VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

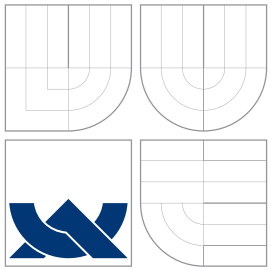
PLATFORMNĚ NEZÁVISLÁ REPREZENTACE SIMULAČNÍCH MODELŮ NA BÁZI XML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

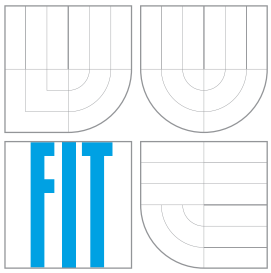
AUTOR PRÁCE
AUTHOR

DAVID DURMAN

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLATFORMNĚ NEZÁVISLÁ REPREZENTACE SIMULAČNÍCH MODELŮ NA BÁZI XML

PLATFORM NEUTRAL REPRESENTATION OF SIMULATION MODELS BASED ON XML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID DURMAN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VLADIMÍR JANOUŠEK, Ph.D.

BRNO 2009

Abstrakt

S rozmanitostí platforem pro popis modelů systémů s diskrétními událostmi vyvstává problém, jak tyto modely reprezentovat tak, aby bylo možné je napříč platformami sdílet, ať už pro účely validace těchto modelů, jejich efektivnějšího spouštění, nebo pro účely znovupoužití či začlenění modelů do větších celků. Tato práce rozšiřuje a implementuje jazyk DEVSML pro reprezentaci DEVS modelů, u nichž pro popis chování atomických komponent využívá stavových diagramů, které reprezentuje pomocí SCXML. Akce a stráže stavových diagramů popisuje jazykem Scheme. Dále byl vytvořen vizuální nástroj pro tvorbu stavových diagramů, který je začleněn do systému SmallDEVS a také transformátor modelů popsaných pomocí DEVSML do prostředí Adevs.

Abstract

The variety of platforms for describing models of discrete event systems brings about the problem of how these models should be represented in order to be easily shared across platforms (shared for the purposes of model validation, effective execution, model reusing, or model integration). This study develops and implements the language DEVSML for the representation of the DEVS models. For the description of these models components, I use state diagrams, which are represented by SCXML. Actions and guards of the state diagrams are described using the language Scheme. This study also presents a visual tool for creation of the state diagrams, which is incorporated in the SmallDEVS system. For the Adevs environment, a transformer of the models described by DEVSML is implemented.

Klíčová slova

DEVS, State diagrams, DEVSML, Smalltalk, SmallDEVS, SCXML, Adevs, Scheme

Keywords

DEVS, State diagrams, DEVSML, Smalltalk, SmallDEVS, SCXML, Adevs, Scheme

Citace

David Durman: Platformně nezávislá reprezentace simulačních modelů na bázi XML, diplomová práce, Brno, FIT VUT v Brně, 2009

Platformně nezávislá reprezentace simulačních modelů na bázi XML

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Janouška.

.....
David Durman
25. května 2009

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Ing. Vladimíru Janouškovi Ph.D. za poskytnutí cenných rad.

Obsah

1 Úvod **5**
- 1.1 Motivace . . . 5
- 1.2 Struktura práce . . . 6

2 Základní koncepty **7**
- 2.1 Entity v modelování a simulaci . . . 7
- 2.2 Typy systémů a rozdělení forem jejich specifikace . . . 8

3 Discrete event system specification (DEVS) **10**
- 3.1 Atomický DEVS . . . 10
- 3.2 Spojovaný DEVS . . . 11
- 3.3 DEVS simulátor . . . 12

4 StateChart **16**
- 4.1 Notace a sémantika . . . 17
- 4.2 Ekvivalence StateChartů a atomických DEVS komponent . . . 19
- 4.3 SCXML . . . 20
- 4.4 Rozšíření SCXML pro potřeby DEVSML . . . 21

5 Nástroj pro tvorbu stavových diagramů **23**
- 5.1 Architektura . . . 23
- 5.2 Použití . . . 24
- 5.3 Popis akcí/stráží pomocí jazyka Smalltalk. . . . 26
- 5.4 Popis akcí/stráží pomocí jazyka Scheme. . . . 26
- 5.5 Pořadí provádění akcí stavového automatu . . . 27
- 5.6 Skriptové vytváření stavového diagramu . . . 29
- 5.7 Automatická změna stavového diagramu za běhu . . . 31

6 DEVSML **35**
- 6.1 Struktura modelu v DEVSML . . . 35
- 6.2 Popis chování atomických komponent v DEVSML . . . 38

7 Převod DEVSML do prostředí Adevs **40**
- 7.1 Adevs . . . 41
- 7.2 QHsm . . . 41
- 7.3 Guile . . . 42
- 7.4 Demonstrace převodu na modelu Alternating Bit Protocol (ABP) . . . 43

8 Implementace **47**
8.1 Implementace stavového automatu 47
8.1.1 Konstrukce pomocí vnořeného příkazu `switch` 47
8.1.2 Stavová tabulka 48
8.1.3 Objektově orientovaný návrhový vzor Stav 48
8.1.4 Meta vzor dědičnost chování 49
8.2 Implementace rozhraní mezi DEVS komponentou a stavovým automatem . 49

9 Závěr **52**
9.1 Dosažené cíle 52
9.2 Možná rozšíření 52

A Spojovaná komponenta systému Alternating Bit Protocol popsaná skriptem. **56**

B Snímek obrazovky probíhající simulace modelu Síťového přepínače **63**

C Obsah CD **65**

Seznam obrázků

2.1 Základní entity v modelování a simulaci. 8
2.2 Rozdělení formalismů pro specifikaci systémů. 9

3.1 Interpretace DEVS formalismu. 11
3.2 Korespondence hierarchického modelu s hierarchickým simulátorem. 13

4.1 Ukázka StateChartu modelujícího stopky. 18

5.1 Ukázka nástroje pro tvorbu stavových diagramů. 24
5.2 Ukázka inspektoru atomického DEVS modelu a v něm zabudovaného nástroje pro tvorbu stavových diagramů. 25
5.3 Architektura nástroje pro tvorbu stavových diagramů. 25
5.4 Stavový automat pro demonstraci provádění akcí. 28
5.5 Dynamický stavový diagram. 33
5.6 Atomická DEVS komponenta řízená dynamickým stavovým diagramem. . . 33
5.7 Automatická změna stavového diagramu za běhu simulace. 34

6.1 Generátor DEVSML. 36
6.2 Spojovaná DEVS komponenta síťového přepínače. 37
6.3 Stavový diagram atomické komponenty procesor. 38

7.1 Transformace jazyka DEVSML do prostředí Adevs. 40
7.2 Hierarchie tříd Adevs použitých v této práci. 41
7.3 Spojovaná DEVS komponenta systému Alternating Bit Protocol (ABP). . . 43
7.4 Atomické DEVS komponenty spojovaného modelu ABP: Sender, Receiver. . 45
7.5 Atomické DEVS komponenty spojovaného modelu ABP: Gen, Acceptor, Trans, Ack. 46

8.1 Návrhový vzor Stav. 48
8.2 Meta vzor dědičnost chování. 50
8.3 Implementace rozhraní mezi DEVS komponentou a stavovým automatem. . 51

B.1 Kompletní pohled na modelovací nástroj. 64

Seznam výpisů

4.1 Ukázka SCXML popisu StateChartu 4.1. 20
5.1 Změna interpretru. 25
5.2 Skriptové vytvoření stavového diagramu a příslušných slotů. 30
5.3 Vytvoření modelu atomické komponenty a rozhraní se stavovým diagramem. 31
5.4 Otevření inspektoru nad modelem. 31
5.5 Dynamické vytvoření stavu a přechodů. 32
6.1 DEVSML popis komponenty z obrázku 6.2. 35
6.2 Atomická komponenta procesor v DEVSML. 38
7.1 Stavová metoda v QHsm. 42

Kapitola 1

Úvod

1.1 Motivace

Matematická teorie systémů poskytuje prostředky pro reprezentaci a získávání znalostí o dynamických systémech. [23, 28, 13] Tyto prostředky, nebo lépe formalismy, popisují jak strukturu, tak i chování modelovaného systému. Často jsou tyto formalismy velice úzce spjaty s konkrétním simulačním jazykem. Nicméně, formalismy disponují nezávislou koncepční existencí [11]. Je tedy vhodné hledat takovou reprezentaci modelů, která by byla naprosto nezávislá na modelovacím prostředí. Nutno dodat, že nalezení nezávislé reprezentace není nikterak jednoduchá záležitost. Překážkou je popis chování systému, kde je na nejnižší úrovni použito konstrukcí, které lze chápat napříč různými programovacími paradigmaty rozdílně. Jako řešení se nabízí modelovat chování systému formalismem, který tyto konstrukce dovede na jemnější úroveň. V ideálním případě na takovou úroveň, kde jsou již všechny stavy systému formálně popsány a komunikace s okolím se, v případě nutnosti, děje pouze pomocí volání externích funkcí či služeb. Takto reprezentované modely lze označovat za platformně nezávislé (PIM - platform independent models), a lze je potom automatizovaně transformovat do platformně závislých modelů (PSM - platform specific models), které jsou spouštěny na konkrétní platformně.

Postup tvorby systémů, založený na předchozích myšlenkách, se označuje jako modelem řízená architektura (MDA - model driven architecture), strategická iniciativa oznámená začátkem roku 2002 organizací OMG (Object Management Group). Cílem je snížit dopady neustálých změn v oblasti technologií na životnost a kvalitu produktu. Při tomto přístupu představuje PIM model vyšší abstrakci modelovaného systému, která přežívá technologické změny. Technologické změny se tedy dotknou pouze PSM modelu, který je z PIM modelu automaticky syntetizován.

Formálně popsané platformně nezávislé modely s přesně definovanou sémantikou, v porovnání s např. UML modely[1], nedegradují pouze na dokumentační popis modelovaného systému, ale automatizovaný převod na platformně závislé modely z principu umožňují. Navíc je možné (a také žádoucí) tyto modely již v době návrhu prověřovat v simulačním prostředí.

Tato práce popisuje chování atomických komponent modelů s diskrétními událostmi pomocí stavových diagramů. Akce stavů a přechodů popisuje pomocí jazyka Scheme. Kompletní model potom převádí do platformně nezávislého jazyka DEVSML. Modely popsané jazykem DEVSML jsou automaticky převedeny na platformně závislé modely pro prostředí

[1] Výjimku tvoří StateCharty, které již jasně definovanou sémantiku vykazují.

Adevs.

1.2 Struktura práce

Práce začíná popisem formalismu pro specifikaci modelů s diskrétními událostmi (DEVS). Dále se zabývá vizuálním formalismem pro popis komplexních systémů StateCharts. V této kapitole je vysvětlena ekvivalence StateChartů a atomických DEVS komponent a představen jazyk SCXML a jeho rozšíření o časové a výstupní parametry stavů. Poté je uveden návrh vizuálního nástroje pro tvorbu stavových diagramů založených na konceptu StateChartů. Poslední sekce této kapitoly demonstruje možnost automatické změny stavového diagramu za běhu simulace, což je vlastnost, která je možná díky dynamickému prostředí jazyka Smalltalk. Další kapitola pojednává o jazyku DEVSML a jeho rozšíření o popis atomických komponent pomocí upravené verze jazyka SCXML. Poté je uvedena koncepce automatického převodu modelů z DEVSML do prostředí Adevs a vše je demonstrováno na příkladu. Práce je zakončena poznámkami k implementaci.

Kapitola 2

Základní koncepty

Počítačová simulace je disciplína, která nám umožňuje vytvářet modely skutečných, nebo teoretických systémů, tyto modely spouštět na počítači a zkoumat tak jejich výstupy. Díky tomu se můžeme o modelovaném systému dozvědět nové informace. Proces modelování sestává ze specifikace zdrojového systému. Znalosti o zdrojovém systému, které hodláme specifikovat, lze rozdělit do čtyř úrovní [19]:

úroveň 0 (zdrojová) Na této úrovni známe proměnné, které chceme měřit a jak je získat.

úroveň 1 (datová) Máme k dispozici data posbíraná měřením zdrojového systému.

úroveň 2 (generativní) Přechodem na tuto úroveň jsme schopni data získaná měřením zdrojového systému generovat použitím kompaktnější reprezentace zdrojového systému. Reprezentací je myšlen například soubor rovnic, kterými data získáme. Reprezentace pomocí rovnic již představuje novou znalost o zdrojovém systému.

úroveň 3 (strukturální) Tato úroveň přináší přesnější pohled na generativní systém z úrovně 2. Generativní systém je zde chápán jako kolekce komponent, které mezi sebou komunikují takovým způsobem, aby požadovaná data vytvářela.

Základní problém teorie systémů je analýza systémů, odvozování systémů a návrh systémů.

Analýza systému spočívá v hledání charakteristik chování systému, jehož struktura je známa. Analýza systému je přechod z vyšších úrovní, výše uvedeného popisu, do nižších. Jinými slovy se snažíme pochopit, jak systém pracuje.

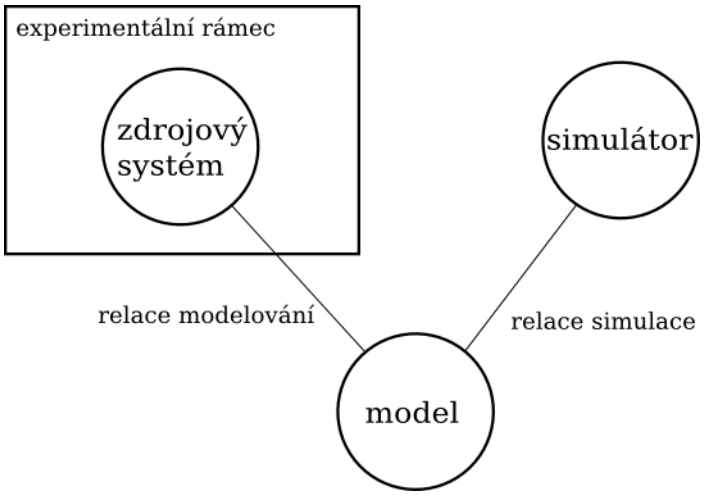
Odvozování systému znamená nalézt jeho neznámou strukturu pozorováním tohoto systému. Je to tedy přechod z nižší úrovně znalosti o systému do vyšší. Máme vypozorovaná data a snažíme se nalézt způsob, jak tyto data generovat.

Návrhem systému rozumíme nalezení alternativní struktury úplně nového systému, nebo změnu návrhu již existujícího systému. [30]

2.1 Entity v modelování a simulaci

Základní entity v modelování a simulaci a jejich vztahy ukazuje obrázek 2.1. *Zdrojový systém* je skutečné, nebo virtuální prostředí, které se snažíme modelovat. *Experimentální rámec* je množina podmínek, na které se při pozorování systému zaměřujeme. Je to prvotní akt, který nás k modelování systému vůbec vede. Můžeme například zjišťovat, jak počet a

rychlost aut projíždějících křižovatkou ovlivňuje průjezdnost křižovatky. V tomto příkladě počet a rychlost aut představuje experimentální rámec. Je zřejmé, že pro jeden zdrojový systém je možné nalézt celou řadu experimentálních rámců v závislosti na tom, co přesně chceme zjišťovat. *Modelem* můžeme označit soubor instrukcí, pravidel, rovnic, nebo omezujících podmínek pro generování vstupně/výstupního chování [30]. Jinak lze model nazvat programem pro generování dat. *Simulátor* je výpočetní systém, který je schopen model spouštět a generovat jeho chování. Simulátorem může být procesor, síť procesorů, lidská mysl, nebo abstraktněji, algoritmus [30].



Obrázek 2.1: Základní entity v modelování a simulaci.

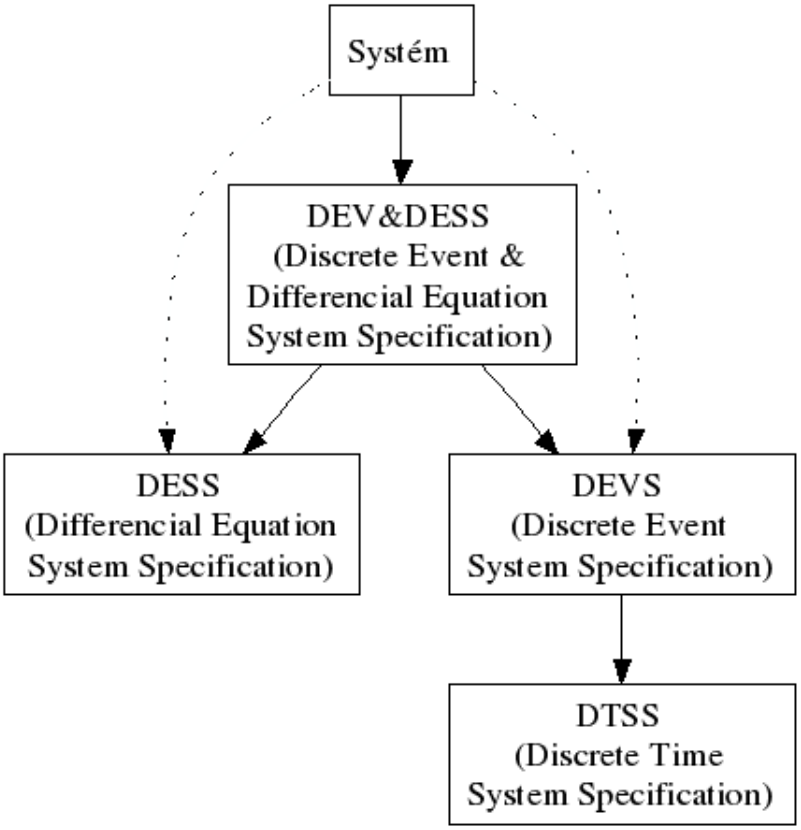
Relace mezi výše uvedenými entitami popisují základní vztahy mezi entitami. *Relace modelování* vyjadřuje vztah mezi zdrojovým systémem a modelem. Tato relace se také nazývá, více intuitivním pojmem, platnost (validita). Určuje, jak moc generované chování modelu odpovídá zdrojovému systému. *Relace simulace* je vztah mezi simulátorem a modelem. Tato relace zajišťuje, že simulátor správně provádí instrukce modelu. Důležitým pojmem v modelování a simulaci je *homomorfismus*. Homomorfismus je pojem abstraktní algebry, který mapuje jednu strukturu na druhou a zachovává při tom jejich strukturální vlastnosti. Vztah mezi simulátorem a modelem musí zachovávat homomorfismus. Tím je myšleno, že pokud model prochází určitou posloupností stavů, simulátor musí tuto posloupnost respektovat a musí projít korespondující posloupností stavů.

2.2 Typy systémů a rozdělení forem jejich specifikace

Na nejvyšší úrovni lze rozdělit systémy do dvou kategorií: *spojité* a *diskrétní*. Spojité systémy jsou takové systémy, které jsou obvykle popsány obyčejnými, nebo parciálními diferenciálními rovnicemi. Všechny proměnné spojitého systému jsou funkcemi času. Systém se vyvíjí spojitým posunem času směrem dopředu. Diskrétní systém je takový systém, jehož stav se mění okamžitě v určitých časových momentech.

Jiné rozdělení, směřující k formalismu použitého v této práci, je uvedeno na obrázku 2.2. DESS (Differencial Equation System Specification) je formalismus pro popis spojitých systémů pomocí diferenciálních rovnic. DTSS (Discrete Time System Specification) je specifikace systému popsaná diferenčními rovnicemi, tedy rekurentními vztahy. DEVS (Discrete

Event System Specification) je formalismus pro popis systémů s diskrétními událostmi. DEV&DESS je multiformalismus, který kombinuje DESS a DEVS. Protože reálné modely často nejsou výhradně diskrétní nebo spojité, je tento formalismus v praxi velice užitečný. Tato práce se zabývá formalismem DEVS, ostatní formalismy jsou tedy nadále ponechány stranou.



Obrázek 2.2: Rozdělení formalismů pro specifikaci systémů.

Kapitola 3

Discrete event system specification (DEVS)

DEVS formalismus poskytuje výpočetní základ pro implementaci struktury a chování systémů s diskrétními událostmi. První část kapitoly popisuje systémovou specifikaci *Atomický* DEVS, která slouží pro popis atomických komponent. Další část představuje *Spojovaný* DEVS, což je prostředek pro spojování atomických komponent do větších celků a dovoluje tak formovat hierarchické síťové struktury. Konec kapitoly je věnován DEVS simulátoru, algoritmu, který jednoznačně definuje provádění modelu.

