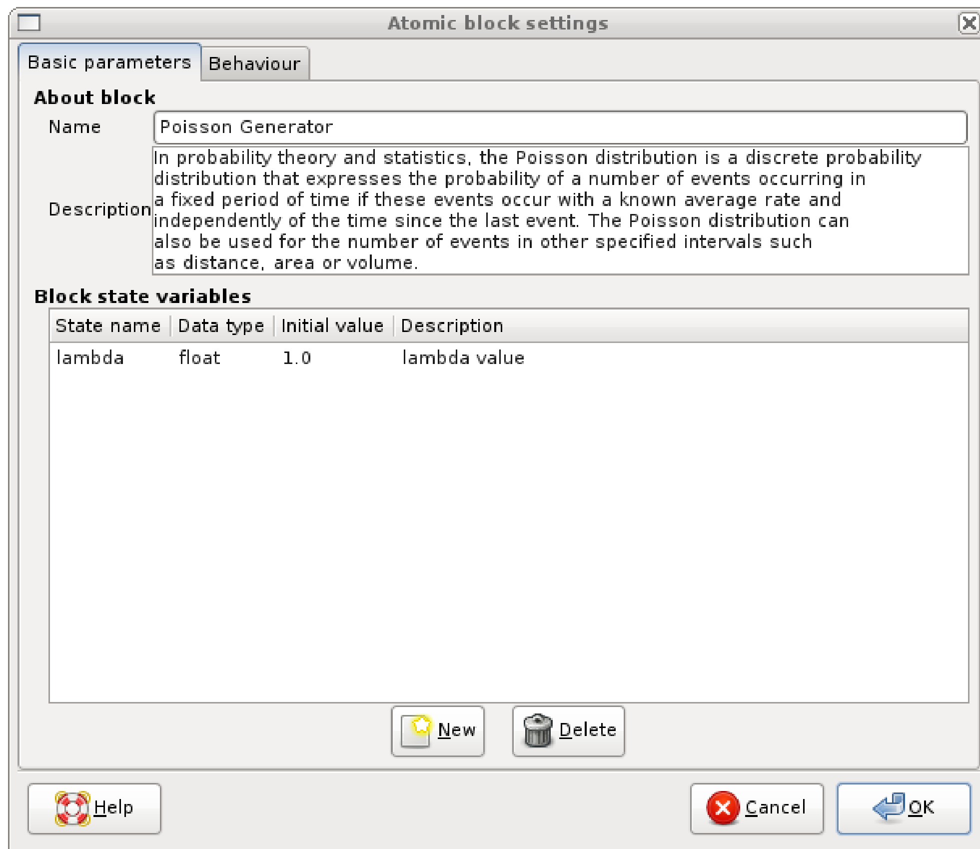
Vstupní tok požadavků je určen Poissonovým rozložením, které definuje počet příchodů za jednotku času. Vytvoříme atomický model reprezentující generátor, pojmenovaný *Poisson Generator* a definujeme mu stavovou proměnnou *lambda*, která určuje střední hodnotu výskytu jevu, a generátor náhodných čísel *random_gen* z knihovny *adevs*. Dále určíme funkci posuvu času – *Time advance function*

```
return 1.0;
```

a výstupní funkci – *Output function*

```
double incoming_num = random_gen.poisson(lambda);
adevs::PortValue<double> output(1, double(incoming_num));
yb.insert(output);
```

Výstupní port je s pořadovým číslem 1. Nový atomický DEVS má totiž standardně dva porty, vstupní (s číslem portu 0) a výstupní (s číslem portu 1).



Obrázek 4.2: Dialog pro nastavení parametrů a chování atomického bloku

4.8.2 Zařízení s frontou

Složený DEVS určený pro vlastní zařízení bude obsahovat dva atomické modely. První s názvem *Queue* bude implementovat funkci fronty. Bude vybaven čtyřmi porty (vstup do fronty, délku fronty, požadavek o výrobek a výdej výrobku). Obsahuje čtyři stavové proměnné (počet výrobků ve frontě, příznak volného zařízení, příznak požadavku o zaslání dat a délky fronty). Interní přechodovou funkci – *Internal transition function* implementuje kód:

```
send_length = false;
if (items_count > 0 && facility_is_free) {
  items_count--;
  facility_is_free = false;
  send_length = true;
}
```

