VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta elektrotechniky a informatiky

Modelování a simulace

Doc. Ing. Zdena Rábová, CSc.
Prof. RNDr. Milan Češka, CSc.
Doc. Ing. Jaroslav Zendulka, CSc.
Dr. Ing. Petr Peringer
Ing. Vladimír Janoušek, Ph.D.

(draft 1 — 3. ledna 2005)

PŘEDMLUVA K NOVÉMU UPRAVENÉMU VYDÁNÍ

Text starých skript z roku 1992 byl převeden do sázecího systému LaTeX a bylo opraveno několik chyb. Prozatím nejsou v textu téměř žádné obrázky, protože nejdříve musí být upraven rozsah textu. Z původních 230 stran bude text skript redukován asi na 100–150 stran. Zcela vypuštěna bude referenční příručka knihovny SIMLIB, protože její aktuální verze bude dostupná na WWW.

Tento text je pracovní a neúplná verze. Zveřejněn je pouze ve formátu PDF, po vydání nové verze skript je jakékoli jeho použití zakázáno. Text lze využít pro potřeby výuky předmětu "Modelování a simulace" na FIT VUT v Brně.

Text neprošel jazykovou úpravou a mohou v něm být chyby. Sazba nebyla kontrolována.

TODO: změna symbolů které neuměl starý sázecí systém zpět na původní, zkrátit a přepracovat kapitoly "Úvod...", "Simulace čísl. sys.", "SIMLIB", "Petriho sítě", "Analytické řešení modelů", ...

P. Peringer

PŘEDMLUVA

Tato skripta jsou určena pro předmět modelování a simulace, navazující na znalosti, které studenti získali v matematice, zvláště numerické matematice, matematické statistice a v předmětech počítače a programování, programovací techniky, logické systémy a základy překladačů.

Předmětem skript jsou metody a prostředky modelování a simulace reálných nebo navrhovaných systémů na počítači. Při vytváření simulačního modelu předpokládáme jeho zápis v některém simulačním jazyce a dále existenci procesoru, jenž je schopen simulační program provádět. Z tohoto schématu vyplývá specifikace simulačních modelů. Proces jejich výstavby vychází z poznatků o modelovaném objektu a přes formulaci abstraktního modelu ústí v simulační program, reprezentovaný posloupností symbolů simulačního jazyka. Vytváření abstraktních modelů má poněkud jiný charakter, než je stanovení algoritmu řešení problému v obecném smyslu. Podobně i simulační jazyky, jejichž prvotní funkcí je usnadnění popisu modelovaného systému, se v základních prostředcích liší od algoritmických programovacích jazyků. Cílem těchto skript je postihnout právě tyto specifické oblasti problematiky vytváření simulačních modelů.

Konkrétní obsahová náplň skript je v mnoha směrech ovlivněna výzkumnou činností ústavu. Při výběru simulačních jazyků pro praktickou část výuky jsme se zaměřili na objektově orientovanou simulaci a využití jazyka C++, ve kterém jsme implementovali spojitou, diskrétní i kombinovanou část simulačního systému.

Čtenářům skript budeme vděčni za upozornění a připomínky k věcné i formální správnosti textu.

Brno, červenec 1992

autoři

Obsah

Kapitola 1

Úvod — modelování systémů na počítačích

Pod pojmem modelování

rozumíme cílevědomou činnost, která slouží k získávání informací o jednom systému prostřednictvím jiného systému — modelu. O pojmu systém pojednáme podrobněji dále. Zpočátku pouze uvedeme, že systém budeme chápat jako soubor elementárních částí, prvků systému, jež mají mezi sebou určité vazby. Mezi dvěma systémy může existovat jistá podobnost. Poněvadž je model také systémem, využíváme této podobnosti při modelování. Význam modelování spočívá v tom, že umožňuje ekonomické studium systémů. Je výhodnější, rychlejší a často jedině možné získávat informace o systémech experimentováním na jejich modelech, než na originálech. V tomto smyslu náleží do modelování výstavba všech modelů, fyzikálních i matematických, statických i dynamických.