3.1 Atomický DEVS

Atomický DEVS[30] je struktura

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle, \tag{3.1}$$

kde

X je množina vstupních hodnot

S je množina stavů

Y je množina výstupních hodnot

$\delta_{int} : S \to S$ je interní přechodová funkce

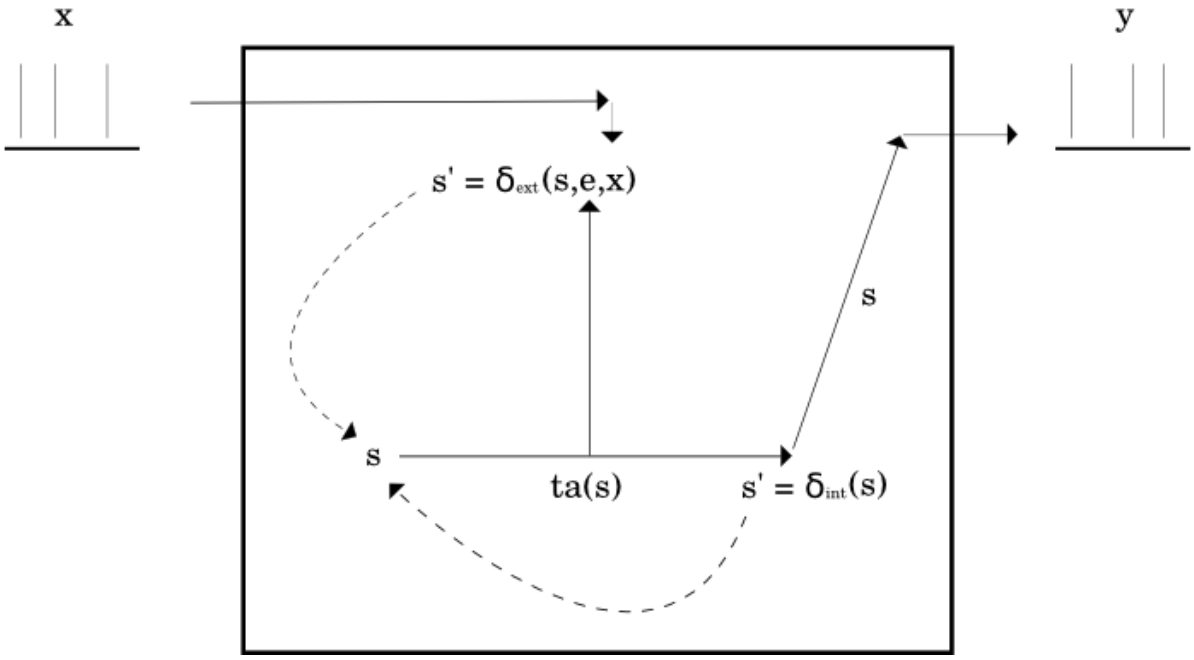
$\delta_{ext} : Q \times X \to S$ je externí přechodová funkce, kde

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ je množina totálních stavů

e je čas, který uběhl od posledního přechodu

$\lambda : S \to Y$ je výstupní funkce

$ta : S \to R^{+}_{0,\infty}$ je funkce posunu času



Obrázek 3.1: Interpretace DEVS formalismu.

Interpretaci znázorňuje obrázek 3.1. Pokud nenastane žádná vnější událost, systém setrvá ve stavu s po dobu $ta(s)$. Po vypršení této doby systém pošle na výstup $\lambda(s)$ a přejde do stavu $\delta_{int}(s)$. Poznamenejme, že výstup nastává vždy před interním přechodem.

Jestliže nastala externí událost $x \in X$, systém je v totálním stavu (s, e), kde $e \leq ta(s)$, potom dojde ke změně stavu na stav $\delta_{ext}(s, e, x)$.

Obyčejně zavádíme vstupní a výstupní porty a stavové proměnné, což lze formalizovat pomocí strukturovaných množin.

3.2 Spojovaný DEVS

Spojovaný DEVS[30] je struktura

$$N = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select \rangle, \tag{3.2}$$

kde

X je množina vstupních hodnot (událostí)

Y je množina výstupních hodnot (událostí)

D je neprázdná konečná množina názvů vnitřních komponent

$\{M_d\}$ je množina vnitřních komponent (atomických nebo spojovaných modelů)

$EIC \subseteq X \times (\bigcup_{i \in D} M_i.X)$ je relace externího vstupního propojení

$EOC \subseteq (\bigcup_{i \in D} M_i.Y) \times Y$ je relace externího výstupního propojení

$IC \subseteq (\bigcup_{i \in D} M_i.Y) \times (\bigcup_{i \in D} M_i.X)$ je relace interního propojení

$Select : 2^D \rightarrow D$ je funkce výběru konfliktních komponent

Množina vstupních a výstupních událostí poskytuje rozhraní s okolím. Interní propojení definuje spojení vnitřních komponent. Interakce s vnějšími komponenty je dána relacemi externího vstupního a výstupního propojení. Detailnější popis lze nalézt v [30].

Spojování komponent představuje koncept *zpětné vazby* [29]. Komponenta umístí hodnotu na jeden z jejích portů, přitom skutečný cíl není znám, dokud se komponenta nestane součástí většího celku a schéma propojení není známo. Tento mechanismus umožňuje návrh komponenty a její testování jako samostatné jednotky, umístění komponenty do báze modelů a její načtení na požádání, a znovupoužití komponenty všude tam, kde je její chování žádané a propojení s jinými komponentami dává smysl.

Simulátor obstarává naplánování a interakci komponent. Nejprve jsou komponenty inicializovány do jejich počátečních stavů. Simulační čas t_s je nastaven na čas t_0. Poté je u každé komponenty zjištěn zbývající čas do dalšího interního přechodu, tedy pro aktuální stav s_j komponenty j o čas $ta_j(s_j) - e_j$. Tyto události jsou seřazeny do neklesající posloupnosti a simulační čas je posunut o $min(ta_j(s_j) - e_j)$. O stejný časový přírůstek je posunut zbývající čas (e_j) do dalšího interního přechodu u každé komponenty. Pokud má více komponent naplánováno interní přechod na stejný čas, funkce výběru konfliktních komponent ($Select$) seřadí tyto komponenty do posloupnosti, podle níž má dojít k provedení přechodu.

Před samotným provedením interního přechodu je volána výstupní funkce. Relace interního propojení mapuje tyto výstupy na vstupy vnitřních komponent, což vyústí ve vyvolání externích událostí u ovlivněných komponent. Ty mohou změnit svůj stav, vynulovat zbývající čas (e_j) a aktualizovat funkci posunu času (ta_j). Teprve poté dojde k samotnému vnitřnímu přechodu. Tato posloupnost kroků se opakuje pro všechny konfliktní komponenty. Následně je seznam naplánovaných vnitřních přechodů znovu uspořádán a celý proces se opakuje. Detailněji je proces simulace popsán v následující sekci.

3.3 DEVS simulátor

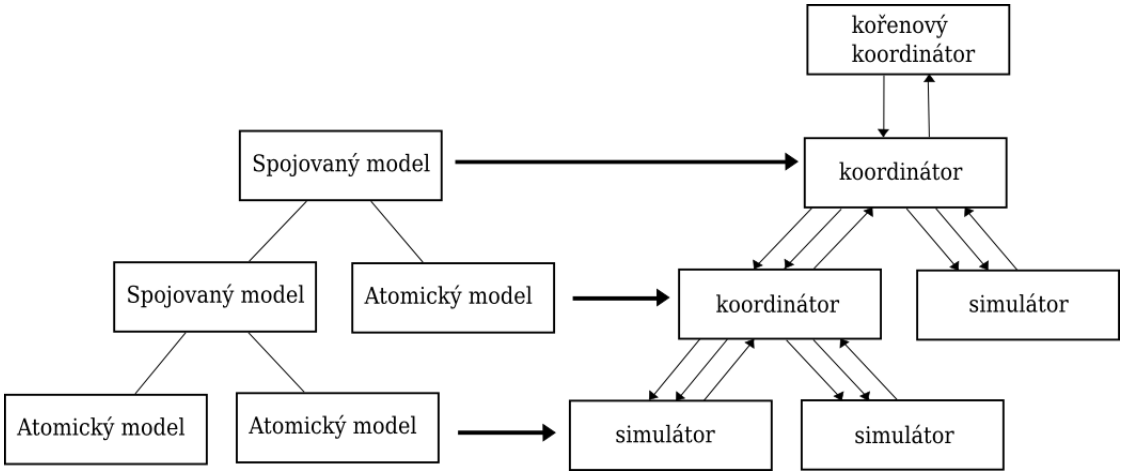
Protože je tato práce zaměřena hlavně na formalismus DEVS a na propojení atomických DEVS modelů se stavovými diagramy, považuji za důležité uvést algoritmus provádění (simulace) DEVS modelů tak, jak je uveden v [30]. Čtenář má tak lepší přehled o principech, které jsou v DEVS formalismu uplatněny.

Simulátory systémů s diskrétními událostmi pracují na základním konceptu kalendáře událostí. Kalendář událostí je seznam událostí seřazený podle času příchodu událostí. Nejbližší naplánovaná událost je z kalendáře vyjmuta a následně jsou provedeny příslušné stavové přechody. Na tomto konceptu pracuje také DEVS simulátor. DEVS simulátor je hierarchická struktura, která kopíruje hierarchickou strukturu samotného modelu. Skládá se ze tří entit:

Kořenový koordinátor je nadřazen všem simulátorům a koordinátorům a řídí celou simulaci.

Koordinátor obsahuje kalendář událostí a zajišťuje synchronizaci komponent spojovaného DEVSu.

Simulátor simuluje atomický DEVS model.



Obrázek 3.2: Korespondence hierarchického modelu s hierarchickým simulátorem.

Na obrázku 3.2 je vidět mapování hierarchického modelu na hierarchický simulátor. Koordinátor a simulátor spolu komunikují protokolem, který je složen ze čtyř zpráv:

(i,t) Inicializační zpráva. Posílá ji nadřazený koordinátor svým potomkům.

(*,t) Interní přechodová zpráva. Posílá ji nadřazený koordinátor svým potomkům. Má na starosti naplánování událostí.

(x,t) Vstupní zpráva. Je poslána nadřazeným koordinátorem svým potomkům, kteří díky ní mohou reagovat na externí událost.

(y,t) Výstupní zpráva. Je poslána potomky svému rodiči. Informují tak o svých výstupech.

Simulátor pro atomický DEVS potom vypadá takto:

```
proměnné:
        parent  // rodičovský koordinátor
        tl      // čas poslední události
        tn      // čas další události
        atomic  // asociovaný atomický model s totálním stavem (s, e)
        y       // současný výstup asociovaného modelu

metody:

        (i, t):
                tl = t - e
                tn = tl + ta(s)

        (*, t):
                if t != tn then chyba (špatná synchronizace)
                y = λ(s)
                pošli zprávu (y, t) rodiči
                s = δ_INT(s)
```

```
                tl = t
                tn = tl + ta(s)
        (x, t):
                if not (tl <= t <= tn) then chyba (špatná synchronizace)
                e = t - tl
                s = δ_EXT(s, e, x)
                tl = t
                tn = tl + ta(s)
```

Globální proměnná `t` obsahuje aktuální simulační čas. Inicializační zpráva slouží k počátečnímu nastavení lokálních simulačních časů. K výpočtu času následující interní události je zde volána funkce posunu času (`ta`). Interní přechodová zpráva je volána nadřazeným koordinátorem v době, kdy má být proveden interní přechod. Ještě před provedením interního přechodu je zjištěn výstup atomického modelu a zpráva o výstupu je zaslána nadřazenému koordinátoru. Poté je proveden samotný interní přechod a upraveny lokální simulační časy. Vstupní zpráva informuje o výskytu externí události. Předem je vypočítán čas uplynulý od poslední události (`e`) jako rozdíl aktuálního simulačního času a času poslední události. Následně je volána externí přechodová funkce modelu s příslušnými parametry a znovu upraveny lokální simulační časy.

Koordinátor je asociován se spojovanými DEVS komponentami. Zaujímá tedy místo v síťových a hierarchických modelech. Jeho úkolem je synchronizovat podřízené simulátory-/koordinátory a zachycovat externí vstupy. Koordinátor udržuje kalendář naplánovaných událostí, seřazený od nejbližší po nejzazší. Pokud jsou dvě události některých podřízených komponent naplánovány na stejný čas, přichází na řadu funkce `Select`, pomocí které jsou tyto komponenty seřazeny do požadované posloupnosti. Algoritmus koordinátoru ukazuje následující výpis:

```
proměnné:
        coupled // asociovaný spojovaný model
        parent  // rodičovský koordinátor
        tl      // čas poslední události
        tn      // čas následující události
        event-list      // kalendář událostí
        d*      // vybraný bezprostřední potomek
metody:
        (i, t):
                pošli (i, t) všem potomkům
                sort(event-list)
                tl = max {tl_d | d ∈ D}
                tn = min {tn_d | d ∈ D}
        (*, t):
                if t != tn then chyba (špatná synchronizace)
                d* = first(event-list)
                pošli zprávu (*, t) komponentě d*
                sort(event-list)
                tl = t
                tn = min {tn_d | d ∈ D}
        (x, t):
                if not (tl <= t <= tn) then chyba (špatná synchronizace)
```

```
        sendToAllInfluenced(x, t)
        tl = t
        tn = min {tn_d | d ∈ D}
(y, t): // zpráva přijatá od vybraného bezprostředního potomka d*
        if isExternal(d*, y) then
                sendOutputToParent(y, t)
        sendToAllInfluenced(y, t)
```

kde `sort(event-list)` je funkce, která řadí kalendář událostí podle času následujících událostí podřízených komponent a podle funkce Select spojovaného modelu, funkce `sendToAllInfluenced(x, t)` posílá všem potomkům, kteří jsou ovlivněni vstupem `x`, tu část vstupu `x`, která se jich týká (součástí `x` jsou porty, které v tomto případě musí odpovídat vstupním portům ovlivněných potomků), predikát `isExternal(d*, y)` vrací pravdivou hodnotu, jestliže komponenta `d*` svým výstupem `y` ovlivňuje okolí komponenty `coupled` a funkce `sendOutputToParent(y, t)` posílá rodiči tu část výstupu `y`, která se ho týká (dáno externím propojením).

Kapitola 4

StateChart

StateCharty [16] jsou vizuální formalismus pro popis chování komplexních reaktivních systémů. Poskytují možnost vyjádřit sémantiku logiky systému na velice jemné úrovni. Poznamenejme, že sémantika StateChartů je jednoznačně definovaná [1].

Formálně lze grafickou reprezentaci StateChartů popsat následovně[26]:

$$SM = \langle E, S, A, L, T, V, C \rangle, \tag{4.1}$$

kde

E je množina událostí

S je množina stavů

A je množina akcí

$L = \{(e, a) | e \in E, a \in A\}$ je množina označení přechodů

$T = \{S_{Source}, l, S_{Target}\}$**, kde** $l \in L, T \subset 2^S \times L \times 2^S, S_{Source} \subset S$ **a** $S_{Target} \subset S$**,** je množina přechodů

V je množina proměnných

C je množina stráží přechodů

Význam jednotlivých množin je intuitivní. Akce mohou být použity pro přiřazení hodnot proměnným, nebo pro generování událostí. Množinu stavů lze efektivně redukovat použitím hierarchických a paralelních stavů.

Konfiguraci StateChartu pro časový krok v čase t definujeme takto:

$$SC_t = \langle X, \Pi, \Theta, \xi \rangle, \tag{4.2}$$

kde

[1] Mezi původní sémantikou Harelových StateChartů, sémantikou STATEMATE a sémantikou UML StateChartů ve skutečnosti existují drobné rozdíly.

X je maximální stavová konfigurace posledního přechodu v čase t_i

Π je množina externích událostí, které nastaly v časovém intervalu $[t_i, t)$

Θ je množina stráží přechodů vyhodnocených jako pravda v čase t^-

ξ je funkce vracející hodnotu proměnné v čase t^-; $\xi(variable) = value$

V definici konfigurace t_i značí začátek aktuálního časového kroku a zřejmě platí $t_i \leq t$. Maximální stavová konfigurace X představuje maximální množinu složených ortogonálních stavů, které jsou tvořeny pouze základními stavy, tedy stavy, které nemají žádné následovníky. Konfigurace StateChartu reprezentuje totální stav modelu. Na konci časového kroku je množina Θ a funkce ξ aktualizována na základě akcí, které se provedly v tomto časovém kroku.

4.1 Notace a sémantika

Jak již bylo uvedeno v úvodu této kapitoly, StateCharty jsou vizuální formalismus, jejich síla tedy tkví ve vizuální reprezentaci. Tato spolu se sémantickou jednoznačností tvoří atraktivní prostředek pro tvorbu modelů operujících v reálném čase.

Na obrázku 4.1 je ukázka StateChartu převzatého z [10] demonstrujícího některé jeho možnosti. Stavy jsou vyznačeny zaoblenými obdélníky. Rozlišujeme stavy *základní* (`zero`, `reg`, ...), stavy *složené* (`stopwatch`) a stavy *ortogonální* (`display`, `run`). Všimněme si také *pseudo-stavu* [2] zobrazeného jako černě vyplněný kruh, který značí počáteční pseudo-stav. Stavy jsou mezi sebou propojeny přechody (šipky), ty mohou obsahovat událost, která daný přechod aktivuje a stráž, která, je-li vyhodnocena jako pravda, dovolí aktivovat daný přechod.

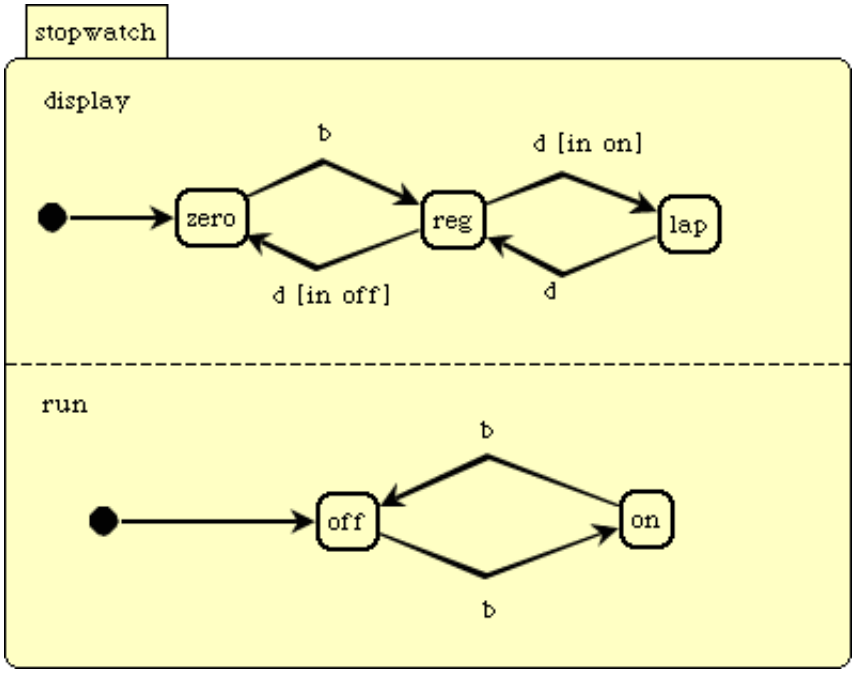
Je důležité si uvědomit, že koncept StateChartů přesahuje jakoukoli notaci, ať už grafickou, nebo textovou. UML specifikace jasně rozlišuje sémantiku a notaci StateChartů. Dále je třeba poznamenat, že notace UML StateChartů není čistě vizuální. Syntaxe akcí a stráží přechodů není definována. V praxi to znamená, že notace je závislá na konkrétním simulačním jazyce. Tato práce se snaží od této závislosti oprostit použitím jazyka Scheme, který je platformně nezávislý. Více o tomto problému pojednávají pozdější kapitoly.

Formální model StateChartů nedisponuje konceptem zabývajícím se časem. Tento nedostatek lze ovšem obejít zavedením speciálních časových funkcí. Příkladem takové funkce je *timeout(e, n)*, která po n časových krocích vyšle událost e. V následující kapitole je této funkce využito při mapování StateChartů na atomické DEVS komponenty.

Časový krok ve StateChartech je dále rozdělen na konečnou posloupnost *μ-kroků*. Každý *μ-krok* koresponduje s jedním přechodem (v případě ortogonálních stavů s několika přechody). Externí událost aktivuje jednu a více vnitřních událostí. Životnost vnitřní události je dána jedním *μ-krokem*. Po obsluze všech vnitřních událostí pro daný časový krok StateChart vstupuje do stabilní konfigurace. Význam *μ-kroků* se projevuje při použití složených a ortogonálních stavů.

Pro dokončení popisu sémantiky StateChartů je třeba formálně definovat reakci systému:

[2] Označení *pseudo-stav* používáme pro takové stavy, které nemají všechny vlastnosti ostatních stavů, jako například akce `onentry` a `onexit`, spouštěné při vstupu, resp. výstupu ze stavu.