V případě, že fronta výrobků není prázdná a zařízení je volné, výrobek opouští frontu. Externí přechodová funkce určuje chování modelu při výskytu vnější události.

```
Bag<PortValue<double> >::const_iterator iter = xb.begin();
for (; iter != xb.end(); iter++) {
  if ((*iter).port == 0 && (*iter).value > 0) { // input coming
```

```
      items_count += (*iter).value;
      send_length = true;
    } else if ((*iter).port == 3) { // request for data from receiver
      facility_is_free = true;
    }
  }
```

Pokud dojde k příchodu nového výrobku do fronty, je do ní zařazen a je nastaven příznak požadavku o zaslání délky fronty. Při výskytu zprávy informující o volném zařízení je nastaven příslušný příznak. Funkce posuvu času je určena stavem, ve kterém se fronta nachází.

```
if (send_length || (facility_is_free && items_count > 1)) // length sending
  return 0;
return 10e100;  // nekonečno
```

K okamžitému vyřízení dojde v případě existence požadavku o zaslání délky fronty nebo je-li zařízení uvolněné a zároveň fronta není prázdná. Výstupní funkce vytváří výstupní události na základě obsahu stavových proměnných.

```
if (facility_is_free && items_count > 0) {
  PortValue<double> val(1, 1);
  yb.insert(val);
}
if (send_length) {
  PortValue<double> val(2, items_count);
  yb.insert(val);
}
```

Pokud je cílové zařízení volné a existují výrobky ve frontě, je vytvořena zpráva informující o přesunu výrobku do zařízení. Rovněž je realizována výstupní událost při vzniku požadavku na zaslání délky fronty.

Druhým atomickým blokem složeného atomického modelu je zařízení – *facility*. Obsahuje dvě stavové proměnné: příznak zda zařízení obsluhuje výrobek a příznak platný pouze při startu simulace. K interní přechodové funkci

```
is_working = false;
start_state = false;
```

dojde pouze v případě obsazení zařízení a při výskytu obsluhy prvního výrobku. Externí přechodová funkce

```
Bag<adevs::PortValue<double> >::const_iterator iter = xb.begin();
for (; iter != xb.end(); iter++) {
  if ((*iter).port == 0) { // input
      is_working = true;
  }
}
```

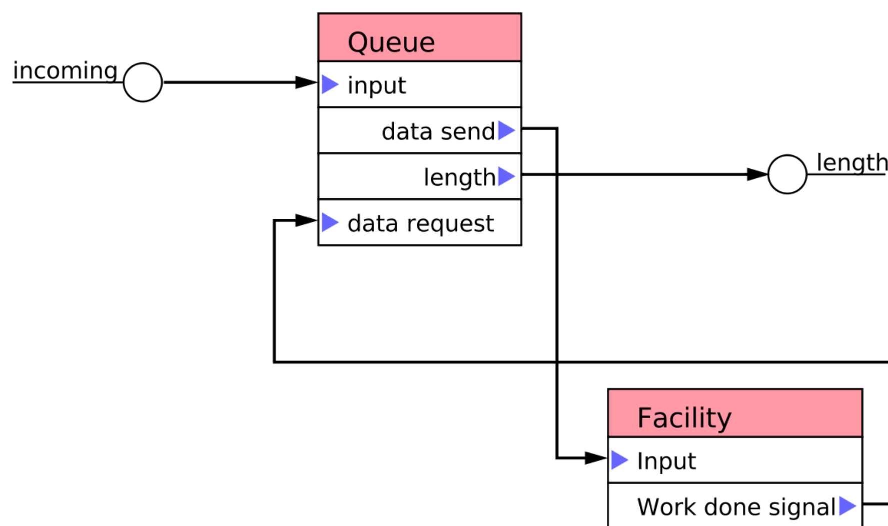
ošetřuje vnější události, které vzniknou požadavkem o zpracování výrobku. Funkce posuvu času

```
if (is_working) {
  return random_gen.normal(5.0, 0.3);
} else if (start_state) {
  return 0;
}
return 10e100;
```

zajišťuje dobu obsluhy pomocí normálního rozložení se střední hodnotou $\mu = 5.0$ a rozptylem $\sigma^2 = 0.3$. Výstupní funkce

```
PortValue<double> val(1, 1);
yb.insert(val);
```

podává zprávu o dokončení práce na výrobku. Výsledný složený model propojíme podle obrázku 4.3.