Je-li modelovaný systém jednoduchý, nebo můžeme-li formulovat tak zjednodušující předpoklady, aby byl model řešitelný analyticky, popíšeme chování systému matematickými vztahy a hledané veličiny stanovujeme matematickými prostředky. Výsledky získáváme ve formě funkčních vztahů, ve kterých se jako proměnné vyskytují parametry modelu. Řešení konkrétního modelu získáme dosazením konkrétních hodnot do funkčních vztahů. Výsledné hodnoty jsou tedy funkcí jednoho, či více obecných parametrů. Hlavní předností analytického řešení je menší časová náročnost řešení matematického modelu. Jde však o modely jednoduché nebo podstatně zjednodušené.

V současné době je však stále aktuálnějším problémem analýza složitých systémů, jejichž specifickými vlastnostmi jsou velká rozsáhlost, neúplnost daných informací, kvalitativní charakter parametrů, velká dynamičnost probíhajících procesů a složitý charakter vztahů mezi prvky systému. Komplexní analýzu těchto systémů umožnil teprve rozvoj číslicových počítačů, které jsou dnes nejperspektivnějším prostředkem pro studium složitých dynamických systémů prostřednictvím jejich modelů.

Metody modelování systémů na počítačích využíváme zvláště v těchto případech:

- neexistuje-li úplná matematická formulace problému nebo nejsou-li známé analytické metody řešení matematického modelu;
- vyžadují-li analytické metody tak zjednodušující předpoklady, že je nelze pro daný model přijmout;

- jsou-li analytické metody dostupné pouze teoreticky a praktické řešení by bylo tak obtížné a dlouhé, že je metoda modelování problému na počítači jednodušší cestou jeho řešení;
- je-li žádoucí modelovat historii procesu v určitém časovém intervalu za účelem odhadu některých parametrů;
- je-li modelování na počítači jedinou možností získání výsledků v důsledku obtížnosti provádění experimentů ve skutečném prostředí;
- potřebujeme-li pro pozorování systému měnit časové měřítko – modelování systému na počítači umožňuje urychlování nebo zpomalování příslušných dějů.

Proces modelování systémů na počítačích můžeme velmi zjednodušeně rozdělit do tří základních etap:

(1) Formování účelového a zjednodušeného popisu zkoumaného systému – *vytvoření abstraktního modelu.*

(2) Zápis abstraktního modelu formou programu – *vytvoření simulačního modelu.*

(3) Experimentování s reprezentací simulačního modelu na počítači – *simulace.*

Než budeme tyto etapy stručně charakterizovat, vymezíme ještě tři základní časové pojmy:

a) *Reálný čas*, ve kterém probíhá skutečný děj v reálném systému.

b) *Modelový čas*, jenž tvoří časovou osu modelu; může být reálný, zrychlený nebo zpomalený.

c) *Strojový čas*, který představuje čas spotřebovaný na výpočet programu; jeho délka závisí na složitosti modelovaného systému a na vlastnostech vytvořeného programu; není však závislá na hodnotách, kterých nabývá modelový čas.

1.1 Vytvoření abstraktního modelu

Poněvadž nedovedeme postihnout reálný svět v celé komplikovanosti, zajímáme se jen o jeho ohraničené části, objekty, na kterých uvažujeme a studujeme reálné systémy. Reálný systém však nemusí být uvažován pouze na reálném objektu. Často jsme nuceni vytvářet návrh systému, jenž se má teprve realizovat. V tom případě postupujeme tak, že vycházíme ze znalostí analogických systémů, které mohou sloužit jako základ pro další úvahy.