Obrázek 4.1: Ukázka StateChartu modelujícího stopky.

$$SR_t = \langle \Upsilon, \Pi^g \rangle, \tag{4.3}$$

kde

Υ je množina přechodů, které mají být provedeny souběžně v časovém intervalu $[t_i, t)$; $\Upsilon \subset T$

Π^g je množina událostí generovaných Υ

Reakce systému SR_t následuje konfiguraci SC_t. Υ a Π^g představují historii přechodů a vnitřních událostí, které byly provedeny v časovém kroku t. Jinými slovy každá položka těchto množin koresponduje s jedním *μ-krokem*. Obě množiny jsou generovány provedením *μ-konfigurací* (μSC), kde $\mu SC_0 = SC_t$. μSC je definována podobně jako konfigurace SC_t a byla neformálně představena s vysvětlením *μ-kroků*. Detailní formální definici lze nalézt v [15].

Délka časového kroku je určena nejbližší naplánovanou událostí StateChartu. Události naplánované na danou dobu jsou přidány do množiny externích událostí a StateChart provede svůj první *μ-krok*. Po provedení každého *μ-kroku* je množina generovaných událostí rozeslána přes celý StateChart a na požádání jsou aktualizovány příslušné proměnné. *μ-kroky* jsou opakovaně prováděny, dokud StateChart nedosáhne stabilní konfigurace. Časy naplánovaných událostí jsou aktualizovány a celá procedura se opakuje.

4.2 Ekvivalence StateChartů a atomických DEVS komponent

Ekvivalence StateChartů a atomických DEVS komponent poskytuje formální základ pro rozšíření jazyka DEVSML o popis chování atomických DEVS komponent pomocí podmnožiny StateChartů. Dále je v práci tato podmnožina StateChartů označována jako stavové diagramy, nebo stavové automaty. Pokud vím, tak ustálená formální grafická notace pro popis chování atomických DEVS komponent neexistuje. Atomické DEVS komponenty vychází stejně jako StateCharty z koncepce konečných automatů. Z tohoto důvodu a z důvodů uvedených v předchozích sekcích jsou StateCharty logickým kandidátem pro popis chování atomických DEVS komponent.

Tato sekce představuje obousměrné mapování StateChartů a atomických DEVS komponent tak, jak je to uvedeno v [26]. Symboly jsou pro větší přehlednost označeny názvem příslušného formalismu. Jejich význam odpovídá významu uvedeném v předchozích definicích.

$$X_{DEVS} \leftrightarrow \Pi_{StateChart} \tag{4.4}$$
$$Y_{DEVS} \leftrightarrow \Pi^{g}_{StateChart} \tag{4.5}$$

DEVS události jsou složeny z názvu a hodnoty. Naopak, události StateChartů jsou založeny pouze na názvu. Proto je nutné mapovat DEVS událost na dvojici (proměnná, událost) StateChartu.

$$(s \in S_{DEVS}, e) \in Q_{DEVS} \leftrightarrow X_{StateChart} \tag{4.6}$$

$X_{StateChart}$ je definováno jako maximální stavová konfigurace v určitém časovém okamžiku. To lze mapovat na totální stav DEVSu.

$$\delta_{DEVS} : Q_{DEVS} \times (X_{DEVS} \cup \{\emptyset\}) \rightarrow S_{DEVS} \leftrightarrow \tag{4.7}$$
$$\delta_{StateChart} : X_{StateChart} \times \Upsilon_{StateChart} \rightarrow X_{StateChart} \tag{4.8}$$

δ_{DEVS} zapouzdřuje externí a interní přechod do jedné funkce. $\delta_{StateChart}$ mapuje stavovou konfiguraci před časovým krokem a množinu přechodů $\Upsilon_{StateChart}$ na novou stavovou konfiguraci po časovém kroku.

$$\lambda_{DEVS} \leftrightarrow \lambda_{StateChart} \tag{4.9}$$

Výstupní funkce StateChartu $\lambda_{StateChart}$ je nepřímo specifikovaná systémovou reakcí. Lze ji definovat jako podmnožinu akcí, které jsou přiřazeny přechodům $\Upsilon_{StateChart}$ provedeným během časového kroku. Akce obsahují události, které během časového kroku aktualizují proměnné, nebo způsobí jiné přechody. Stejně jako DEVS výstupní funkce, i tato funkce vrací výstupní hodnoty před koncem příslušného časového kroku.

Funkce posunu času je v DEVS formalismu již zahrnuta. Aby jsme ji mohli mapovat do StateChart formalismu, je třeba vytvořit speciální funkci *timeout(s, t)*. Parametr s specifikuje stav, do kterého StateChart vstoupí po uplynutí doby t. Mapování potom vypadá následovně:

$$ta_{DEVS}(s) \rightarrow ta_{StateChart} : timeout(s', ta_{DEVS}(s)) \tag{4.10}$$

Jestliže StateChart nepoužívá časové funkce (samozřejmě vyjma výše uvedené), opačné mapování je triviální:

$$ta_{DEVS}(s) = 0 \leftarrow ta_{StateChart} \tag{4.11}$$

(Pokud StateChart časové funkce používá, je třeba napodobit plánování StateChart simulátoru přidáním speciálních DEVS komponent.)

4.3 SCXML

SCXML (StateChart eXtensible Markup Language) [7] je jazyk pro popis StateChartů. Jedná se o nadcházející standard W3C pocházející z pracovní skupiny Voice Browser. Kombinuje koncepty z CCXML (Call Control eXtensible Markup Language) a Harelových stavových diagramů. Původně byl vytvořen jako vylepšený popis stavových modelů z CCXML, tedy jazyka, který je jednou ze součástí popisu multimodální interakce [3] na webu. Je navržen jako řídící jazyk pro novou verzi VoiceXML 3.0 [4] a má tak nahradit CCXML. Poslední pracovní verze vyšla 16. května 2008.

Dodejme, že i přes původní účely vzniku je tento jazyk navržen zcela obecně jako notace pro StateCharty. Jako alternativu k SCXML je možné uvažovat XMI [5]. Ve své práci používám SCXML z důvodu srozumitelnější syntaxe a snazší manipulace než v případě XMI.

Výpis 4.1 ukazuje SCXML reprezentaci StateChartu 4.1. Příklad netřeba dále komentovat, výklad je intuitivní. XPath predikát *In()* má význam při použití ortogonálních stavů, kde se dotazuje na stav v paralelní větvi.

Specifikace jazyka SCXML v poslední podobě obsahuje – pro reprezentaci výrazů a dalších konstrukcí použitých například ve strážích přechodů a akcích – jazyky ECMAScript a XPath [6]. Jak již bylo uvedeno, v této práci je pro reprezentaci výrazů použit jazyk Scheme.

```
<scxml initial="stopwatch" xmlns="http://www.w3.org
   /2005/07/scxml" version="1.0">
<state id="stopwatch">
```

[3] Multimodální interakce poskytuje uživateli rozhraní se systémem. Tradičně je používána klávesnice a myš, ale v úvahu je třeba brát i další možnosti, jako třeba ovládání hlasem a další.

[4] VoiceXML je jazyk pro popis hlasových dialogových systémů.

[5] XMI (XML Metadata Interchange) je standard skupiny Object Management Group (OMG) pro výměnu metadat prostřednictvím XML. Může být použit pro popis jakýchkoli metadat, jejichž metamodel lze vyjádřit v Meta-Object Facility (MOF), tedy standardu určeného pro modelem řízené inženýrství.

[6] Původně byl jazyk XPath navržen pro navigaci uvnitř XML dokumentu, v dnešní době je to bohatý jazyk pro specifikaci výrazů.

```
    <parallel>
      <state id="display">
        <initial>
          <transition target="zero"/>
        </initial>
        <state id="zero">
          <transition event="b" target="reg"/>
        </state>
        <state id="reg">
          <transition event="d" target="lap" cond="In(on)"/>
          <transition event="d" target="zero" cond="In(off)"
             />
        </state>
        <state id="lap">
          <transition event="d" target="reg"/>
        </state>
      </state>
      <state id="run">
        <initial>
          <transition target="off"/>
        </initial>
        <state id="off">
          <transition event="b" target="on"/>
        </state>
        <state id="on">
          <transition event="b" target="off"/>
        </state>
      </state>
    </parallel>
  </state>
  </scxml>
```

Výpis 4.1: Ukázka SCXML popisu StateChartu 4.1.

4.4 Rozšíření SCXML pro potřeby DEVSML

Jazyk SCXML tak jak je uveden v pracovním návrhu organizace W3C [7] pro popis atomickým DEVS komponent nestačí. Tento jazyk byl navržen pro jiné účely, neobsahuje tedy prvky, jako je například časová prodleva setrvání ve stavu, nebo výstup stavového automatu. Proto bylo třeba jazyk SCXML rozšířit tak, aby vyhovoval potřebám popisu atomických DEVS komponent.

Jak je uvedeno v prvním odstavci této sekce, rozšíření se týká čekacích dob v každém ze stavů a specifikace výstupu stavového automatu pro jednotlivé stavy. Přibyly tedy dva nové uzly: `<wait>` a `<output>`. Oba uzly se ve stromu XML dokumentu vyskytují jako potomci uzlu `<state>`. Následuje popis těchto uzlů:

`wait` Uzel neobsahuje žádné atributy. Jediný potomek tohoto uzlu je textový element,

který obsahuje Scheme výraz, vracející číselnou hodnotu, čekací dobu v daném stavu (ekvivalent funkce posunu času atomického DEVSu ta).

`output` Uzel obsahuje jediný atribut, a to atribut `port`, tedy název portu, na který má být výstup odeslán. Hodnotu výstupu určuje Scheme výraz, který je uveden jako dětský textový element.

Dodejme, že dané rozšíření nijak neporušuje koncept StateChartu a z něho odvozeného stavového diagramu, který je v DEVSML použit. Výstup (`output`) lze brát jako součást reakce systému, přesněji generovanou událost. Formální definice již byla uvedena, odkazuji tímto čtenáře na (4.3) a (4.9). Také parametr `wait` je v souladu z konceptem, lze jej namapovat na speciálně zavedenou funkci `timeout(s, t)` uvedenou v (4.10) a (4.11).

Kapitola 5

Nástroj pro tvorbu stavových diagramů

DEVS formalismus vychází z koncepce stavových automatů. Atomické komponenty jsou popsány stavy a externí či interní události jejich stavy mění. To vede k popisu atomických komponent právě pomocí stavových diagramů. Popis pomocí kódu atomických DEVS funkcí není přirozeným způsobem popisu DEVS modelů. Popis pomocí grafického jazyka je běžnější pro inženýry, kteří vytváří modely strojů a zařízení. [12] Proto byl pro pohodlnou tvorbu stavových diagramů navržen vizuální nástroj. Ukázku tohoto nástroje lze vidět na obrázku 5.1. Vizuální popis stavového prostoru atomických komponent vede k lepšímu pochopení a orientaci v modelu. Vizuální reprezentace tedy slouží také jako dokumentační popis.

Vnitřně je diagram reprezentován pomocí meta vzoru dědičnost chování. Stavy zde vystupují jako metody. Správné volání metod zajišťuje mechanizmus, který v reakci na externí událost a současný stav vyvolá patřičné stavové metody, v nichž signalizuje akci, která se má provést. Více o meta vzoru dědičnost chování pojednává sekce 8.1.4.

5.1 Architektura

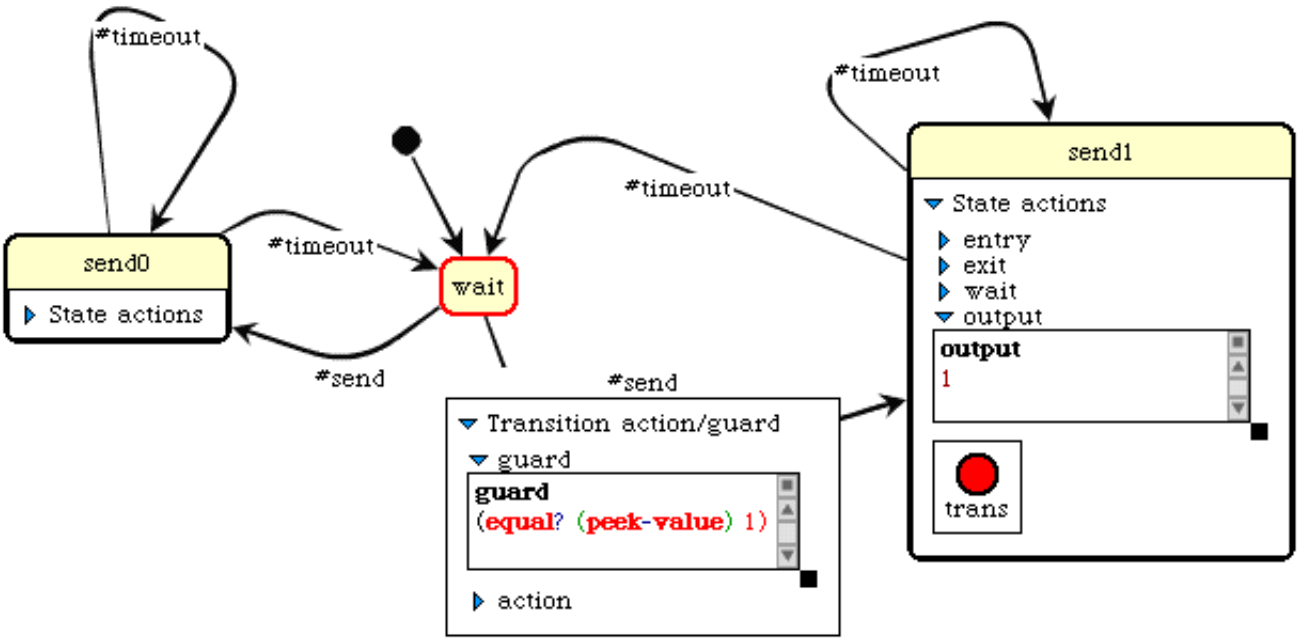
Při návrhu architektury nástroje pro tvorbu stavových diagramů byl kladen důraz na nezávislost modelu na grafickém uživatelském rozhraní (dále jen GUI). Cílem bylo také celou aplikaci koncipovat tak, aby vytvářela samostatný celek bez zásadních změn stávajícího GUI systému SmallDEVS. Vznikl tak nezávislý [1] balík, který lze snadno do systému SmallDEVS importovat.

Dalším požadavkem bylo, aby aplikace byla do GUI SmallDEVS přirozeně začleněna. Požadavku bylo docíleno integrací nástroje pro tvorbu stavových diagramů do inspektoru atomických komponent (viz. obrázek 5.2). Uživateli je tak dopřán komfort jednotného uživatelského rozhraní. Celá tvorba stavového diagramu se odehrává v inspektoru atomické komponenty a práce uživatele tak není narušena dalšími zbytečnými grafickými prvky.

Základní architektura systému je vidět na obrázku 5.3. Jak je vidět, grafické prvky diagramu (grafické zobrazení stavů, přechodů) operují nad jednotlivými prvky modelu diagramu. Nad tím vším stojí inspektor, který udržuje odkaz na kreslící plochu a seskupuje grafické prvky dohromady.

Je důležité poznamenat, že informace o grafických vlastnostech prvků stavového diagramu (pozice, zaoblení hran) jsou ukládány přímo v modelu stavového diagramu. To je

[1] Některé změny v GUI SmallDEVS byly nutné, ale jedná se opravdu o jemné úpravy.



Obrázek 5.1: Ukázka nástroje pro tvorbu stavových diagramů.

důležité při znovuotevření inspektoru, kdy jsou vlastnosti načteny a diagram vypadá naprosto stejně, tak jak vypadal před zavřením inspektoru.

5.2 Použití

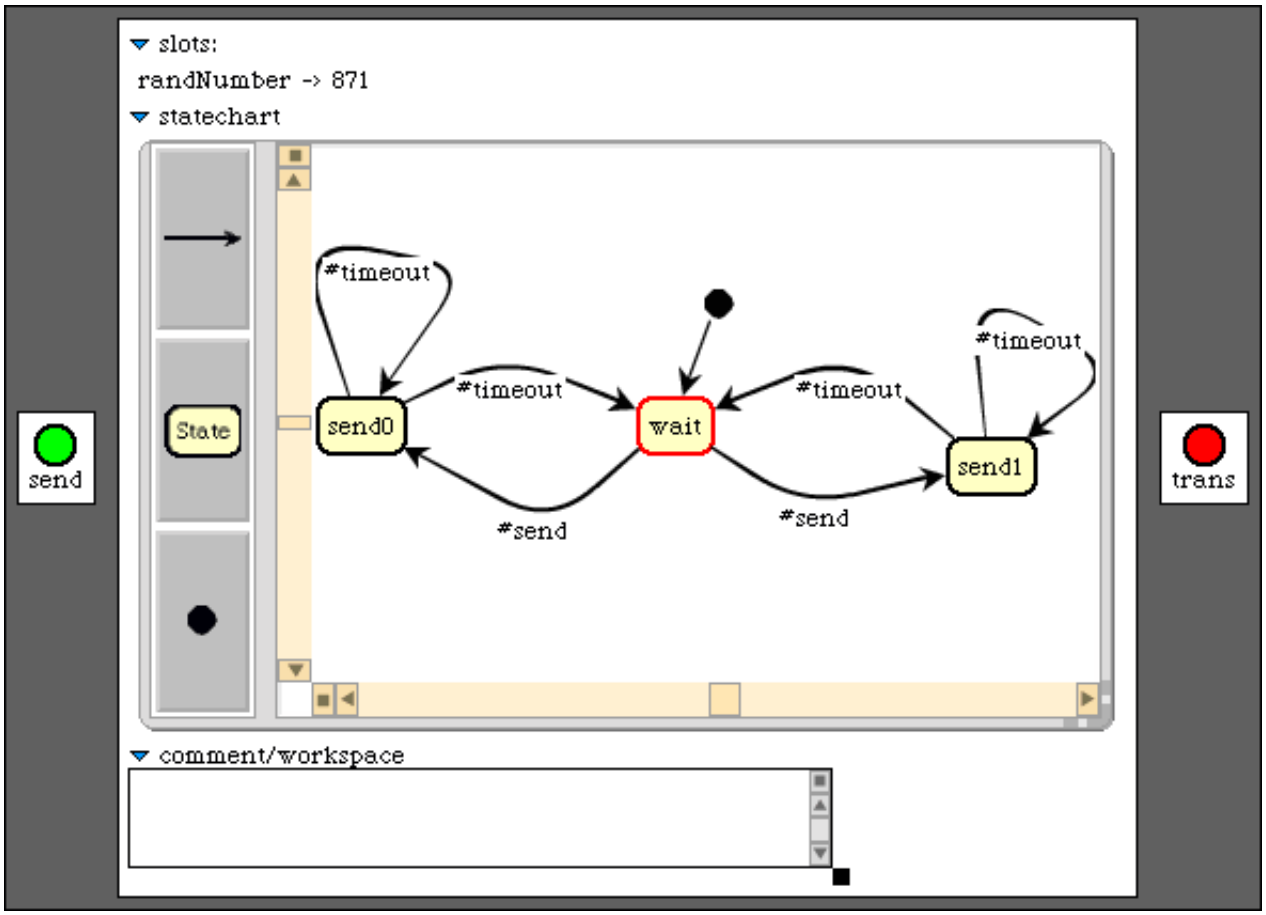
Práce s nástrojem je intuitivní a jednoduchá. Je-li vytvořena atomická komponenta, je možné nad ní spustit inspektor. Existují tři typy inspektorů podle typu popisu chování komponenty, které lze nad atomickou komponentou otevřít.

První z nich je klasický inspektor, ve kterém je chování komponenty popisováno funkcemi atomického DEVSu. Jedná se o inspektor, který se již ve SmallDEVS GUI nacházel a nemá se stavovými diagramy nic společného.

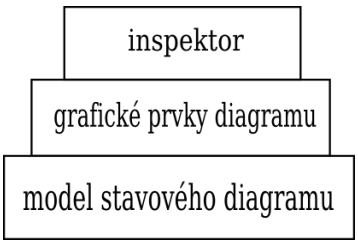
Druhý typ inspektoru slouží k vytváření stavových diagramů s popisem akcí/stráží v jazyce Smalltalk. Takto vytvořené stavové diagramy nejsou platformně nezávislé a proto je není možné exportovat do jazyka DEVSML.

Třetí typ inspektoru slouží k vytváření stavových diagramů s popisem akcí/stráží v jazyce Scheme. Takto vytvořené modely jsou zcela platformně nezávislé a lze je do jazyka DEVSML exportovat. Poznamenejme, že v jednom spojovaném modelu lze vytvářet atomické komponenty s různým popisem. V extrémním případě můžou být některé komponenty popsány pomocí DEVS funkcí, další pomocí stavových diagramů s popisem akcí/stráží v jazyce Smalltalk a jiné popsané pomocí stavových diagramů s inskripčním jazykem Scheme. Zřejmě by se ale jednalo o vzácný případ a, jak již bylo uvedeno, takové modely nelze do jazyka DEVSML exportovat.