Obrázek 4.3: Model reprezentující zařízení s frontou

4.8.3 Monitorovací blok

Pro potřebu zápisu délky fronty zařízení bude model obsahovat blok určený pro zápis hodnoty, rozšířené o čas výskytu její změny, do souboru. Aby bylo možné zapisovat několik hodnot, rozšíříme počet vstupních portů. Blok nebude mít žádné výstupní porty. Vnitřní přechodová funkce

```
t += ta();
write_output = false;
```

zakazuje vytváření výstupu do příchodu další vnější události. Vnější přechodová funkce

```
t += e;
Bag<PortValue<double> >::const_iterator iter = xb.begin();
for (; iter != xb.end(); iter++) {
```

```
    if ((*iter).port == 0) {
      v1 = (*iter).value;
    } else if ((*iter).port == 1) {
      v2 = (*iter).value;
    } else if ((*iter).port == 2) {
      v3 = (*iter).value;
    } else if ((*iter).port == 3) {
      v4 = (*iter).value;
    } else if ((*iter).port == 4) {
      v5 = (*iter).value;
    }
  }
  write_output = true;
```

nastaví jednotlivé hodnoty a nastaví příznak zápisu do souboru. Funkce posuvu času

```
  if (write_output)
    return 0;
  else
    return 10e100;
```

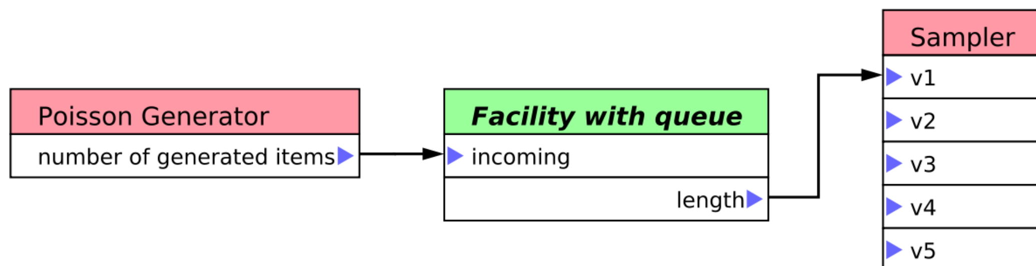
při požadavku o zápis spustí okamžitě výstupní funkci

```
  fprintf(fp, "%f %f %f %f %f %f\n", t, v1, v2, v3, v4, v5);
```

která realizuje zápis do souboru.

4.8.4 Propojení bloků a simulace

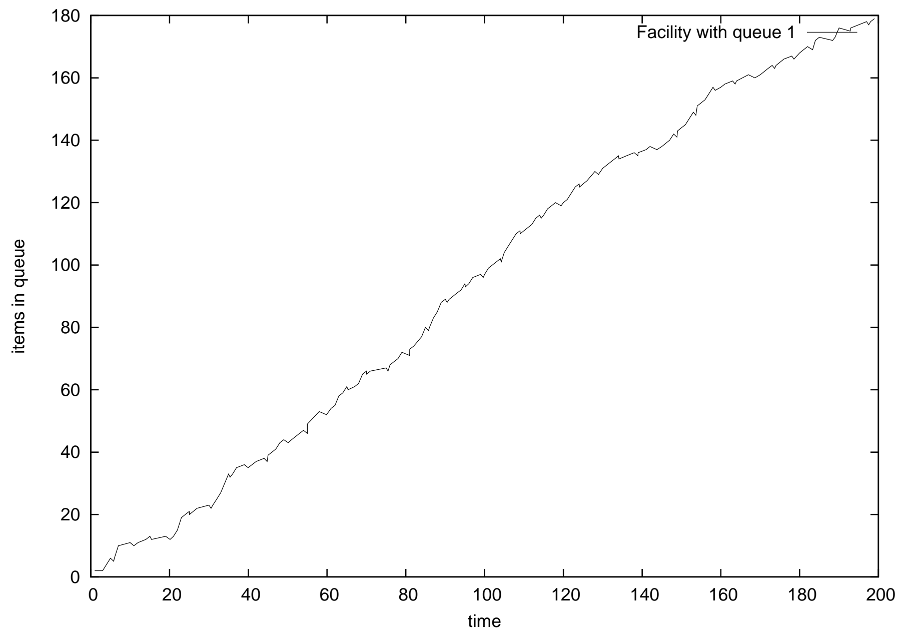
Propojíme bloky tak, aby výstup generátoru byl připojen ke vstupu zařízení a port určený pro délku fronty byl připojen k monitorovacímu bloku (viz obrázek 4.4). Poté je možné