Budováním abstraktního modelu rozumíme formulaci zjednodušeného popisu systému abstrahujícího od všech nedůležitých skutečností vzhledem k cíli a účelu modelu. Jádrem tohoto procesu je identifikace jeho vhodných složek, které mají vliv na efektivnost, či neefektivnost systému a dále rozhodnutí, zda tyto složky budou součástí systému nebo jeho okolí. Různé specifické cíle a účely modelů můžeme zahrnout do několika základních tříd:

a) *Vyhodnocení* - určení, jak je navržený systém vhodný v absolutním smyslu; jeho chování studujeme pro určitá specifická kritéria.

b) *Srovnávání* - porovnávání funkcí systémů vzhledem k jejich určitým alternativním složkám nebo operačním strategiím.

c) *Predikce* - vyhodnocení chování systému za určitých, v reálném systému potenciálních, podmínek.

d) *Analýza citlivosti* - určení těch faktorů (parametrů), jež jsou pro činnost celého systému nejvýznamnější.

e) *Optimalizace* - nalezení takové kombinace parametrů, která vede k nejlepší odezvě systému.

f) *Funkcionální* vztahy - objevení povahy závislostí mezi nejvýznamnějšími parametry a odezvou systému.

Tento výčet účelů a cílů modelu není jistě vyčerpávající a mohou existovat i jiné důvody, pro které se rozhodneme metody modelování na počítačích využít. Explicitní vymezení účelu modelu má však významný dopad na celý proces budování abstraktního modelu i na vlastní experimentování se simulačním modelem. Je-li např. cílem modelu absolutní vyhodnocení navrženého nebo existujícího systému, pak musí být model velmi přesný. Je-li však cílem modelu relativní srovnávání dvou nebo více systémů, pak může být přesnost modelu pouze relativní a absolutní odchylky odezvy modelu a reálného systému se mohou dosti lišit. Z hlediska teorie systémů předpokládáme mezi modelovaným a abstraktním systémem homomorfní vztah, který vyžaduje korespondenci prvků abstraktního systému s podsystémy modelovaného systému a korespondenci struktur abstraktního a modelovaného systému (viz *kap. 2*).

1.2 Vytvoření simulačního modelu

Pod pojmem simulační model rozumíme abstraktní model zapsaný formou programu v programovacím jazyce. Na rozdíl od dvojice

modelovaný systém - abstraktní systém,

kde předpokládáme homomorfní vztah, vyžadujeme mezi dvojicí

abstraktní systém - simulační model

izomorfní vztah, jenž představuje silnější vztah ekvivalence mezi abstraktními systémy - shodnost struktur a chování prvků uvažovaných systémů.

S rozvojem metod modelování systémů na číslicových počítačích byla vyvinuta třída speciálních programovacích jazyků, simulační jazyky, které poskytují prostředky usnadňující efektivní popis jednotlivých složek modelu, jejich chování a propojení (struktury) a umožňují tak snadnější zápis a verifikaci izomorfních simulačních modelů. Pro konkrétní počítač musí být vytvořen překladač a procesor simulačního jazyka. Jejich úkolem je pak převést simulační model na konkrétní datovou informační strukturu implementovanou v počítači a nad touto strukturou provádět operace vymezené sémantikou jazyka a realizované jeho procesorem.

Podle přístupu k reprezentaci abstraktního systému simulačním modelem rozeznáváme různé simulační jazyky. Určujícím faktorem pro jejich klasifikaci je charakter časové množiny a změn stavových, vstupních a výstupních proměnných těch systémů, pro jejichž modelování mají simulační jazyky sloužit. Mluvíme pak o spojitých simulačních jazycích pro modelování

spojitých systémů, o diskrétních pro modelování diskrétních systémů a o jazycích kombinovaných pro modelování systémů obsahujících spojité i diskrétní prvky (viz *kap. 2*).

1.3 Simulace

Simulací označujeme etapu experimentování s reprezentací simulačního modelu. Jejím cílem je analýza chování systému v závislosti na vstupních veličinách a na hodnotách parametrů. Vlastní etapě simulace předchází verifikace simulačního modelu, kdy ověřujeme korespondenci simulačního a abstraktního modelu, zpravidla tedy izomorfní vztah mezi těmito dvěma modely. Analogicky s programy v běžných programovacích jazycích představuje verifikace simulačního modelu jeho ladění jak po stránce syntaktické, tak, a to zvláště, po stránce sémantické.