Pokud je již jednou atomická komponenta otevřena s určitým interpretrem, je informace



Obrázek 5.2: Ukázka inspektoru atomického DEVS modelu a v něm zabudovaného nástroje pro tvorbu stavových diagramů.



Obrázek 5.3: Architektura nástroje pro tvorbu stavových diagramů.

o použitém interpretru uložena. Při dalším otevření s jiným interpretrem nebude tento brán v potaz. Změna interpretru je možná, ale je třeba ji provést ručně ve workspace. Výpis 5.1 ukazuje změnu na interpretr jazyka Scheme.

```
| i |
i := StatechartLispInterpreter new.
i statechart: self statechart.
```

```
self statechart interpreter: i.
```

Výpis 5.1: Změna interpretru.

Při automatické změně stavového diagramu za běhu simulace (viz. sekce 5.7) se po vytvoření nových prvků diagram samostatně neaktualizuje. Automatická změna diagramu za běhu simulace je spíše experimentální a aktualizace by tudíž zbytečně zpomalovala klasické modely. Ruční rekonstrukce je samozřejmě možná, lze ji vyvolat výběrem volby *reconstruct diagram* z menu diagramu. Před tímto krokem doporučuji pozastavit simulaci.

Před každým spuštěním simulace dochází k validaci stavového diagramu. Validace je velice jednoduchá, kontroluje pouze, jestli stavový diagram není prázdný a jestli obsahuje právě jeden počáteční pseudo-stav, ze kterého vede právě jeden přechod do některého ze zbylých stavů.

V probíhající simulaci je současný stav v diagramu zvýrazněn, uživatel má tak přehled o tom, co se v diagramu děje. Další příjemná vlastnost je možnost ručního výběru současného stavu. To je velice užitečné při ladění modelu.

5.3 Popis akcí/stráží pomocí jazyka Smalltalk.

Jak již bylo uvedeno, existují dva inskripční jazyky, které lze využít pro popis akcí/stráží stavových diagramů. Jedním z nich je právě jazyk Smalltalk. Modely popsané jazykem Smalltalk nejsou platformně nezávislé, na druhou stranu ale umožňují využít všech výhod tohoto jazyka.

Na popis jazykem Smalltalk nejsou kladena žádná omezení, je možné použít libovolné konstrukce, objekty a metody. Zřejmě je výhodné použít tento jazyk při konstrukci složitějších modelů pro vytvoření prvotního prototypu. V této fázi je možné využít všech prostředků, které jazyk Smalltalk, konkrétně prostředí Squeak, s různými rozšířeními nabízí. Jako příklad můžeme uvést balík `PlotMorph`, pomocí kterého lze vytvářet různé grafy a sledovat tak třeba stavové, vstupní a výstupní trajektorie modelu.

Metody, které jsou v popisu akcí/stráží k dispozici v reakci na externí událost:

peekValue Vrací hodnotu externí události.

peekPort Vrací název portu, na který externí událost právě dorazila.

elapsed Vrací čas uběhnutý od poslední události.

Nastavení/získání hodnot slotů se provádí pomocí přístupových metod. Například nastavení slotu s názvem `mySlot` na hodnotu `5` se provede takto: `self mySlot: 5`. Pro získání hodnoty slotu `mySlot` se použije konstrukce: `self mySlot`.

Pro nastavení čekací doby stavu na hodnotu ∞ je vyhrazena konstrukce `Float infinity`.

5.4 Popis akcí/stráží pomocí jazyka Scheme.

Druhou možností, jak popisovat akce/stráže stavového diagramu je popis pomocí jazyka Scheme [27]. Scheme je multiparadigmatický programovací jazyk, který vznikl z funkcionálního jazyka Lisp [20]. Jazyk Scheme je typický svým minimalistickým přístupem. Snaží se, aby jádro jazyka poskytovalo co nejmenší počet primitiv a ostatní, složitější konstrukce nabízely přiložené knihovny.

Podporu pro práci s interpretry jazyka Lisp a z něho odvozených derivátů poskytuje v prostředí Squeak Smalltalku balík `LispKit` [4]. Tento balík obsahuje i interpretr jazyka Scheme, který je v této práci použit. Je tedy zaručena kompatibilita jazyků použitých pro popis akcí/stráží stavových diagramů mezi prostředími SmallDEVS a Adevs (s pomocí knihovny Guile, viz. dále). Jazyk Scheme je také platformně nezávislý a existuje mnoho jeho implementací pro různé platformy.

Pokud vytváříme modely, které mají být převoditelné do prostředí Adevs s použitím knihovny Guile, je dobré si nejprve ověřit, jestli použité funkce knihovna Guile nabízí. Pokud ne, je možné do vygenerovaného souboru `ModelName/scripts/libs/DEVSML_scheme_primitives.scm` dodat jakýkoli Scheme kód a rozšířit tak možnosti Guile. Načtené funkce budou k dispozici ve všech stavových diagramech všech atomických komponent celého modelu. V prostředí SmallDEVS lze načítat nové knihovny ve workspace atomické komponenty, kde jsou nové funkce potřeba. Lze to provést provedením například příkazu:

```
self statechart interpreter top: #(require $'random)
```

Předchozí příkaz načetl knihovnu `random`. Knihovna `random` je umístěna v adresáři `./ulisp/slib/` (uvedeno relativně vzhledem k image Squeaku). V tomto adresáři se musí nacházet všechny knihovny načítané tímto způsobem.

Funkce, které jsou v popisu akcí/stráží k dispozici v reakci na externí událost jsou analogií metod použitých v popisu pomocí jazyka Smalltalk:

(peek-value) Vrací hodnotu externí události.

(peek-port) Vrací název portu, na který externí událost právě dorazila.

(elapsed) Vrací čas uběhnutý od poslední události.

Nastavení/získání hodnot slotů se provádí pomocí funkcí `(set-slot slotName slotValue)`, resp. `(get-slot slotName)`, kde `slotName` je textový řetězec obsahující název slotu a `slotValue` je s-výraz vracející hodnotu, která má být danému slotu přiřazena.

Pro nastavení čekací doby stavu na hodnotu ∞ je vyhrazena funkce `(float-infinity)`.

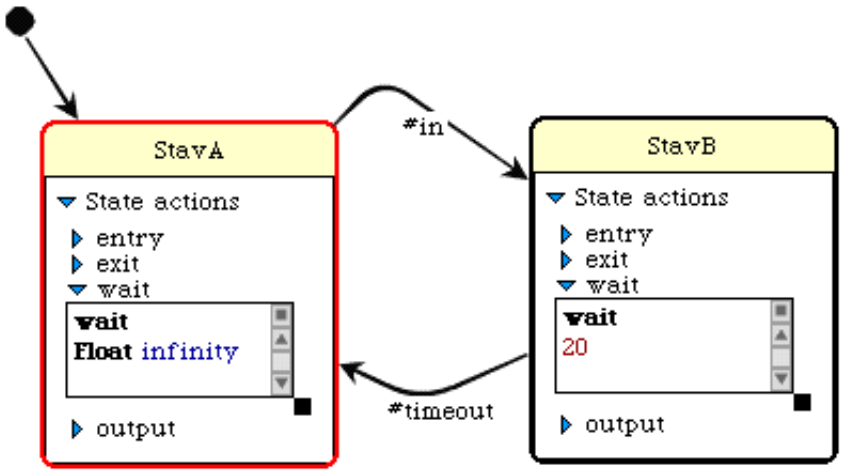
5.5 Pořadí provádění akcí stavového automatu

Na obrázku 5.4 je ukázka jednoduchého stavového automatu, na kterém bude demonstrováno pořadí provádění jednotlivých akcí stavů/přechodů. Jak je vidět na obrázku, automat se na počátku simulace nachází ve stavu `StavA`. V tomto stavu setrvává, dokud na port `#in` nedorazí externí událost, následně přechází do stavu `StavB`. Ve stavu `StavB` automat naopak setrvává 20 časových jednotek a poté přechází zpět do stavu `StavA` (externí události ignoruje). Stráže přechodů jsou nastaveny na hodnotu `true`. Poznamenejme, že při inicializaci modelu (nastává také při restartu simulace) proběhne akce `entry` stavu `StavA`.

Sled akcí při příchozí externí události na portu `#in` je následující:

1. `StavA` $\Longrightarrow$ `StavB` (přechodová akce)
2. `StavA exit`
3. `StavB entry`

4. ... 20 časových jednotek
5. `StavB output`
6. `StavB` $\Longrightarrow$ `StavA` (přechodová akce)
7. `StavB exit`
8. `StavA entry`



Obrázek 5.4: Stavový automat pro demonstraci provádění akcí.

Pro úplnost uvádím jak by vypadalo pořadí provedení akcí dle specifikace UML statecharts [22]:

1. `StavA exit`
2. `StavA` $\Longrightarrow$ `StavB` (přechodová akce)
3. `StavB entry`
4. ... 20 časových jednotek
5. `StavB output`
6. `StavB exit`
7. `StavB` $\Longrightarrow$ `StavA` (přechodová akce)
8. `StavA entry`

Je vidět, že dle specifikace UML statecharts je akce opuštění stavu (`exit`) provedena před přechodovou akcí. Tato interpretace je bližší konceptu Mealyho automatu, kde jsou akce prováděny v přechodech. Protože ale provedení akce trvá určitý časový okamžik, představuje tato interpretace koncepční problém. Během provádění akce není automat v žádném

stavu (zdrojový stav již opustil a do cílového se ještě nedostal). Naproti tomu Moorův automat uvažuje provádění akcí ve stavech. Stav systému je tedy v každém časovém okamžiku přesně definován. [2]

5.6 Skriptové vytváření stavového diagramu

Při implementaci byl kladen důraz na možnost jednoduchého skriptového vytváření stavového diagramu a jednoduchého začlenění do objektového modelu atomických komponent systému SmallDEVS. Rozhraní pro skriptové vytváření automatu je fasádou nad podsystémem rozhraní elementů stavového diagramu. Rozhraní elementů je zcela fasádou zastíněno a uživatel je při skriptování automatu od vnitřní architektury zcela odloučen.

Vytvoření automatu a jeho začlenění do atomické DEVS komponenty lze rozdělit do tří kroků:

1. vytvoření stavového diagramu,
2. vytvoření rozhraní mezi stavovým diagramem a atomickou DEVS komponentou a
3. přidání rozhraní jako delegáta do modelu atomické DEVS komponenty.

K vytvoření stavového diagramu slouží tři metody hlavního objektu `Statechart`. Všechny vrací odkaz na nově vytvořený prvek. Následuje krátký popis metod a parametrů, které přebírají:

`addSimpleState: anAsocArray` Přidá nový stav do diagramu. Asociativní pole, které přebírá jako parametr obsahuje položky s klíči (všechny hodnoty pod těmito klíči mají formu textového řetězce):

- `#name` Název stavu.
- `#entry` Vstupní akce stavu.
- `#exit` Výstupní akce stavu.
- `#wait` Čekací doba stavu.
- `#output` Výstup stavového automatu pro daný stav.
- `#outputEvent` Výstupní port.

`addStartState` Přidá nový počáteční pseudo-stav.

`addTransition: anAsocArray` Zavede přechod mezi dvěma stavy (nebo jedním a tím stejným, pokud se jedná o vlastní přechod). Asociativní pole obsahuje položky s klíči:

- `#from` Zdrojový stav (počátek) přechodu. Hodnotou je textový řetězec udávající název stavu, nebo přímo objekt stavu. Pokud je to možné, doporučuji předávat objekt stavu. Při zadání názvu se prohledávají všechny stavy diagramu a hledá se příslušný objekt. Poznamenejme, že se nejedná o kritický problém, protože stavy jsou (většinou, výjimku tvoří dynamické vytváření stavů, viz. 5.7) vytvářeny před samotnou simulací, takže ke zpomalení průběhu simulace nedochází.

[2] Poznamenejme, že Mealyho a Moorův automat jsou matematicky ekvivalentní. Jeden může být vždy převeden na druhý a naopak.

`#to` Koncový stav přechodu. Platí zde to stejné, co pro hodnotu pod klíčem `#from`.

`#event` Textový řetězec s názvem portu, na který externí událost dorazila a vyvolala tak daný přechod. Speciálním případem je uvedení hodnoty `'timeout'`. Událost `timeout` je vyvolána po uplynutí čekací doby stavu.

`#guard` Textový řetězec se strážní podmínkou. Hlídá provedení přechodu.

`#action` Textový řetězec s akcí, která se v reakci na přechod provede.

Rozhraní mezi atomickou DEVS komponentou a stavovým diagramem zajišťuje objekt `StatechartDEVSTrait`. Rozhraní implementuje funkce atomického DEVSu a zajišťuje tak komunikaci se stavovým diagramem. Objekt `StatechartDEVSTrait` stačí pouze vytvořit a přidat jej jako delegáta do modelu atomické DEVS komponenty. Kompletní proces ukazují následující výpisy. Vytváří se zde atomická komponenta `Generator`. Komponenta `Generator` pouze v náhodných okamžicích generuje úlohy a posílá je na výstup.

První fáze, vytvoření stavového diagramu a příslušných slotů:

```
generatorMachine := Statechart new.

sInit := generatorMachine addSimpleState: {#name -> 'Init'
    . #wait -> '0'}.
sGenerate := generatorMachine addSimpleState: {
    #name -> 'Generate'.
    #entry -> 'self n: self n + 1'.
    #wait -> '0'.
    #output -> '(self jobPrototype setSizeBetween: self sa
        and: self sb) clone n: self n; yourself'.
    #outputEvent -> 'out'}.
sTimer := generatorMachine addSimpleState: {
    #name -> 'Timer'.
    #wait -> '(self ia to: self ib) atRandom'}.
sStart := generatorMachine addStartState.

generatorMachine addTransition: {#from -> sStart. #to ->
   sInit}.
generatorMachine addTransition: {#from -> sInit. #to ->
   sGenerate. #event -> 'timeout'}.
generatorMachine addTransition: {#from -> sGenerate. #to
   -> sTimer. #event -> 'timeout'}.
generatorMachine addTransition: {#from -> sTimer. #to ->
   sGenerate. #event -> 'timeout'}.

jobPrototype := PrototypeObject new.
jobPrototype addSlot: 'n' withValue: 0.
jobPrototype addSlot: 'size' withValue: 0.
jobPrototype addSlot: 'name' withValue: 'aJob'.
jobPrototype addMethod: 'setSizeBetween: sl and: sh self
   size: (sl to: sh) atRandom'.
```

```
generatorMachine addSlot: 'jobPrototype' withValue:
    jobPrototype.
generatorMachine addSlot: 'ia' withValue: 2.
generatorMachine addSlot: 'ib' withValue: 7.
generatorMachine addSlot: 'sa' withValue: 5.
generatorMachine addSlot: 'sb' withValue: 10.
generatorMachine addSlot: 'n' withValue: 0.
```

Výpis 5.2: Skriptové vytvoření stavového diagramu a příslušných slotů.

Druhá a třetí fáze, vytvoření modelu atomické komponenty, vytvoření rozhraní mezi atomickou komponentou a stavovým diagramem a přidání rozhraní jako delegáta do modelu komponenty:

```
generatorModel := AtomicDEVSPrototype new.
generatorModel addOutputPorts: {#out}.
generatorModel removeDelegate: 'defaultTrait'.
generatorModel addDelegate: 'statechartDEVSTrait'
    withValue: StatechartDEVSTrait new.
```

Výpis 5.3: Vytvoření modelu atomické komponenty a rozhraní se stavovým diagramem.

V předchozím výpise si všimněme odstranění implicitního delegáta `defaultTrait`. Delegát obsahuje DEVS funkce s implicitním tělem. U stavového diagramu jsou DEVS funkce implementovány v objektu `statechartDEVSTrait`, je tedy nutné implicitního delegáta odstranit.

V poslední řadě otevření inspektoru nad modelem popsaným stavovým diagramem:

```
generatorModel openAsStatechart.
```

Výpis 5.4: Otevření inspektoru nad modelem.

5.7 Automatická změna stavového diagramu za běhu

Stávající implementace dovoluje vytvářet nové stavy a přechody ve stavovém diagramu automaticky za běhu simulace. Je tedy možné, aby se chování atomické komponenty samo dynamicky měnilo. Stavový diagram tedy může za běhu upravovat sám sebe a automaticky tak formovat požadované chování.

Nejlépe je vše vysvětlit na demonstračním příkladu. Uvažujme automat uvedený na obrázku 5.5, který na začátku obsahuje 2 stavy: `Idle` a `Process`. Stav `Idle` je počátečním stavem. Automat je součástí atomické DEVS komponenty se vstupním portem `#in` a výstupním portem `#unknown` (5.6). Automat si ve vnitřním slotu udržuje kolekci známých objektů (objektů, které již někdy dříve dorazily na vstupní port).

Automat pracuje tak, že čeká ve stavu `Idle` na vnější událost. Jestliže na vstupní port dorazil objekt, se kterým se automat ještě nesetkal (nemá ho uložen v kolekci známých objektů), přechází do stavu `Process`. Bezprostředně po vstupu do stavu `Process` (akce `#entry`), vytvoří automat nový stav reprezentující nový objekt a dále dva nové přechody:

První, vycházející ze stavu `Idle` a vstupující do nového stavu. Tento přechod reaguje na vstupní událost na portu `#in` a jeho strážní podmínka hlídá, aby se do nového stavu přešlo jen tehdy, jestliže nový stav reprezentuje objekt, který se objeví na portu `#in`. Druhý, přecházející z nového stavu do stavu `Idle` v reakci na událost `#timeout` (tedy po vypršení čekací doby nového stavu). Čekací dobou je simulována aktivita, která by v reakci na nový objekt mohla eventuálně probíhat.

Po vypršení čekací doby stavu `Process` (stanovena na 10 časových jednotek), přechází automat zpět do stavu `Idle`.

Připomeňme, že při opětovném přijetí stejného objektu má již automat pro tento objekt vyhrazen stav a přechází do něj. V tomto stavu čeká 5 časových jednotek a poté se vrací do stavu `Idle`. [3]

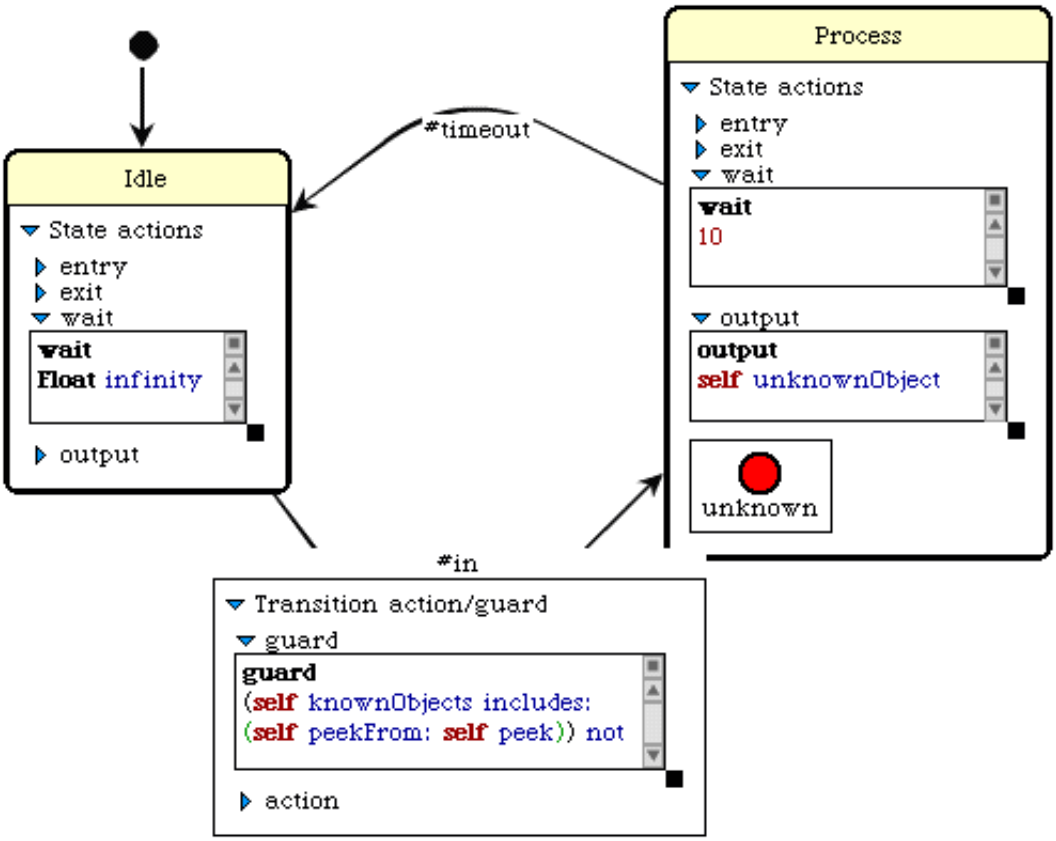
Níže uvedený výpis ukazuje akci `#entry` stavu `Process`, kde se dynamicky vytváří nový stav a nové přechody.

```
self unknownObject: (self peekValue).
self knownObjects add: self unknownObject.
self addTransitionDynamicaly: {
 #from -> 'Idle'.
 #to -> (self addSimpleStateDynamicaly: {
       #name -> ('DynState', self unknownObject asString).
       #wait -> '5'}).
 #event -> 'in'.
 #guard -> ('(''', self unknownObject asString,''' = (self
     peekValue) asString)')
}.
self addTransitionDynamicaly: {
 #from -> ('DynState', self unknownObject asString).
 #to -> 'Idle'.
 #event -> 'timeout'
}
```

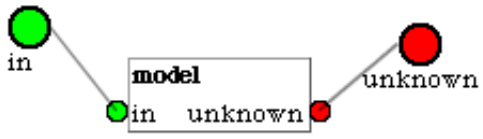
Výpis 5.5: Dynamické vytvoření stavu a přechodů.