Obrázek 4.4: Model reprezentující zařízení s frontou

vygenerovat zdrojový kód určený pro knihovnu *adevs* pomocí nabídky *File/Export/ADEVS export...*. Po zadání názvu souboru a potvrzení lze model přeložit pomocí

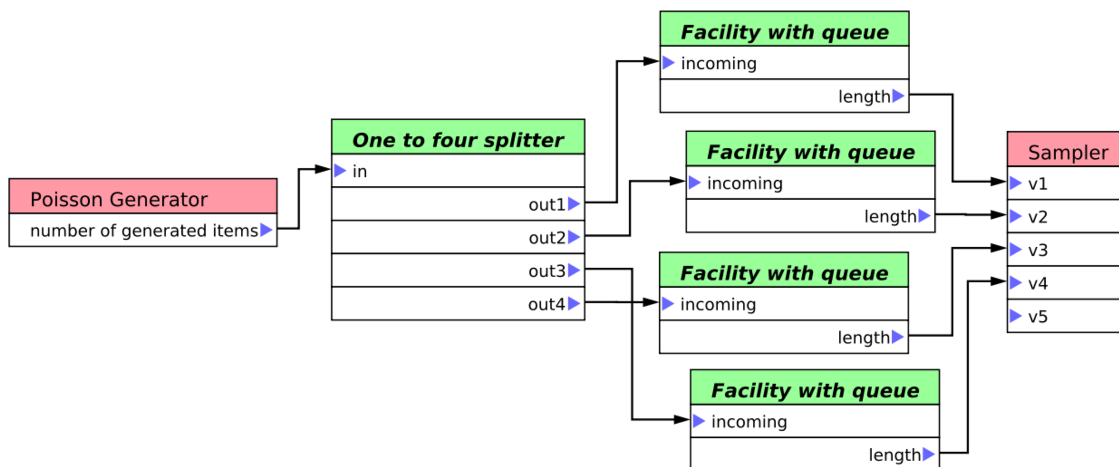
```
  g++ [název_souboru].cc -I/[cesta k adevs]/adevs-2.1/include/ \
      -ladevs -L[cesta k adevs]/adevs-2.1/src/
```



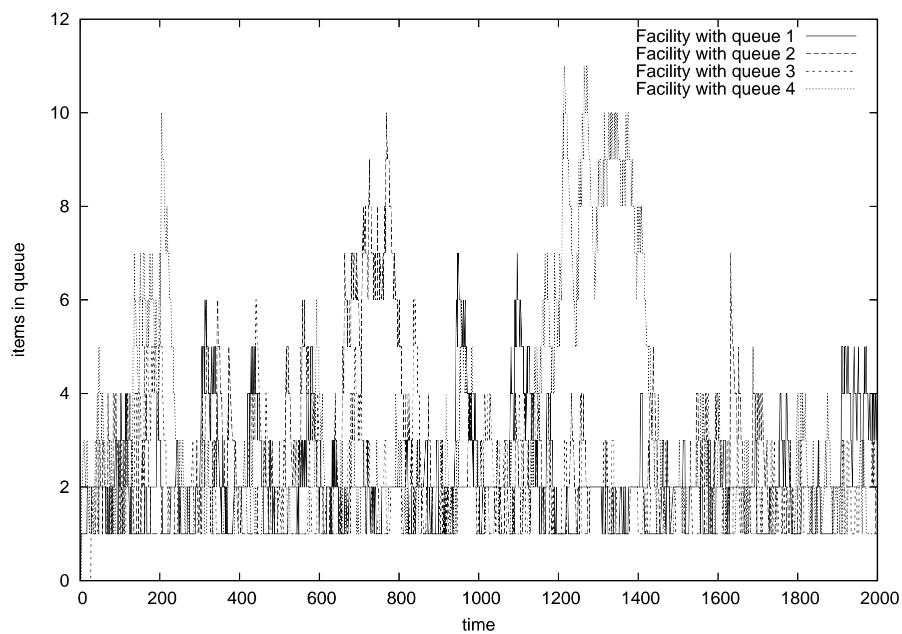
Obrázek 4.5: Výsledky simulace s jedním zařízením