Úryvek ze simulačního běhu demonstrují obrázky 5.7a, 5.7b, 5.7c a 5.7d.

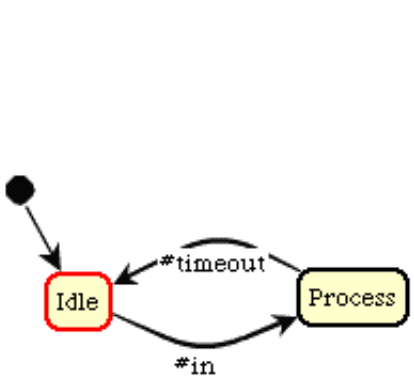
[3] Délky čekacích dob ve stavech Process a v nově dynamicky vytvářených stavech byly stanoveny zcela náhodně.



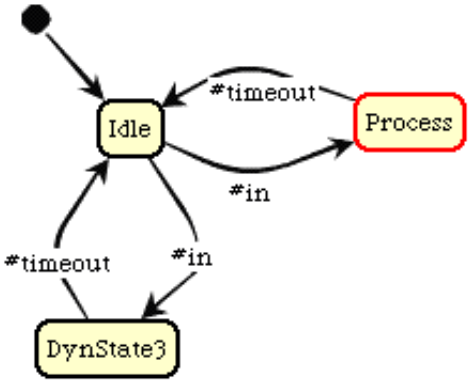
Obrázek 5.5: Dynamický stavový diagram.



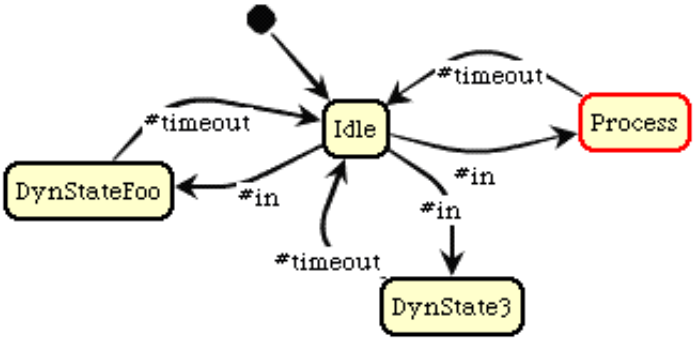
Obrázek 5.6: Atomická DEVS komponenta řízená dynamickým stavovým diagramem.



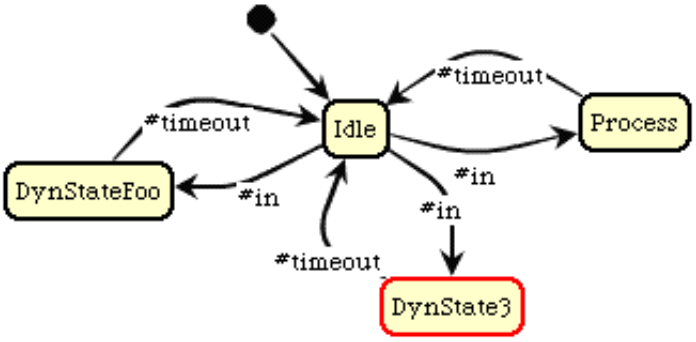
(a) Začátek simulace. Dosud nepřišla žádná externí událost.



(b) Na port `#in` dorazil nový objekt - `SmallInteger` s hodnotou `3`. Poznamenejme, že ještě před tímto krokem bylo pro přehlednost vynecháno provedení přechodu ze stavu `Process` zpět do stavu `Idle`.



(c) Na port `#in` dorazil nový objekt - `String` s hodnotou `'Foo'`.



(d) Na port `#in` dorazil již známý objekt - `SmallInteger` s hodnotou `3`.

Obrázek 5.7: Automatická změna stavového diagramu za běhu simulace.

Kapitola 6

DEVSML

Jazyk DEVSML (DEVS Meta Language) [18] je XML jazyk pro popis struktury i chování DEVS modelů. Jedná se o platformně nezávislý jazyk, který umožňuje sdílet modely napříč platformami. Modely popsané pomocí DEVSML mohou být beze změny transformovány do různých simulačních prostředí.

Obrázek 6.1 ukazuje, jak by mohl obecně vypadat generátor DEVSML. Podle typu popisu atomických komponent se generuje příslušný XML popis chování atomických DEVS komponent. Tato práce se zabývá modelováním atomických komponent pomocí stavových diagramů, popis pomocí Petriho sítí je naznačen pouze ilustračně jako možné rozšíření. Jazyk SmalltalkML [1] jsem nejprve uvažoval jako skriptovací jazyk pro jazyk SCXML. Jelikož by ale převod takového jazyka do prostředí C/C++ nebyl vůbec snadný, navíc by zřejmě nepřinesl žádné výhody, použil jsem proto jazyk Scheme. Výhody použití jazyka Scheme vysvětluje kapitola 7.

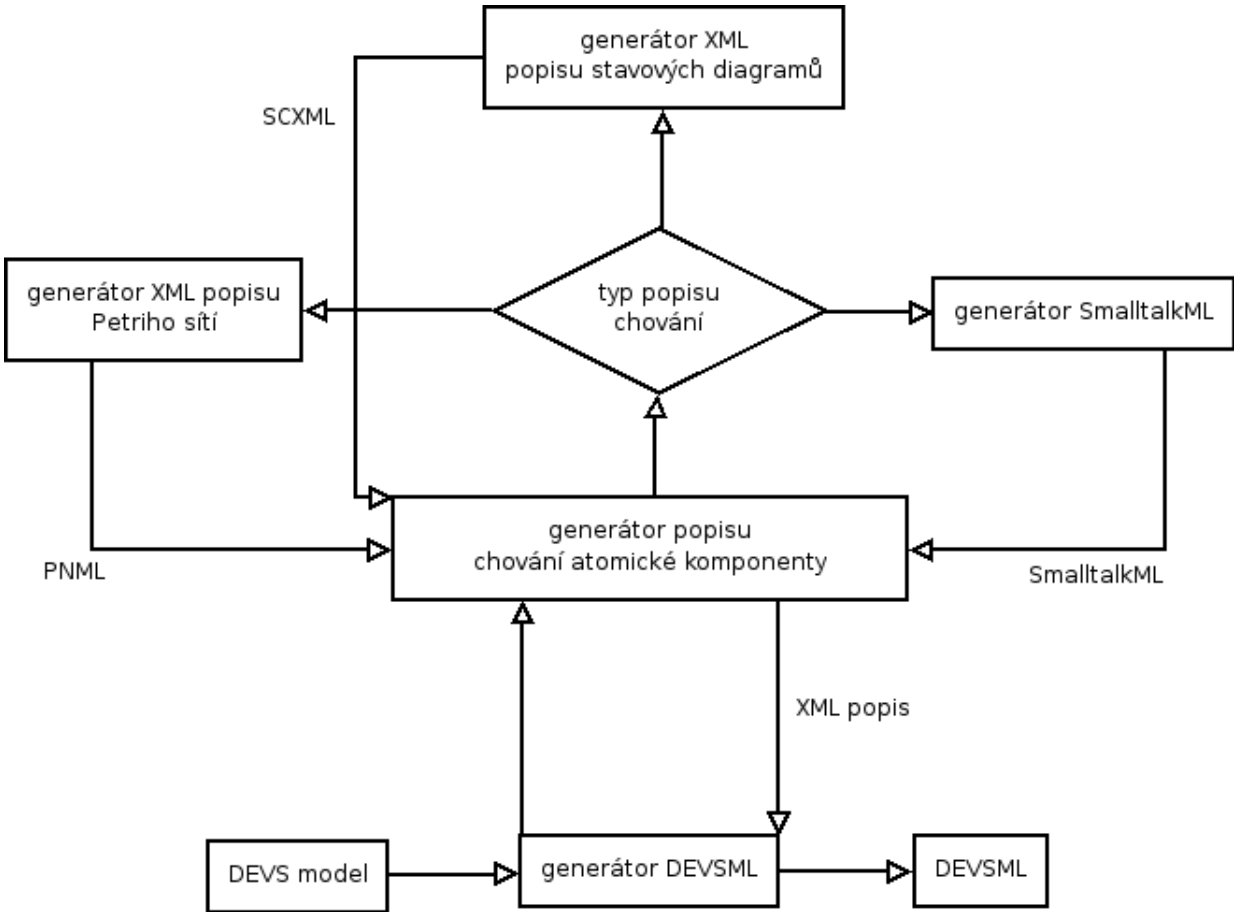
6.1 Struktura modelu v DEVSML

Díky hierarchické struktuře XML dokumentu může být hierarchická struktura DEVS modelu mapována do XML velice snadno [18].

Na obrázku 6.2 je ukázka spojované komponenty síťového přepínače převzatá z [30]. Komponentu tvoří tři atomické komponenty: přepínač (switch) a dva procesory (processor1, processor2). V závislosti na vnitřní konfiguraci přepínače je vstup spojované komponenty, který se dále stává vstupem přepínače, poslán na vstup buď prvního nebo druhého procesoru. Úkolem každého z procesorů je pak zpracovat vstup a výsledek poslat na výstup spojované komponenty. Výpis 6.1 ukazuje ekvivalentní zápis v jazyce DEVSML.

```
<coupled name="netswitch">
  <ports>
    <input>
      <port name="in"/>
    </input>
    <output>
      <port name="out"/>
    </output>
  </ports>
```

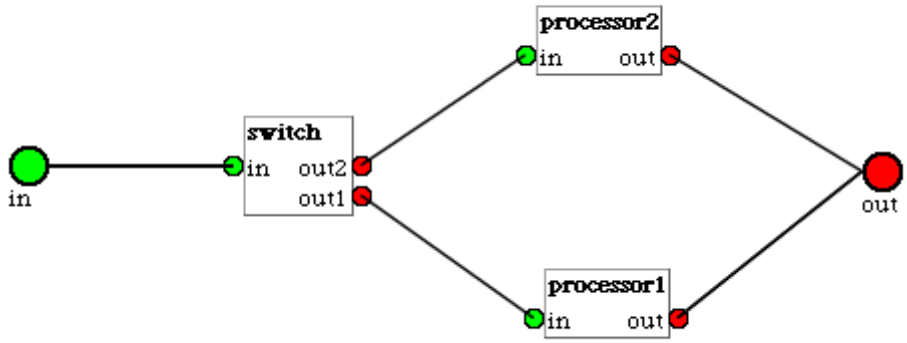
[1] Tento jazyk měl být původně vytvořen na základě inspirace z jazyka JavaML [8].



Obrázek 6.1: Generátor DEVSML.

```
    <D>
      <component name="switch"/>
      <component name="processor1"/>
      <component name="processor2"/>
    </D>
    <influences>
      <influence source="self" source-port="in"
             target="switch" target-port="in" />
      <influence source="switch" source-port="out1"
             target="processor1" target-port="in" />
      <influence source="switch" source-port="out2"
             target="processor2" target-port="in" />
      <influence source="processor1" source-port="out"
             target="self" target-port="out" />
      <influence source="processor2" source-port="out"
             target="self" target-port="out" />
    </influences>
  </coupled>
```

Výpis 6.1: DEVSML popis komponenty z obrázku 6.2.



Obrázek 6.2: Spojovaná DEVS komponenta síťového přepínače.

Výpis 6.1 předvádí všechny prvky DEVSML popisu spojovaných komponent. Žádné jiné elementy než ty, které jsou ve výpise uvedeny, se v DEVSML specifikaci spojovaných komponent nenachází. Význam se zdá být zřejmý, ale pro úplnost přece jenom uvádím krátký popis elementů:

`coupled` Obaluje celou spojovanou komponentu, jediným atributem je atribut `name`, jenž udává název spojované komponenty.

`ports` Bez atributu. Pouze zapouzdřuje vstupní a výstupní porty.

`input` Obal vstupních portů. Bez atributu.

`output` Obal výstupních portů. Bez atributu.

`port` Port. Atribut `name` udává název portu.

`D` Bez atributu. Kontejner výpisu komponent.

`component` Vnitřní komponenta. Atribut `name` udává název komponenty.

`influences` Kontejner obalující popis propojení komponent. Bez atributu.

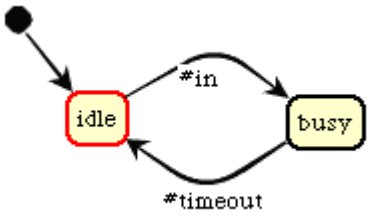
`influence` Propojení dvou komponent. Atributy:

- `source` Název zdrojové komponenty. V případě, že jde o obalovou komponentu, uvádí se klíčové slovo `self`.
- `source-port` Název zdrojového portu.
- `target` Název cílové komponenty. Pro obalovou komponentu zde platí to stejné, co u atributu `source`.
- `target-port` Název cílového portu.

6.2 Popis chování atomických komponent v DEVSML

Popis atomických komponent v DEVSML je navržen tak, aby umožňoval použití nejen stavových diagramů, ale i dalších formalismů. Z tohoto důvodu jsou zavedeny tzv. *profily*. Každá atomická komponenta je popsána právě jedním *profilem*. Cílem této práce je implementovat *profil* `scxml`. Jak již název napovídá, jedná se o popis jazykem SCXML. Obecně je ale možné uvažovat i jiné *profily*, jako například popis pomocí XML reprezentace Petriho sítí, SmalltalkML, atd.

Na obrázku 6.3 je vidět atomická komponenta procesor popsaná pomocí stavového diagramu. Jako inskripční jazyk je použit jazyk Scheme. Procesor čeká na příchozí úlohu, po jejím přijetí ji nějakou dobu zpracovává a po dokončení ji pošle na výstup. Úlohy přijaté během doby zpracování jsou zahozeny. Všimněme si stavové akce `wait`. Ta určuje dobu, po kterou stavový automat v daném stavu setrvá. Poznamenejme, že stavový automat žádným časovým řízením nedisponuje. Časový okamžik uvedený u akce `wait` není nic jiného, než funkce posunu času ($ta(s)$) klasického DEVSu. Po uplynutí této časové doby a v případě, že nenastala žádná externí událost, je uměle vyvolána událost `timeout`, která vždy označuje interní přechod.



Obrázek 6.3: Stavový diagram atomické komponenty procesor.

Následující výpis ukazuje ekvivalentní zápis atomické komponenty procesor v jazyce DEVSML:

```
<atomic name="procesor" profile="scxml">
  <ports>
    <input>
      <port name="in"/> </input>
    <output>
      <port name="out"/> </output></ports>
  <scxml initial="idle" version="1.0" xmlns="http://www.w3.org
     /2005/07/scxml">
    <state id="busy">
      <onentry></onentry>
      <onexit></onexit>
      <wait>(get-slot &quot;jobSize&quot;)</wait>
      <output port="out">(get-slot &quot;jobSize&quot;)</output
         >
      <transition event="timeout" target="idle">
        nil
      </transition>
    </state>
```

```
        <state id="idle">
          <onentry></onentry>
          <onexit></onexit>
          <wait>(float-infinity)</wait>
          <output></output>
          <transition event="in" target="busy">
            (set-slot &quot;jobSize&quot; (peek-value))
          </transition>
        </state>
      </scxml>
    </atomic>
```

Výpis 6.2: Atomická komponenta procesor v DEVSML.

I zde uvádím stručný popis jednotlivých elementů DEVSML specifikace atomických komponent:

`atomic` Obaluje atomickou komponentu. Atribut `name` udává název komponenty, atribut `profile` použitý profil. V současné době je podporován pouze profile `scxml`.

`ports` Kontejner všech portů komponenty. Bez atributu.

`input` Obal vstupních portů.

`output` Obal výstupních portů.

`port` Port. Atribut `name` udává název portu.

`scxml` Začátek upraveného SCXML popisu stavového diagramu komponenty. Důležitý je atribut `initial`, který obsahuje název počátečního stavu.

`state` Stav. V atributu `id` je název stavu.

`onentry` Vstupní akce stavu. Bez atributu. Vnitřní textový element obsahuje výrazy v jazyce Scheme.

`onexit` Výstupní akce stavu. Obsah stejný jako u elementu `onentry`.

`wait` Čekací doba stavu. Vnitřní textový element obsahuje výraz jazyka Scheme, který musí vracet číselnou hodnotu.

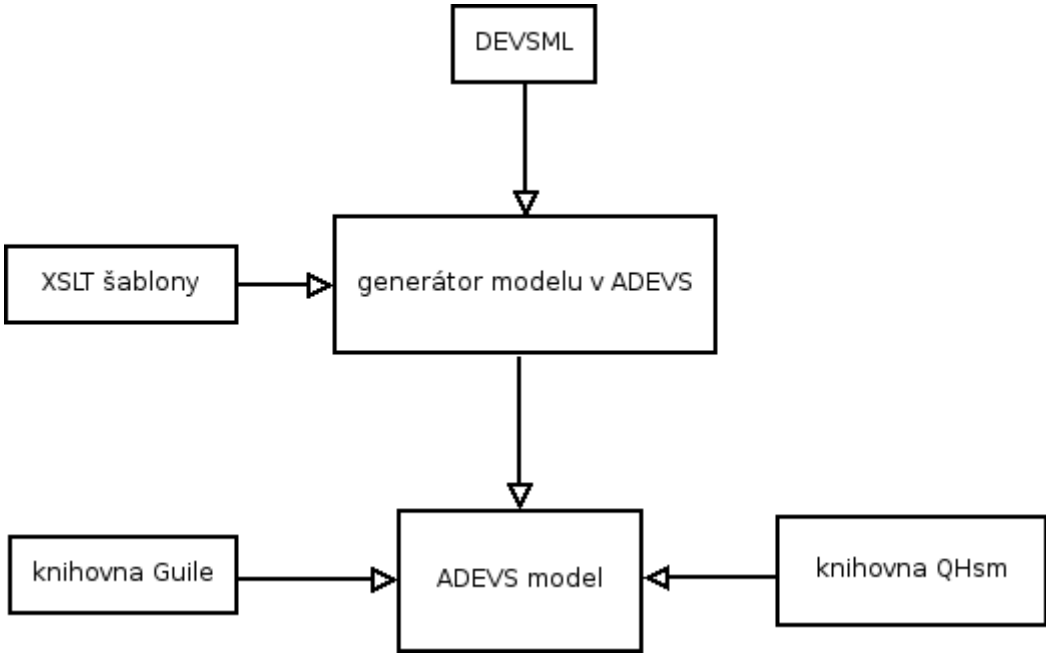
`output` Výstup stavového diagramu pro daný stav. Atribut `port` obsahuje název portu, na který bude výstup poslán. Ve vnitřním textovém elementu jsou výrazy jazyka Scheme, jejichž vyhodnocení se získá výstupní hodnota.

`transition` Přechod. V atributu `event` je uveden název portu (v případě externí události, která má vyvolat daný přechod), nebo klíčové slovo `timeout`, pokud se jedná o interní přechod. Atribut `target` obsahuje název cílového stavu. V atributu `cond` je specifikována strážní podmínka jako výraz v jazyce Scheme. A konečně vnitřní textový element obsahuje přechodovou akci popsanou výrazy jazyka Scheme.

Kapitola 7

Převod DEVSML do prostředí Adevs

V této kapitole je představen návrh a knihovny použité v implementaci transformátoru modelů popsaných jazykem DEVSML do prostředí Adevs. Zároveň je tak i vodítkem pro tvorbu transformátorů do jiných prostředí. Návrh systému je vidět na obrázku 7.1. Vstupem transformátoru je model popsaný jazykem DEVSML, výstupem pak ekvivalentní model popsaný jazykem C/C++ a využívající knihoven Adevs [1], QHsm [5] a Guile [3].



Obrázek 7.1: Transformace jazyka DEVSML do prostředí Adevs.

Transformátor převádí DEVSML pomocí XSLT transformací na C/C++ kód. Modely DEVS komponent a samotný simulátor jsou popsány pomocí knihovny Adevs. Každé atomické komponentě je přiřazen stavový automat popsaný knihovnou QHsm. Akce/stráže zapsané v jazyce Scheme jsou interpretovány knihovnou Guile.

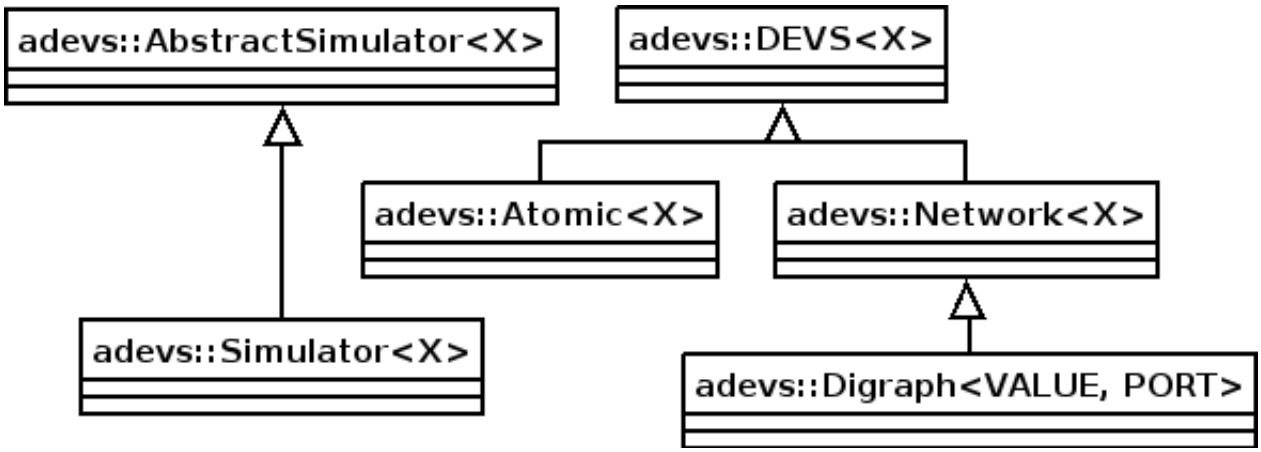
Následující sekce popisují použité knihovny podrobněji.

7.1 Adevs

Adevs [1] (A Discrete EVent System Simulator) je C++ knihovna pro konstrukci simulací modelů založených na diskrétních událostech popsaných pomocí formalismů Parallel DEVS a Dynamic Structure DEVS (DSDEVS).

Třídy použité v této práci jsou vidět na obrázku 7.2. Každá DEVS komponenta modelu odpovídá jedné C++ třídě. Spojované komponenty dědí od třídy `adevs::Digraph`, atomické komponenty od třídy `adevs::Atomic`. V třídě atomických komponent se přetěžují DEVS funkce. Spojovaná komponenta pouze vytvoří objekty svých komponent a zavede příslušná propojení. Třída `adevs::Simulator` implementuje simulátor. Konstruktoru objektu simulátoru je předána vnější obalová spojovaná komponenta. Na rozdíl od systému SmallDEVS, kde je možné za běžící simulace explicitně vyvolávat externí události, v prostředí Adevs toto možné není, znamená to tedy, že simulace je ukončena v případě, že jsou všechny komponenty „ustáleny“, jinými slovy funkce posunu času (ta) vrací u všech komponent speciální hodnotu `DBL_MAX` (ekvivalent `Float infinity` v systému SmallDEVS).

Na závěr sekce poznamenejme, že knihovna Adevs umí pracovat i s proměnnou strukturou DEVS modelů. Tato práce se ale tímto problémem nadále nezabývá.



Obrázek 7.2: Hierarchie tříd Adevs použitých v této práci.

7.2 QHsm

QHsm [25] (Quantum Hierarchical State Machines) je knihovna pro tvorbu hierarchických stavových diagramů v C/C++. Stavové diagramy jsou implementovány na základě meta vzoru dědičnosti chování (více v 8.1.4). Stavy jsou zde reprezentovány pomocí ukazatelů na členské metody třídy `QHsm`. Stavové členské metody vrací ukazatel na rodičovský stav (další ukazatel na členskou metodu). Tímto způsobem je udržována informace o hierarchické struktuře stavového diagramu. Stavové metody přebírají jako parametr objekt třídy `QEvent`, který v sobě nese signál dané události. Typicky má potom stavová metoda následující strukturu:

```
QState ConcreteMachine::stateA(ConcreteMachine* me, QEvent
    e){
  switch (e->sig){
    case Q_ENTRY_SIG:
      A_entry();          // vstupní akce stavu
      return Q_HANDLED();
      break;
    case Q_EXIT_SIG:
      A_exit();           // výstupní akce stavu
      return Q_HANDLED();
      break;
    case SIGNAL_1:        // uživatelem definovaný signál
      if (AtoB_guard()){        // strážní podmínka přechodu
        AtoB_action();        // přechodová akce
        return Q_TRAN(&ConcreteMachine::stateB);
      }
      break;
    default:
      break;
  }
  return Q_SUPER(&QHsm::top);
}
```

Výpis 7.1: Stavová metoda v QHsm.

Jak lze vidět v předchozím výpise, převod z grafické podoby stavového diagramu na příslušný spustitelný kód je přímočarý. Tato knihovna se také osvědčila při převodu z podmnožiny SCXML použité v DEVSML na spustitelný C++ kód.

Alternativou ke knihovně QHsm by mohla být také knihovna The Boost Statechart Library [2] (dále jen BSL). Původně jsem tuto knihovnu v práci uvažoval, nakonec jsem se ale pro knihovnu QHsm rozhodl ze dvou důvodů. Za prvé jsem implementaci QHsm použil i pro stavové diagramy ve SmallDEVS. Za druhé je knihovna BPL konstruována dle UML specifikace provádění akcí (rozdíly viz. sekce 5.5). Z důvodu zvolení QHsm pro stavové diagramy ve SmallDEVS by bylo nežádoucí, aby se modely po transformaci chovaly jinak.

7.3 Guile

Guile [3] je knihovna určená pro rozšíření aplikací napsaných v programovacím jazyce C o moduly. Jedná se o interpretr programovacího jazyka Scheme zapouzdřený do knihovny. Knihovna navíc umožňuje rozšíření sebe sama.

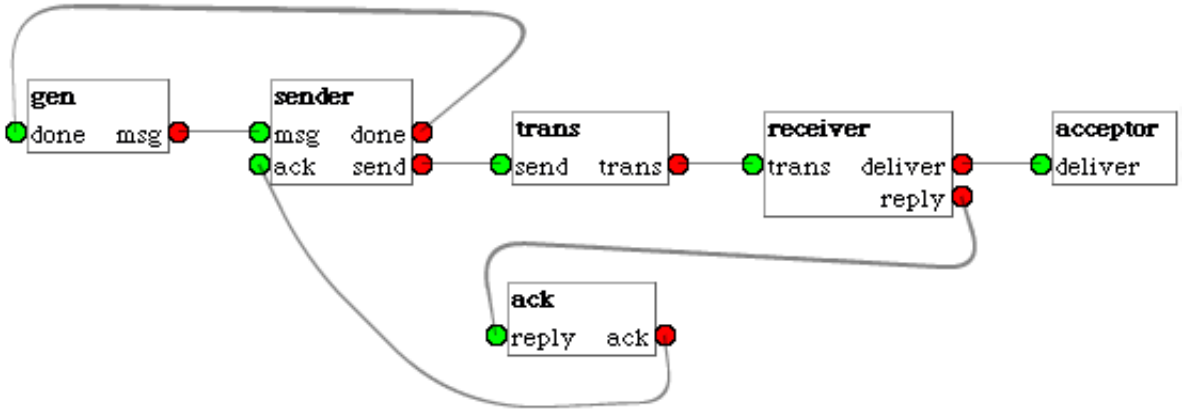
Guile podporuje jazyk Scheme podle specifikace R5RS [6]. Navíc tuto specifikaci rozšiřuje o:

- interaktivní dokumentační systém,
- podporu pro síťové programování dle specifikace POSIX a
- GOOPS - Guile framework pro objektově orientované programování.

Knihovna je použita nejen pro interpretaci akcí/stráží napsaných v jazyce Scheme, ale struktury `SCM`, tedy objekty jazyka Scheme (Scheme je dynamicky typovaný jazyk), jsou použity i pro komunikaci mezi komponentami. Tím je docíleno použití jednotného datového typu pro komunikaci přes porty komponent, což vyžaduje knihovna Adevs. Navíc jsou tyto struktury použity také pro práci se sloty. Shrneme-li to, tak všechny datové toky procházejí přes knihovnu Guile.

7.4 Demonstrace převodu na modelu Alternating Bit Protocol (ABP)

Tato sekce si klade za cíl názorně předvést vytvoření a převod modelu ze SmallDEVS do Adevs. K tomuto účelu byl vybrán model systému Alternating Bit Protocol (ABP). Celý model byl vytvořen ve SmallDEVS s použitím nástroje pro tvorbu stavových diagramů. K dispozici je i skriptově vytvořený ekvivalentní model, který je uveden v příloze.



Obrázek 7.3: Spojovaná DEVS komponenta systému Alternating Bit Protocol (ABP).

Celkový proces sestává ze tří kroků:

1. Vytvoření modelu s popisem akcí/stráží v jazyce Scheme.
2. Export modelu do formátu DEVSML.
3. Transformace modelu popsaného jazykem DEVSML do prostředí Adevs.

Další odstavec stručně popisuje systém Alternating bit Protocol. Následuje ukázka vytvořených stavových diagramů. Konec sekce se věnuje samotnému převodu do prostředí Adevs.

ABP je protokol síťové vrstvy, který je odolný proti ztrátě či duplikaci posílaných zpráv. Na obrázku 7.3 je vidět struktura modelu systému ABP. Systém sestává ze šesti atomických komponent. Úkolem komponenty Sender je posílat zprávy přes síťový kanál komponentě Receiver. Komponenta Receiver přijaté zprávy potvrzuje skrze komponentu Ack. Předpokládáme, že síťový kanál vedoucí přes komponentu Trans může zprávy ztrácet, nebo duplikovat (ale ne porušit [1]). Potvrzovací kanál vedoucí přes komponentu Ack může zprávy

[1] Pokud bychom uvažovaly opravdu spolehlivý protokol, zprávy bychom musely opatřit kódy pro detekci chyb, které by porušení dokázaly identifikovat. V uvedeném příkladě kódy pro detekci chyb neuvažuji.

ztrácet. Sender a Receiver disponují časovačem, který kontroluje, jestli v daném časovém intervalu byla zpráva doručena. Pokud nebyla, předpokládá se, že zpráva byla ztracena a je odeslána znovu. Model Gen posílá zprávy modelu Acceptor. Protokol musí zajistit, že data přijatá modelem Acceptor budou identická z daty generovanými modelem Gen. Zprávy jsou tagovány jedním bitem (0/1). Potvrzovací zprávy jsou tvořeny jedním bitem (0/1). Pokud Sender pošle zprávu tagovanou bitem 0 a obdrží potvrzení s bitem 1, nebo vypršel daný časový interval, došlo zřejmě k duplikaci/ztrátě poslané zprávy a zpráva je tedy poslána znovu. Podrobnější popis protokolu lze nalézt v [9], [21] nebo v [24].

Na obrázcích 7.4 a 7.5 jsou zobrazeny stavové diagramy všech komponent. Připomeňme, že událost `#timeout` značí interní přechod (δ_{INT}) atomického DEVS modelu a ještě před ním se provede případný výstup modelu (λ). Atomická komponenta Trans obsahuje slot `randNumber`, který udržuje náhodné číslo v rozsahu $< 0, 1000 >$ [2]. Náhodným číslem je explicitně řešen nedeterminismus při přeposlání/ztrátě zprávy. Náhodné číslo je inicializováno ve vstupní akci stavu `send0`, resp. `send1`. Tím je zaručeno, že proběhne opětovná inicializace před každým přechodem (a tedy i před jeho strážní podmínkou) vedoucím ze stavu `send0`/`send1`. Tato inicializace je nutná, neboť před každým přístupem ke slotu (funkce `(get-slot)`) musí být alespoň jednou tento slot nastaven (funkce `(set-slot)`).

Po vytvoření kompletního modelu lze model exportovat do formátu DEVSML výběrem příslušné volby v menu vnějšího spojovaného modelu. Výsledný soubor obsahuje platformně nezávislou reprezentaci modelu ABP.

Pokud máme model popsaný jazykem DEVSML (nechť se soubor jmenuje `ABP.devsml`), můžeme přistoupit k transformaci modelu do prostředí Adevs. Pro tento účel byl vytvořen skript pro interpretr Bash. Převod je tedy přímočarý:

```
$ ./devsml2adevs.sh ABP.devsml
```

Skript pouze volá XSLT procesor s příslušnými šablonami. Výsledkem je adresář `ABP`, který obsahuje všechny potřebné soubory pro překlad včetně `Makefile`. Následuje tedy překlad:

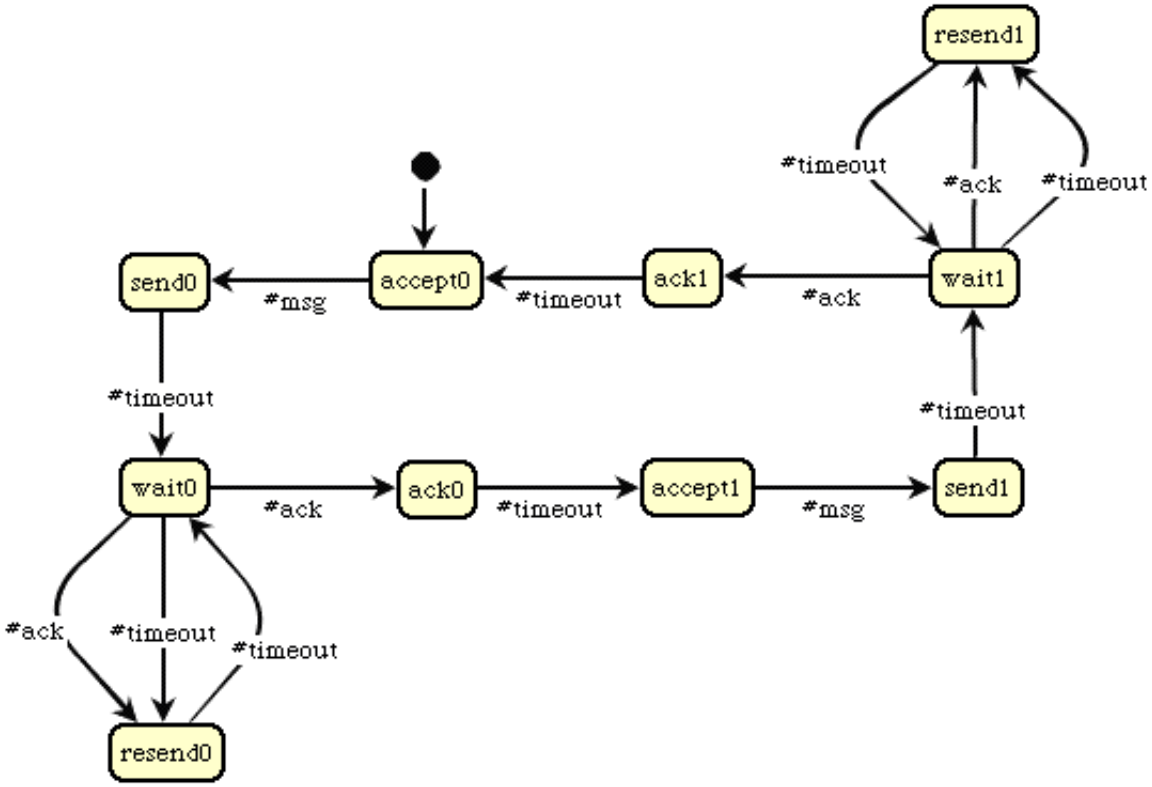
```
$ cd ./ABP
$ make
```

Tímto vznikne spustitelný program s názvem `ABP`. Zbývá tedy spustit samotnou simulaci:

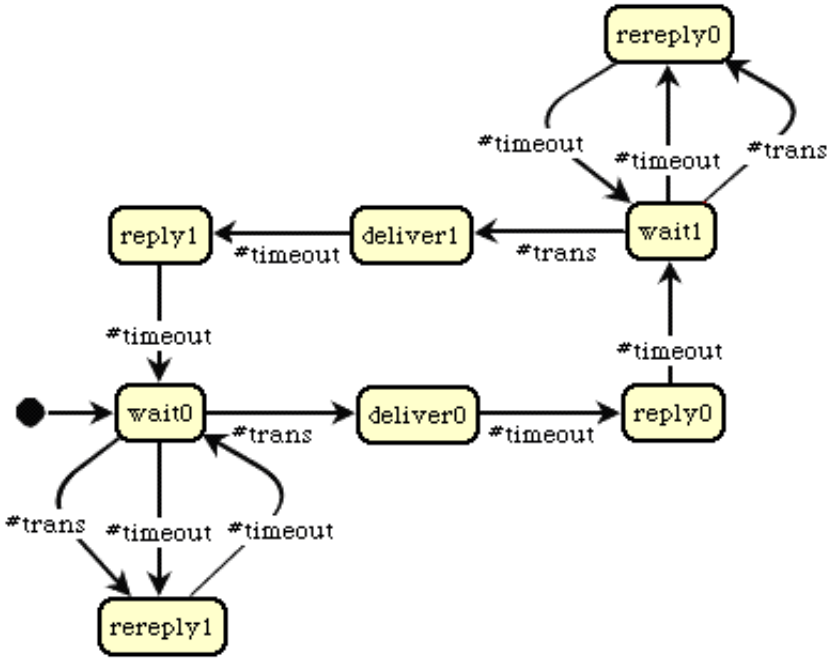
```
$ ./ABP
```

Jen pro zajímavost, počet řádků kódu modelu popsaného skriptem ve SmallDEVS je zhruba 200, počet řádků DEVSML popisu je 240 a počet řádků kódu výsledného modelu v C/C++ pro prostředí Adevs je zhruba 4200.

[2] Takto rozsáhlý interval byl vybrán z důvodu experimentálně zjištěného rychlejšího generování náhodných čísel Scheme funkcí `random` knihovny `LispKit`.

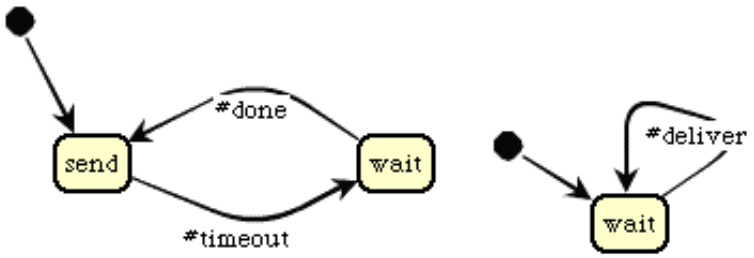


(a) Sender



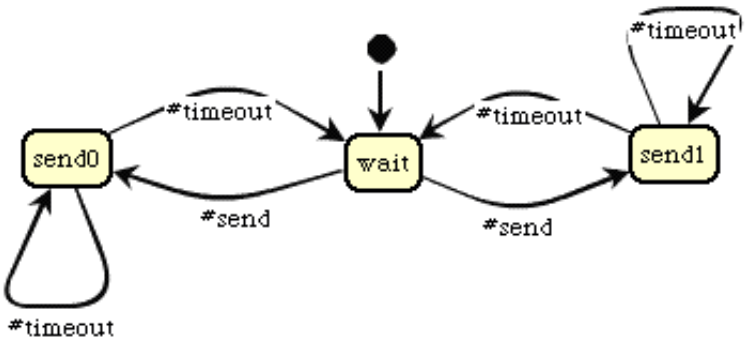
(b) Receiver

Obrázek 7.4: Atomické DEVS komponenty spojovaného modelu ABP: Sender, Receiver.

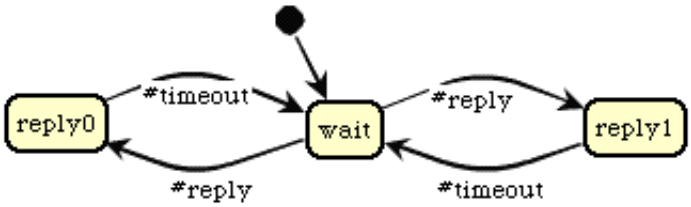


(a) Gen

(b) Acceptor



(c) Trans



(d) Ack

Obrázek 7.5: Atomické DEVS komponenty spojovaného modelu ABP: Gen, Acceptor, Trans, Ack.