4.8.5 Výsledky

Z výsledků simulace můžeme zjistit, že fronta se stále prodlužuje a jedno zařízení je tak na obsluhu nedostačující (viz obrázek 4.5). Model rozšíříme o takový počet zařízení, aby fronty jednotlivých zařízení nerostly donekonečna. K tomu abychom mohli generované výrobky přiřazovat náhodným zařízením vytvoříme blok dělič – *Splitter*. Ten přepošle vstupní hodnotu na jeden nebo na druhý výstupní port. Zapojením těchto bloků do stromu můžeme vytvořit dělič, který rovnoměrně rozděluje vstupní hodnoty do 2^n výstupních portů a případně tyto bloky vložit do jednoho složeného modelu. Optimalizovaný model je na obrázku 4.6, graf znázorňující závislost času na délce front jednotlivých zařízení je na obrázku 4.7.



Obrázek 4.6: Optimalizovaný model



Obrázek 4.7: Výsledky simulace se čtyřmi zařízeními

Kapitola 5

Závěr

Výsledkem této práce je aplikace určená pro vytváření DEVS modelů pomocí grafického editoru. K lepšímu pochopení problému bylo třeba nejdříve nastudovat DEVS formalismus, jehož základní přehled, včetně definice, příkladů a simulace, práce rovněž obsahuje. Při návrhu a implementaci byl kladen důraz na prezentaci modelu uživateli tak, aby byla jasná a přehledná. K tomu je však potřeba značná pečlivost při konstrukci modelů. Význam bloků a portů by měl být díky vhodným názvům a popisům zřejmý hned na první pohled. Vytvořený model lze uložit do XML dokumentu a pomocí XSLT šablony nebo vlastního modulu exportovat do cílového simulačního jazyka či knihovny. Vytvářet vlastní moduly je poměrně jednoduché a nevyžaduje kompilaci celé aplikace. Oproti šablonám je jejich konstrukce jednodušší, jazyk XSLT je totiž poměrně komplikovaný. Kromě toho práce obsahuje popis známých aplikací, které se vytvářením modelů pomocí grafických editorů simulačních modelů zabývají.

Mně práce posloužila především pro hlubší seznámení s problematikou modelování a simulace pomocí DEVS formalismu. Pochopil jsem význam návrhových vzorů a jejich použití pro různé problémy, se kterými jsem se při návrhu setkal. V neposlední řadě jsem se seznámil hlouběji s jazykem XML, jeho validací a se šablonami XSLT.

Grafickým editorům modelů je však nutné vytknout poměrně velkou svázanost oproti silnějším vyjadřovacím schopnostem, které nám nabízí programovací jazyky. Toto omezení lze snížit implementací komplexních exportních modulů. Pro vzájemnou kompatibilitu mezi modely určenými pro různé simulační knihovny by bylo vhodné zavést vlastní simulační metajazyk. Ten by se pomocí modulů převáděl do zdrojového kódu určeného pro jednotlivé simulační knihovny. Tento problém je však nad rámec zadání této diplomové práce a jeho řešení není obecné ani jednoduché. Kromě toho by bylo možné aplikaci rozšířit navrženým rozhraním pro řízení simulace pomocí modulů, tak aby bylo možné přeložit, spustit a řídit simulaci rovnou z grafického editoru simulačních modelů a případně zvážit možnosti interpretace výsledků.

Literatura

[1] Francois E. Cellier. *Continuous System Simulation*. Springer, 2006. ISBN 978-0387261027.

[2] World Wide Web Consortium. Extensible markup language (xml). Dokument dostupný na http://www.w3.org/XML/, 2008. (7. 5. 2008).

[3] World Wide Web Consortium. The extensible stylesheet language family (xsl). Dokument dostupný na http://www.w3.org/Style/XSL/, 2008. (7. 5. 2008).

[4] World Wide Web Consortium. Xml path language (xpath) 2.0. Dokument dostupný na http://www.w3.org/TR/xpath20/, 2008. (7. 5. 2008).

[5] Aaron Crane. Does xml suck? Dokument dostupný na http://xmlsucks.org/but_you_have_to_use_it_anyway/does-xml-suck.html, 2002. (7. 5. 2008)].

[6] Paul A. Fishwick. *Simulation Model Design and Execution*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1995. ISBN 0-130-98609-7.

[7] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. ISBN 978-0201633610.