Kapitola 8

Implementace

Nástroj pro tvorbu stavových diagramů je implementován v jazyce Smalltalk, konkrétně pak v dialektu Squeak. Celá aplikace je začleněna do systému SmallDEVS. Převod jazyka DEVSML do prostředí Adevs je prováděn prostřednictvím XSLT šablon.

Stěžejním prvkem pro běh vytvořeného automatu v rámci systému SmallDEVS je rozhraní mezi DEVS simulátorem a stavovým automatem.

Systém SmallDEVS podporuje tvorbu modelů prototypově objektově orientovaným přístupem [17]. Oproti třídně objektově orientovanému přístupu tento přístup umožňuje mnohem větší flexibilitu při tvorbě modelů. Modely v systému SmallDEVS jsou vytvářeny jako prototypy, jejichž chování může být řízeno delegáty, kteří poskytují metody na požádání. Tuto funkcionalitu umožňuje rozšíření v podobě balíku `Prototypes`. Prototyp objektu obsahuje sloty a metody, které mohou být měněny za běhu.

Implementace rozhraní mezi DEVS simulátorem a stavovým automatem silně prototypově objektově orientovaný přístup využívá. Základní struktura je uvedena na obrázku 8.3. Základním pilířem komunikace mezi DEVS simulátorem a stavovým automatem je objekt `statechartDEVSTrait`. Ten, mimo jiné, obsahuje metody atomického DEVSu. Tyto metody jsou pevně dány a slouží jako prostředník komunikace mezi objektem `statechart` a atomickým DEVS modelem. Objekt `statechart` obsahuje chování vytvořeného automatu. Více o implementaci samotného stavového automatu pojednává následující sekce.

8.1 Implementace stavového automatu

Tato sekce nejprve stručně uvádí tři typické implementace stavových automatů. V poslední podsekci se zaměřuje na implementaci stavového automatu pomocí meta vzoru známého jako dědičnost chování (behavioral inheritance meta-pattern) [25].

8.1.1 Konstrukce pomocí vnořeného příkazu `switch`

Nejjednodušší a zřejmě nejznámější technikou implementace stavových automatů je implementace pomocí vnořeného příkazu `switch`. Tato technika používá pro implementaci stavového automatu vnořený příkaz `switch`, kde je na první úrovni tohoto příkazu rozlišen stav, obvykle reprezentován pomocí výčtového typu, a na druhé úrovni je rozlišen signál události.

Přestože je tato technika velice jednoduchá, není moc elegantní. S rostoucím počtem stavů se vnořený `switch` stává čím dál tím méně čitelným, navíc logika provádění přechodů a popis chování automatu jsou velice úzce propojeny.

8.1.2 Stavová tabulka

Další velice známou technikou implementace stavových automatů je použití stavové (také přechodové) tabulky. Řádky tabulky označují stavy, sloupce potom signály událostí. Jednotlivé buňky tabulky obsahují dvojice (akce, následující stav).

	CHAR	**{**	**}**
code	putchar(c), code	nop(), comment	
comment	nop(), comment	nop(), comment	nop(), code

Tabulka 8.1: Stavová tabulka automatu, který vynechává komentáře ze zdrojových programů v jazyce Pascal.

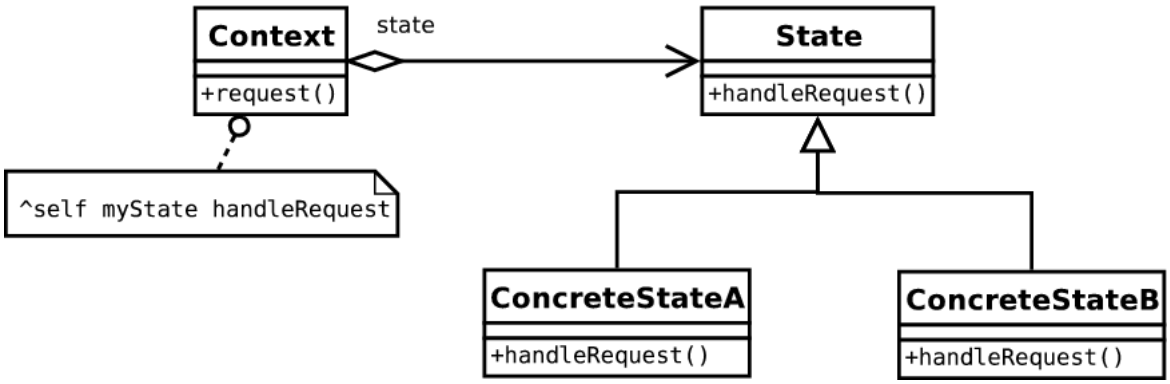
Jednoduchý automat vyjádřený pomocí stavové tabulky ukazuje tabulka 8.1. Automat vynechává komentáře ze zdrojových programů v jazyce Pascal. Jako počáteční stav je brán stav uvedený na prvním řádku tabulky. Celý automat sestává pouze ze dvou stavů a tří signálů událostí.

Prázdné buňky tabulky vyjadřují nedefinované kombinace stav-signál. Obvykle automat při dosažení takových kombinací signály ignoruje a setrvává ve stejném stavu, je ale možné zvolit jiný přístup, například podat zprávu o chybě.

Technika reprezentace stavového automatu pomocí stavové tabulky již odstraňuje problém propojení logiky provádění přechodů a chování automatu. Nicméně, nevýhoda spočívá ve velmi nízké flexibilitě. Při změně topologie automatu je třeba přeskupit (často značnou) část tabulky, což obvykle vede k chybám.

8.1.3 Objektově orientovaný návrhový vzor Stav

Návrhový vzor Stav [14] využívá pro implementaci stavového automatu dědičnosti a polymorfismu. Základní struktura návrhového vzoru Stav je uvedena na obrázku 8.1. Konkrétní stavy jsou tvořeny děděním od abstraktní třídy `State`. Přidání nové události znamená přidat virtuální metodu do třídy `State`. Třída `Context` deleguje všechny události objektu současného stavu, na který je odkazováno pomocí atributu `myState`. Přechod do nového stavu je proveden přiřazením objektu následujícího stavu do atributu `myState`.



Obrázek 8.1: Návrhový vzor Stav.

Výhoda tohoto přístupu spočívá v oddělení implementace jednotlivých stavů. Díky mechanizmu pozdní vazby je posílání událostí efektivní. Provedení přechodu je rovněž velice efektivní (přiřazení jednoho ukazatele). Není třeba provádět výčet událostí a stavů.

Pro potřeby vytváření stavového automatu, kdy stavy a přechody tvoří uživatel postupně v průběhu interakce s nástrojem pro tvorbu stavových diagramů, nebo pro účely automatického dynamického dotváření diagramu za běhu simulace tento přístup není příliš vhodný.

8.1.4 Meta vzor dědičnost chování

Cíl meta vzoru dědičnosti chování je poskytnout obecný procesor událostí, který může být využit s jakýmkoli mechanizmem zasílání událostí a řazení událostí do front. Vzor poskytuje základní stavební kameny pro efektivní konstrukci všech vlastností StateChartu. [25] Navíc lze tento vzor (s menšími úpravami) použít i v prototypově objektově-orientované implementaci. Je tedy ideálním kandidátem pro interaktivní tvorbu stavových diagramů, dokonce i pro automatické dynamické dotváření diagramu za běhu. Není tedy náhoda, že tato práce tento vzor pro implementaci stavového automatu využívá. Implementace meta vzoru dědičnosti chování je součástí knihovny `Connectors`. Tato práce implementaci využívá a modifikuje ji pro své účely.

Struktura meta vzoru dědičnosti chování je uvedena na obrázku 8.2. Hlavní abstraktní třída `QHsm` poskytuje metody `init()` pro inicializaci stavového automatu, `dispatch()` pro zasílání událostí a `tran()` pro provedení přechodů. Odkaz na aktivní stav je udržován v atributu `myState`. Atribut `mySource` udržuje odkaz na zdrojový stav současného přechodu (v případě hierarchických stavových diagramů, kdy může být přechod děděn od nadřazeného stavu se `myState` a `mySource` mohou lišit).

Meta vzor dědičnosti chování se na první pohled podobá návrhovému vzoru řetěz odpovědnosti [14], kde je požadavek (událost) vyslán směrem dolů po řetězu stavové hierarchie, ve kterém má více než jeden stav možnost ho zachytit. Odlišnost je ale v tom, že řetěz odpovědnosti posílá požadavek napříč různými objekty, zatímco vzor dědičnosti chování posílá událost v rámci jednoho objektu.

8.2 Implementace rozhraní mezi DEVS komponentou a stavovým automatem

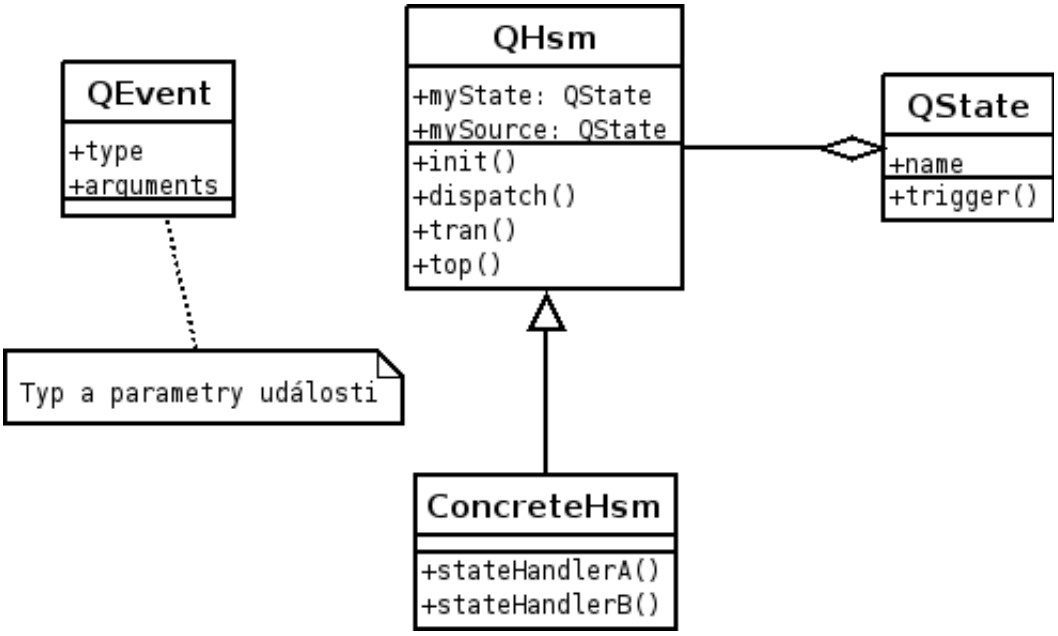
Metody atomických DEVS komponent sloužících jako rozhraní mezi stavovým automatem a atomickou DEVS komponentou jsou navrženy takto:

δ_{int} Vyvolá událost `timeout` stavového automatu.

δ_{ext} Nastaví sloty `elapsed` (čas uplynulý od poslední události) a `inPort` (asociace s klíčem, který odpovídá názvu portu, na který událost přišla a hodnotou, která na daný port dorazila) objektu stavového automatu na patřičné hodnoty, vytvoří objekt události `QEvent` a vyšle jej do stavového automatu.

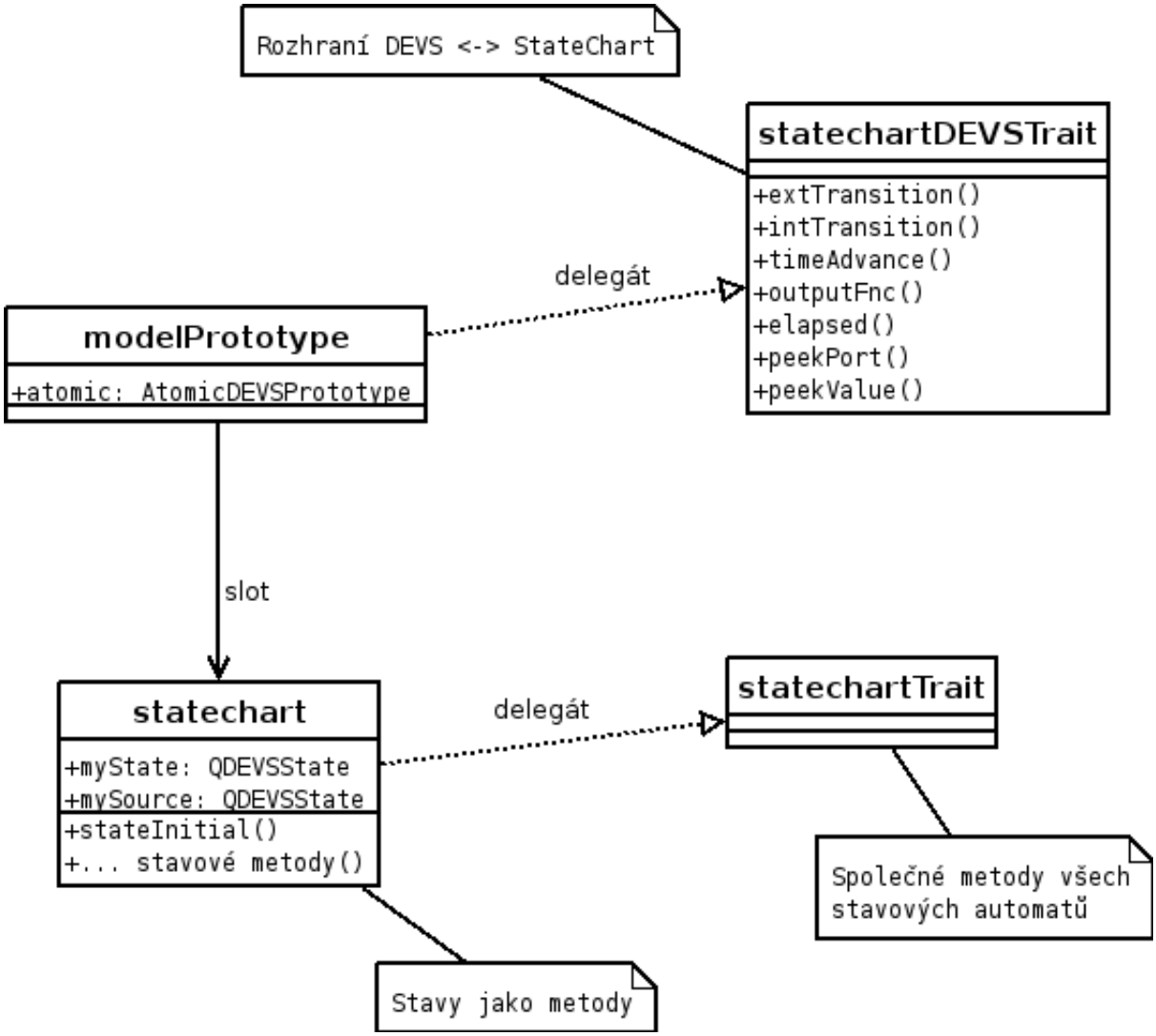
ta Vrací obsah parametru `wait` současného stavu.

λ Předá obsah parametru `output` současného stavu na výstupní port atomické komponenty specifikovaný parametrem `outputPort` současného stavu.



Obrázek 8.2: Meta vzor dědičnost chování.

Všimněme si slotů stavového diagramu (objektu `statechart`) `elapsed` a `inPort`. Sloty jsou vždy před vyvolání externí události v objektu stavového diagramu řádně nastaveny. Nachází se zde protože objekt stavového diagramu žádný odkaz zpět na atomickou komponentu neudržuje. Takovýto odkaz by zaváděl kruhovou závislost, která je v tomto případě zcela zbytečná. Akce/stráže stavového diagramu hodnoty těchto slotů využívají. Přistupují k nim ale nepřímo pomocí speciálních metod (viz. sekce 5.3 a 5.4).



Obrázek 8.3: Implementace rozhraní mezi DEVS komponentou a stavovým automatem.

Kapitola 9

Závěr

Díky popisu atomických DEVS komponent pomocí stavových diagramů jsme schopni nejen atraktivně modelovat složité reaktivní systémy, ale také převádět modely do platformně nezávislé reprezentace. Tato práce navrhla a implementovala platformně nezávislou reprezentaci DEVS modelů. Podařilo se rozšířit jazyk DEVSML o rozšířený jazyk SCXML. DEVSML tak představuje platformně nezávislý jazyk.

9.1 Dosažené cíle

Byl vytvořen vizuální nástroj pro tvorbu stavových diagramů, který byl začleněn do modelovacího/simulačního prostředí SmallDEVS.

Pro modely vytvořené v prostředí SmallDEVS a popsané pomocí stavových diagramů a jazyka Scheme byl implementován generátor, který modely převádí do platformně nezávislého jazyka DEVSML. Dále byl navržen a implementován transformátor modelů popsaných jazykem DEVSML do prostředí Adevs. Transformátor pracuje zcela automaticky bez jakéhokoli ručního zásahu. Výsledné modely pro prostředí Adevs jsou tedy zcela funkční a po překladu připraveny k použití.

9.2 Možná rozšíření

Hlavní rozšíření lze provést použitím hierarchických stavových diagramů pro popis atomických komponent. Přidáním ortogonálních stavů a broadcastingu by stavové diagramy dosáhly deskriptivní síly StateChartů. Dodejme, že vnitřní implementace stavových diagramů použitých v této práci jak v systému SmallDEVS, tak v systému Adevs použití hierarchických stavů umožňuje. Také jazyk SCXML hierarchické stavy podporuje. Toto rozšíření by tedy v praxi znamenalo upravit grafické uživatelské rozhraní nástroje pro tvorbu stavových diagramů, generátor SCXML popisu stavového diagramu a XSLT šablonu pro transformaci stavového diagramu popsaného jazykem SCXML na C++ kód pro knihovnu QHsm.

Dalším pokrokem by byl převod modelu v DEVSML do jiných prostředí. Jako příklad uveďme prostředí PyDEVS, DEVSJava, nebo zpět do prostředí SmallDEVS. Převod do jiných prostředí znamená vytvořit nové XSLT šablony, které transformaci specifikují. Také je třeba nové prostředí připravit pro použití stavových automatů a interpretaci jazyka Scheme.

V rámci kompatibility popisu akcí/stráží stavových diagramů jazykem Scheme napříč

prostředími by bylo vhodné vytvořit společnou sadu knihoven. Stávající implementace začleňování nových knihoven podporuje. Postup je přímočarý a byl v této práci popsán.

Protože knihovna Adevs (i systém SmallDEVS) podporuje práci s variabilní strukturou DEVS modelu, mohlo by být zajímavé tuto podporu začlenit i do DEVSML a umožnit tak generování modelů popsaných formalismem DSDEVS [30] (Dynamic Structure DEVS).

Na závěr uveďme ještě poslední rozšíření, které se týká editace akcí/stráží stavového diagramu za běhu simulace v systému Adevs. Současná implementace dovoluje akce editovat přepisováním souborů, které jsou před každou interpretací znovu načítány a spouštěny. Lze si jistě představit lepší způsob jak po stránce technické (místo souborů použít například sdílenou paměť, nebo jiné techniky), tak po stránce uživatelské (speciální editor/terminál pro tuto činnost). Mohlo by být dosaženo kompletní kontroly nad celým modelem za jeho běhu (včetně například vyvolávání externích událostí, čímž by se dala, mimo jiné, obejít absence této funkčnosti v systému Adevs). Výsledkem by byla možnost tvorby virtuálních programovatelných strojů, které by byly efektivně prováděny.

Literatura

[1] Adevs: A Discrete EVent System simulator. `http://www.ornl.gov/~1qn/adevs/index.html`.

[2] The Boost Statechart library. `http://www.boost.org/doc/libs/1_37_0/libs/statechart/doc/index.html`.

[3] Guile: Project GNU's extension language. `http://www.gnu.org/software/guile/guile.html`.

[4] LispKit: building custom Lisp interpreter in Squeak. `http://www.zogotounga.net/comp/squeak/lispkit.htm`.

[5] QHsm: Quantum Hierarchical State Machines. `http://www.state-machine.com`.

[6] R5RS: Revised^5 Report on the Algorithmic Language Scheme. `http://schemers.org/Documents/Standards/R5RS/`.

[7] StateChart XML (SCXML). `http://www.w3.org/TR/scxml/`.

[8] Badros, G. J.: JavaML: a markup language for Java source code. *Comput. Netw.*, 2000: s. 159–177, ISSN 1389-1286, doi:http://dx.doi.org/10.1016/S1389-1286(00)00037-2.

[9] Bartlett, K. A.; Scantlebury, R. A.; Wilkinson, P. T.: A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 1969: s. 260–261, ISSN 0001-0782, doi:http://doi.acm.org/10.1145/362946.362970.

[10] Breen, M.: Statecharts: Some Critical Observations. 2004. URL `http://citeseer.ist.psu.edu/740806.html`

[11] Concepcion, A. I.; Zeigler, B. P.: DEVS Formalism: A Framework for Hierarchical Model Development. 1988.

[12] Fishwick, P. A.: Simulation Model Design and Execution: Building Digital Worlds. 1995.

[13] Futó, I.; Gergely, T.: TS-PROLOG, a logic simulation language. 1986.

[14] Gamma, E.; Helm, R.; Johnson, R.; aj.: Design Patterns: Elements of Resusable Object-Oriented Software. 1995.

[15] Harel, D.: On the Formal Semantics of Statecharts. 1987.

[16] Harel, D.: Statecharts: A visual formalism for complex systems. 1987.

[17] Janoušek, V.: On the Prototype-Based Object Orientation in Modeling and Simulation. In *Proceedings of Advanced Simulation of Systems 2006*, 2006, ISBN 80-86840-26-3, str. 6.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8173

[18] Janoušek, V.; Polášek, P.; Slavíček, P.: Towards DEVS Meta Language. In *ISC 2006 Proceedings*, 2006, ISBN 90-77381-26-0, s. 69–73.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8109

[19] Klir, G. J.: Architecture of systems complexity. 1985.

[20] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. 1960.

[21] Milner, R.: Communication and concurrency. 1989.

[22] Object Management Group, I.: OMG Unified Modeling Language Specification v1.4. 2001.

[23] Ören, T. I.; c, K. Z. A.: Architecture of MAGEST: A Knowledge-based Modelling and Simulation System. 1984.

[24] Ostroff, J. S.: Temporal logic for real time systems. 1989.

[25] Samek, M.: Practical Statecharts in C/C++: Quantum Programming for Embedded Systems. 2002.

[26] Schulz, S.; Ewing, T.; Rozenblit, J.: Discrete Event System Specification (DEVS) and StateMate StateCharts Equivalence for Embedded Systems Modeling. In *Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the*, 2000, ISBN 0-7695-0604-6, s. 308–316.

[27] Sussman, G. J.; Steele Jr., G. L.: SCHEME: an interpreter for extended lambda calculus. 1975.

[28] Zeigler, B. P.: Theory of modeling and simulation. 1976.

[29] Zeigler, B. P.: Object-Oriented Simulation with Hierarchical, Modular Models. 1995.

[30] Zeigler, B. P.; Praehofer, H.; Kim, T. G.: Theory of Modeling and Simulation; 2nd edition. 2000, iSBN 978-0127784557.

Dodatek A

Spojovaná komponenta systému Alternating Bit Protocol popsaná skriptem.

```
"---Sender--------------------------------"

mSender := Statechart newWithSchemeInterpreter.

sAccept0 := mSender addSimpleState: {#name -> 'accept0'}.
sSend0 := mSender addSimpleState: {
    #name -> 'send0'. #wait -> '0'. #output -> '0'.
       #outputEvent -> 'send'}.
sWait0 := mSender addSimpleState: {#name -> 'wait0'. #wait
    -> '5'}.
sResend0 := mSender addSimpleState: {
    #name -> 'resend0'. #wait -> '0'. #output -> '0'.
       #outputEvent -> 'send'}.
sAck0 := mSender addSimpleState: {
    #name -> 'ack0'. #wait -> '0'. #output -> '#t'.
       #outputEvent -> 'done'}.
sAccept1 := mSender addSimpleState: {#name -> 'accept1'}.
sSend1 := mSender addSimpleState: {
    #name -> 'send1'. #wait -> '0'. #output -> '1'.
       #outputEvent -> 'send'}.
sWait1 := mSender addSimpleState: {#name -> 'wait1'. #wait
    -> '5'}.
sResend1 := mSender addSimpleState: {
    #name -> 'resend1'. #wait -> '0'. #output -> '1'.
       #outputEvent -> 'send'}.
sAck1 := mSender addSimpleState: {
    #name -> 'ack1'. #wait -> '0'. #output -> '#t'.
       #outputEvent -> 'done'}.
```

```
mSender addTransition: {#from -> (mSender addStartState).
   #to -> sAccept0}.
mSender addTransition: {#from -> sAccept0. #to -> sSend0.
   #event -> 'msg'}.
mSender addTransition: {#from -> sSend0. #to -> sWait0.
   #event -> 'timeout'}.
mSender addTransition: {#from -> sWait0. #to -> sResend0.
   #event -> 'timeout'}.
mSender addTransition: {
    #from -> sWait0. #to -> sResend0. #event -> 'ack'.
       #guard -> '(equal? (peek-value) 1)'}.
mSender addTransition: {
    #from -> sWait0. #to -> sAck0. #event -> 'ack'. #guard
        -> '(equal? (peek-value) 0)'}.
mSender addTransition: {#from -> sResend0. #to -> sWait0.
   #event -> 'timeout'}.
mSender addTransition: {#from -> sAck0. #to -> sAccept1.
   #event -> 'timeout'}.
mSender addTransition: {#from -> sAccept1. #to -> sSend1.
   #event -> 'msg'}.
mSender addTransition: {#from -> sSend1. #to -> sWait1.
   #event -> 'timeout'}.
mSender addTransition: {#from -> sWait1. #to -> sResend1.
   #event -> 'timeout'}.
mSender addTransition: {
    #from -> sWait1. #to -> sResend1. #event -> 'ack'.
       #guard -> '(equal? (peek-value) 0)'}.
mSender addTransition: {
    #from -> sWait1. #to -> sAck1. #event -> 'ack'. #guard
        -> '(equal? (peek-value) 1)'}.
mSender addTransition: {#from -> sResend1. #to -> sWait1.
   #event -> 'timeout'}.
mSender addTransition: {#from -> sAck1. #to -> sAccept0.
   #event -> 'timeout'}.

aSender := AtomicDEVSPrototype new.
aSender addInputPorts: {#msg. #ack}.
aSender addOutputPorts: {#send. #done}.
aSender removeDelegate: 'defaultTrait'.
aSender addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aSender addSlot: 'statechart' withValue: mSender.

"---Receiver--------------------------------"

mReceiver := Statechart newWithSchemeInterpreter.
```

```
sWait0 := mReceiver addSimpleState: {#name -> 'wait0'.
   #wait -> '10'}.
sRereply1 := mReceiver addSimpleState: {
    #name -> 'rereply1'. #wait -> '0'. #output -> '1'.
       #outputEvent -> 'reply'}.
sDeliver0 := mReceiver addSimpleState: {
    #name -> 'deliver0'. #wait -> '0'. #output -> '#t'.
       #outputEvent -> 'deliver'}.
sReply0 := mReceiver addSimpleState: {
    #name -> 'reply0'. #wait -> '0'. #output -> '0'.
       #outputEvent -> 'reply'}.
sWait1 := mReceiver addSimpleState: {#name -> 'wait1'.
   #wait -> '10'}.
sRereply0 := mReceiver addSimpleState: {
    #name -> 'rereply0'. #wait -> '0'. #output -> '0'.
       #outputEvent -> 'reply'}.
sDeliver1 := mReceiver addSimpleState: {
    #name -> 'deliver1'. #wait -> '0'. #output -> '#t'.
       #outputEvent -> 'deliver'}.
sReply1 := mReceiver addSimpleState: {
    #name -> 'reply1'. #wait -> '0'. #output -> '1'.
       #outputEvent -> 'reply'}.

mReceiver addTransition: {#from -> (mReceiver
   addStartState). #to -> sWait0}.
mReceiver addTransition: {#from -> sWait0. #to ->
   sRereply1. #event -> 'timeout'}.
mReceiver addTransition: {
    #from -> sWait0. #to -> sRereply1. #event -> 'trans'.
       #guard -> '(equal? (peek-value) 1)'}.
mReceiver addTransition: {#from -> sRereply1. #to ->
   sWait0. #event -> 'timeout'}.
mReceiver addTransition: {
    #from -> sWait0. #to -> sDeliver0. #event -> 'trans'.
       #guard -> '(equal? (peek-value) 0)'}.
mReceiver addTransition: {#from -> sDeliver0. #to ->
   sReply0. #event -> 'timeout'}.
mReceiver addTransition: {#from -> sReply0. #to -> sWait1.
    #event -> 'timeout'}.
mReceiver addTransition: {#from -> sWait1. #to ->
   sRereply0. #event -> 'timeout'}.
mReceiver addTransition: {
    #from -> sWait1. #to -> sRereply0. #event -> 'trans'.
       #guard -> '(equal? (peek-value) 0)'}.
mReceiver addTransition: {#from -> sRereply0. #to ->
   sWait1. #event -> 'timeout'}.
mReceiver addTransition: {
```

```
    #from -> sWait1. #to -> sDeliver1. #event -> 'trans'.
       #guard -> '(equal? (peek-value) 1)'}.
mReceiver addTransition: {#from -> sDeliver1. #to ->
   sReply1. #event -> 'timeout'}.
mReceiver addTransition: {#from -> sReply1. #to -> sWait0.
    #event -> 'timeout'}.

aReceiver := AtomicDEVSPrototype new.
aReceiver addInputPorts: {#trans}.
aReceiver addOutputPorts: {#reply. #deliver}.
aReceiver removeDelegate: 'defaultTrait'.
aReceiver addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aReceiver addSlot: 'statechart' withValue: mReceiver.

"---Trans-------------------------------------"

mTrans := Statechart newWithSchemeInterpreter.
mTrans addSlot: 'randNumber' withValue: 0.

sSend1 := mTrans addSimpleState: {
    #name -> 'send1'. #wait -> '3'. #output -> '1'.
       #outputEvent -> 'trans'.
    #entry -> '(set-slot "randNumber" (random 1000))'}.
sWait := mTrans addSimpleState: {#name -> 'wait'}.
sSend0 := mTrans addSimpleState: {
    #name -> 'send0'. #wait -> '3'. #output -> '0'.
       #outputEvent -> 'trans'.
    #entry -> '(set-slot "randNumber" (random 1000))'}.

mTrans addTransition: {#from -> (mTrans addStartState).
   #to -> sWait}.
mTrans addTransition: {
    #from -> sWait. #to -> sSend1. #event -> 'send'.
       #guard -> '(equal? (peek-value) 1)'}.
mTrans addTransition: {
    #from -> sWait. #to -> sSend0. #event -> 'send'.
       #guard -> '(equal? (peek-value) 0)'}.
mTrans addTransition: {
    #from -> sSend1. #to -> sWait. #event -> 'timeout'.
    #guard -> '(> (get-slot "randNumber") 500)'}.
mTrans addTransition: {
    #from -> sSend1. #to -> sSend1. #event -> 'timeout'.
    #guard -> '(<= (get-slot "randNumber") 500)'}.
mTrans addTransition: {
    #from -> sSend0. #to -> sWait. #event -> 'timeout'.
    #guard -> '(> (get-slot "randNumber") 500)'}.
mTrans addTransition: {
```

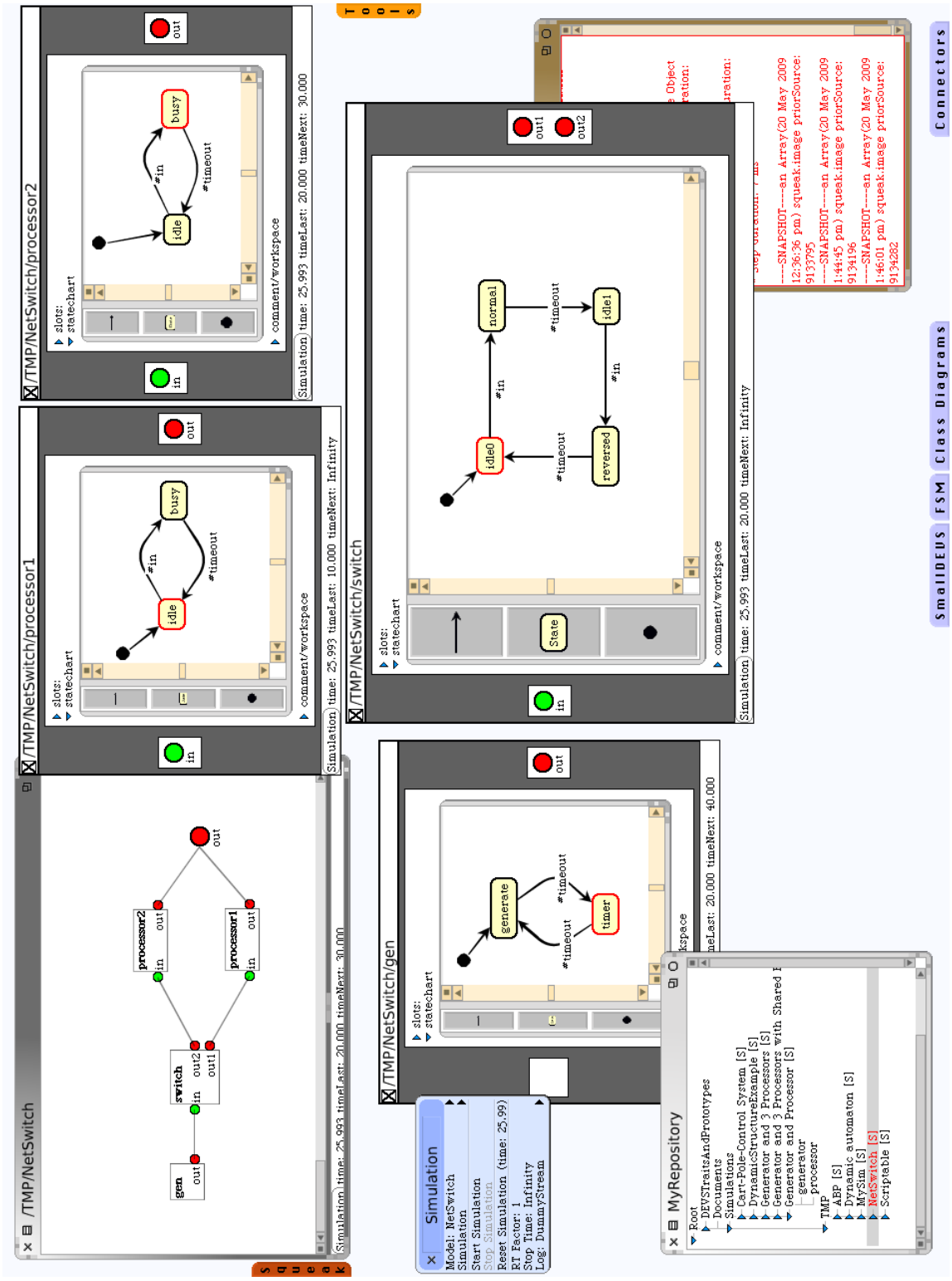
```
    #from -> sSend0. #to -> sSend0. #event -> 'timeout'.
    #guard -> '(<= (get-slot "randNumber") 500)'}.

aTrans := AtomicDEVSPrototype new.
aTrans addInputPorts: {#send}.
aTrans addOutputPorts: {#trans}.
aTrans removeDelegate: 'defaultTrait'.
aTrans addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aTrans addSlot: 'statechart' withValue: mTrans.

"---Ack-----------------------------------------"

mAck := Statechart newWithSchemeInterpreter.

sReply1 := mAck addSimpleState: {
    #name -> 'reply1'. #wait -> '2'. #output -> '1'.
       #outputEvent -> 'ack'}.
sWait := mAck addSimpleState: {#name -> 'wait'}.
sReply0 := mAck addSimpleState: {
    #name -> 'reply0'. #wait -> '2'. #output -> '0'.
       #outputEvent -> 'ack'}.

mAck addTransition: {#from -> (mAck addStartState). #to ->
    sWait}.
mAck addTransition: {
    #from -> sWait. #to -> sReply1. #event -> 'reply'.
       #guard -> '(equal? (peek-value) 1)'}.
mAck addTransition: {
    #from -> sWait. #to -> sReply0. #event -> 'reply'.
       #guard -> '(equal? (peek-value) 0)'}.
mAck addTransition: {#from -> sReply1. #to -> sWait.
   #event -> 'timeout'}.
mAck addTransition: {#from -> sReply0. #to -> sWait.
   #event -> 'timeout'}.

aAck := AtomicDEVSPrototype new.
aAck addInputPorts: {#reply}.
aAck addOutputPorts: {#ack}.
aAck removeDelegate: 'defaultTrait'.
aAck addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aAck addSlot: 'statechart' withValue: mAck.

"---Gen-----------------------------------------"

mGen := Statechart newWithSchemeInterpreter.
```

```
sSend := mGen addSimpleState: {
    #name -> 'send'. #wait -> '0'. #output -> ''''''message
       ''''''. #outputEvent -> 'msg'}.
sWait := mGen addSimpleState: {#name -> 'wait'}.

mGen addTransition: {#from -> (mGen addStartState). #to ->
    sSend}.
mGen addTransition: {#from -> sSend. #to -> sWait. #event
   -> 'timeout'}.
mGen addTransition: {#from -> sWait. #to -> sSend. #event
   -> 'done'}.

aGen := AtomicDEVSPrototype new.
aGen addInputPorts: {#done}.
aGen addOutputPorts: {#msg}.
aGen removeDelegate: 'defaultTrait'.
aGen addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aGen addSlot: 'statechart' withValue: mGen.

"---Acceptor-----------------------------------"

mAcceptor := Statechart newWithSchemeInterpreter.

sWait := mAcceptor addSimpleState: {#name -> 'wait'}.

mAcceptor addTransition: {#from -> (mAcceptor
   addStartState). #to -> sWait}.
mAcceptor addTransition: {#from -> sWait. #to -> sWait.
   #event -> 'deliver'}.

aAcceptor := AtomicDEVSPrototype new.
aAcceptor addInputPorts: {#deliver}.
aAcceptor removeDelegate: 'defaultTrait'.
aAcceptor addDelegate: 'statechartDEVSTrait' withValue:
   StatechartDEVSTrait new.
aAcceptor addSlot: 'statechart' withValue: mAcceptor.

"-----------------------------------------------
Overall System Model
-----------------------------------------------"

system := CoupledDEVSPrototype new.
system addComponents: {#gen -> aGen. #sender -> aSender.
   #trans -> aTrans.
    #receiver -> aReceiver. #acceptor -> aAcceptor. #ack
       -> aAck}.
```

```
system connectSubModelNamed: #gen port: #msg
   toSubModelNamed: #sender port: #msg.
system connectSubModelNamed: #sender port: #done
   toSubModelNamed: #gen port: #done.
system connectSubModelNamed: #sender port: #send
   toSubModelNamed: #trans port: #send.
system connectSubModelNamed: #trans port: #trans
   toSubModelNamed: #receiver port: #trans.
system connectSubModelNamed: #receiver port: #deliver
   toSubModelNamed: #acceptor port: #deliver.
system connectSubModelNamed: #receiver port: #reply
   toSubModelNamed: #ack port: #reply.
system connectSubModelNamed: #ack port: #ack
   toSubModelNamed: #sender port: #ack.
```

Dodatek B

Snímek obrazovky probíhající simulace modelu Síťového přepínače



Obrázek B.1: Kompletní pohled na modelovací nástroj.

Dodatek C

Obsah CD

```
CD/
|-- README
|-- devsml2adevs
|   |-- adevs.statechart.lisp.xsl
|   |-- devsml2adevs.sh
|   |-- devsml2devsml-debug.sh
|   |-- devsml2devsml-debug.xsl
|   |-- examples
|   |   |-- ABP.debug.devsml
|   |   |-- ABP.devsml
|   |   |-- HierarchicalNetSwitch.debug.devsml
|   |   |-- HierarchicalNetSwitch.devsml
|   |   |-- NetSwitch.debug.devsml
|   |   |-- NetSwitch.devsml
|   |   `-- Scriptable.devsml
|   `-- scxml2qhsm.lisp.xsl
|-- install-libs.sh
|-- libs
|   |-- adevs
|   |   |-- INSTALL
|   |   `-- adevs-2.1.tar.gz
|   |-- guile
|   |   |-- INSTALL
|   |   |-- guile-1.8.6.tar.gz
|   |   |-- guile.pdf
|   |   `-- guile.txt
|   |-- qpcpp
|   |   |-- INSTALL
|   |   |-- qpcpp_4.0.04.zip
|   |   `-- qpcpp_man_4.0.04.chm
|   |-- squeak
|   |   |-- INSTALL
|   |   `-- Squeak-3.10-1.i686-pc-linux-gnu.tar.gz
|   `-- xslt2
|       |-- INSTALL
```

```
|       `-- xslt2-2.6.5-install.jar
|-- squeak-image
|   |-- SqueakV39.sources
|   |-- squeak.changes
|   `-- squeak.image
`-- thesis
```