[8] Aaron Isotton. C++ dlopen mini howto. Dokument dostupný na http://www.isotton.com/devel/docs/C++-dlopen-mini-HOWTO/, 2006. (7. 5. 2008).

[9] Hořák Jan. *Editor simulačních modelů*. Vysoké učení technické, Fakulta informačních technologií, 2005. [Bakalářská práce].

[10] Jim Nutaro. Adevs: A discrete event system simulator. Dokument dostupný na http://www.ece.arizona.edu/~nutaro/, 2008. (7. 5. 2008).

[11] WWW stránky a dokumentace. Gtk+ - documentation. Dokument dostupný na http://www.gtk.org/documentation.html. (7. 5. 2008).

[12] WWW stránky a dokumentace. Gtkmm – wrapper gtk+ do c++. Dokument dostupný na http://www.gtkmm.org. (7. 5. 2008).

[13] WWW stránky a dokumentace. Papyrus library programmer's guide. Dokument dostupný na http://libpapyrus.sourceforge.net/reference/html/index.html.

[14] WWW stránky a dokumentace. libxml++ documentation. Dokument dostupný na http://libxmlplusplus.sourceforge.net/docs/reference/html/index.html, 2008. (7. 5. 2008).

[15] Daniel Veillard. The xslt c library for gnome. Dokument dostupný na http://xmlsoft.org/XSLT/, 2008. (7. 5. 2008).

[16] Wikipedia. Graphical user interface — wikipedia, the free encyclopedia. Dokument dostupný na http://en.wikipedia.org/w/index.php?title=Graphical_user_interface&oldid=210746646. (7. 5. 2008).

[17] Wikipedia. Xml — wikipedia, the free encyclopedia. Dokument dostupný na http://en.wikipedia.org/w/index.php?title=XML&oldid=209408891, 2008. (7. 5. 2008).

[18] Bernard Zeigler. *Theory of Modeling and Simulation.* John Wiley & Sons, 1976. ISBN 978-0127784557.

[19] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation.* Academic Press, 2000. ISBN 0-12-778455-1.

Přílohy

XSL Schema uloženého modelu

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="TODO"
xmlns="modelformat"
elementFormDefault="qualified">

<xs:simpleType name="descriptionType">
   <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
   </xs:restriction>
</xs:simpleType>

<xs:element name="model">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="description" type="descriptionType" default="" />
         <xs:element name="block">
            <xs:complexType>
               <xs:attribute name="name" type="xs:decimal" />
               <xs:attribute name="type" type="blockType" />
               <xs:attribute name="id" type="xs:integral" />
               <xs:attribute name="x" type="xs:decimal" />
               <xs:attribute name="y" type="xs:decimal" />
               <xs:sequence>
                  <xs:element name="description" type="descriptionType" default="" />
                  <xs:element name="port" type="portType" minOccurs="0" maxOccurs="unbounded" />
                  <xs:element ref="block" minOccurs="0" maxOccurs="unbounded" />
                  <xs:element name="link" type="linkType" maxOccurs="unbounded" minOccurs="0" />
                  <xs:element name="behaviour" maxOccurs="1" minOccurs="0">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="stateVar" type="stateVarType" default=""
                                       minOccurs="0" maxOccurs="unbounded" />
                           <xs:element name="deltaInt" type="xs:string" default="" />
                           <xs:element name="deltaExt" type="xs:string" default="" />
                          <xs:element name="ta" type="xs:string" default="" />
                           <xs:element name="lambda" type="xs:string" default="" />
                           <xs:element name="deltaCon" type="xs:string" default="" />
                           <xs:element name="initFunc" type="xs:string" default="" />
                         </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:simpleType name="porttypeType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="input" />
```

```
      <xs:enumeration value="output" />
   </xs:restriction>
</xs:simpleType>

<xs:simpleType name="blockType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="atomic" />
      <xs:enumeration value="coupled" />
   </xs:restriction>
</xs:simpleType>


<xs:complexType name="portType">
   <xs:attribute name="name" type="xs:string"   use="required" />
   <xs:attribute name="pos"  type="xs:integral" use="required" />
   <xs:attribute name="type" type="porttypeType"    use="required" />
   <xs:attribute name="desc" type="xs:string"   use="required" />
</xs:complexType>

<xs:complexType name="linkType">
   <xs:attribute name="fromId" type="xs:integral" />
   <xs:attribute name="fromPortNum" type="xs:integral" />
   <xs:attribute name="toId" type="xs:integral" />
   <xs:attribute name="toPortNum" type="xs:integral" />
</xs:complexType>

<xs:complexType name="stateVarType">
   <xs:attribute name="name" type="xs:string" />
   <xs:attribute name="initialValue" type="xs:string" />
   <xs:attribute name="type" type="xs:string" />
   <xs:attribute name="desc" type="xs:string" />
</xs:complexType>
</xs:schema>
```

XSLT šablona pro převod modelu do zdrojového kódu

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/model">
// This file is generated from simed
#include "Coordinator.hh"
#include "Simulator.hh"
<xsl:text disable-output-escaping="yes">#include &lt;math.h&gt;</xsl:text>

     <xsl:apply-templates select="block[*]" />
int main() {
   Block_<xsl:value-of select="generate-id(block)"/>
      *Block_<xsl:value-of select="generate-id(block)"/>_var
      = new Block_<xsl:value-of select="generate-id(block)"/>(NULL);
   RootCoordinator r(Block_<xsl:value-of select="generate-id(block)"/>_var, 0.0, 550.0);
   r.run();
}
</xsl:template>
<xsl:template match="block">
<!--<xsl:for-each select="block">
class Block_<xsl:value-of select="generate-id(.)" />;
</xsl:for-each> -->
<xsl:apply-templates select="./block[*]" />
class Block_<xsl:value-of select="generate-id(.)"/>
        <xsl:if test="@type = 'coupled'">: public Coordinator {
   public:
      Block_<xsl:value-of select="generate-id(.)"/>(Coordinator *c): Coordinator(c)  {}
   private:
      void init_func() {
         <xsl:for-each select="block">
         Block_<xsl:value-of select="generate-id(.)"/>
```

```
            *Block_<xsl:value-of select="generate-id(.)"/>_var
            = new Block_<xsl:value-of select="generate-id(.)"/>(this);
         AddDEVS(Block_<xsl:value-of select="generate-id(.)"/>_var);</xsl:for-each>
         <xsl:for-each select="link">
         Connect(
             <xsl:if test = "@fromId = ../@id">this, </xsl:if>
             <xsl:variable name="from_id_var" select="@fromId" />
             <xsl:for-each select="../block">
                  <xsl:if test = "$from_id_var = @id">
                       Block_<xsl:value-of select="generate-id(.)"/>_var,
                  </xsl:if>
             </xsl:for-each>
             <xsl:value-of select="@fromPortNum"/>,
             <xsl:if test = "@toId = ../@id">this, </xsl:if>
             <xsl:variable name="to_id_var" select="@toId" />
             <xsl:for-each select="../block">
                  <xsl:if test = "$to_id_var = @id">
                       Block_<xsl:value-of select="generate-id(.)"/>_var,
                  </xsl:if>
             </xsl:for-each>
             <xsl:value-of select="@toPortNum"/>);
         </xsl:for-each>
      }
};
        </xsl:if>
        <xsl:if test="@type = 'atomic'">: public Simulator {
   public:
      Block_<xsl:value-of select="generate-id(.)"/>(Coordinator *c): Simulator(c)  {}
   private:
      <xsl:for-each select="behaviour/stateVar">
      <xsl:value-of select="@type" /><xsl:text> </xsl:text> <xsl:value-of select="@name" />;
      </xsl:for-each>
      virtual void init_func() {
         <xsl:for-each select="behaviour/stateVar">
         <xsl:value-of select="@name" /><xsl:text>
              = </xsl:text> <xsl:value-of select="@initialValue" />;
         </xsl:for-each>
      }
      virtual void delta_int(double t) {
         <xsl:value-of disable-output-escaping="yes" select="behaviour/deltaInt" />
      }
      virtual void delta_ext(double t, double e, PortEvent x) {
         <xsl:value-of disable-output-escaping="yes" select="behaviour/deltaExt" />
      }
      virtual double ta() {
         <xsl:value-of disable-output-escaping="yes" select="behaviour/ta" />
      }
      virtual PortEventList lambda(double t) {
         <xsl:value-of disable-output-escaping="yes" select="behaviour/lambda" />
      }
};
        </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Překlad zdrojových kódů

Instrukce pro překlad zdrojových kódů se nachází na přiloženém CD v souboru `README`.