Proces simulace spočívá v opakovaném řešení modelu, v provádění simulačních běhů, které jsou charakterizovány určitými hodnotami parametrů modelu a určitými podněty z okolí. S každým simulačním během je spojeno vyhodnocení výstupních dat simulačního modelu, které představuje informaci o chování systému, tj. o jeho reakcích na podněty z okolí. Simulační běhy, jako základní jednotky simulace, opakujeme tak dlouho, dokud nezískáme dostatečnou informaci o chování systému nebo pokud nenalezneme takové hodnoty parametrů, pro něž má systém žádané chování. Důležitou složkou simulace je neustálá konfrontace informací, které o modelovaném systému máme a které simulací získáváme. Tato konfrontace nám pomáhá rozhodnout jeden z nejobtížnějších problémů modelování – problém validity (platnosti) modelu. Ověřování validity modelu je proces, v němž se snažíme dokázat, že skutečně pracujeme s modelem adekvátním modelovanému systému. Poněvadž nelze absolutně dokázat přesnost modelu, chápeme validitu modelu v relativním smyslu jako jistou hladinu spolehlivosti, která umožňuje přijmout, jako správné, výsledky a závěry odvozené z modelu. V případě, že chování modelu neodpovídá předpokládanému chování originálu, musíme model modifikovat s přihlédnutím k informacím, které jsme získali předcházející simulací.

Pro efektivní realizaci těchto etap modelování potřebujeme jednak prostředky pro práci s abstraktními systémy a jednak prostředky pro programování simulačních modelů a experimentování se simulačními modely. Charakteristika těchto prostředků je náplní dalších kapitol.

Kapitola 2

Základní pojmy z teorie systémů

V této kapitole definujeme základní pojmy teorie systémů, která přináší pro celou oblast modelování stále nové a nové podněty.

V literatuře pojednávající o teorii systémů můžeme pozorovat dva směry při vytváření základních definic. Jednak jde o definici obecného univerzálního systému, kterou lze aplikovat na nejrůznější typy objektů a jednak jde o budování definic určitých tříd systémů, které odrážejí jejich specifické vlastnosti. Dalším pohledem, v němž se tyto definice vzájemně liší, je výběr potřebného matematického aparátu používaného pro výklad základních pojmů.

V našich úvahách budeme vycházet z pojetí obecného kybernetického systému, které vede k dekompozici zkoumaného systému na prvky a vazby mezi nimi. Toto pojetí je velmi rozšířené v mnoha vědních oborech a je rovněž blízké technickému pohledu na zkoumané objekty. Pro vymezení základních pojmů použijeme jednoduchého matematického aparátu (množin, relací, funkcí), jenž je funkční nejen z hlediska výstavby abstraktních modelů, ale má rovněž význam pro vlastní programování simulačních modelů, které jsou v podstatě přísně formalizovaným zápisem daným syntaxí a sémantikou příslušného programovacího jazyka.

Seznam použitých symbolů v kapitole 2:

U	univerzum systému
u_i	i-tý prvek systému
$X(u_i)$	množina vstupních proměnných prvku u_i
$S(u_i)$	množina stavových proměnných prvku u_i
$Y(u_i)$	množina výstupních proměnných prvku u_i
$\sigma_I(u_i)$	vstupní prostor prvku u_i
$\sigma_S(u_i)$	stavový prostor prvku u_i
$\sigma_O(u_i)$	výstupní prostor prvku u_i
T_i	časová množina prvku u_i
z_i	chování prvku u_i
$\|A\|$	počet prvků množiny A
R_{ij}	propojení prvků u_i a u_j
X	množina vstupních proměnných systému S
Y	množina výstupních proměnných systému S
$St(S)$	struktura systému S
R,$\bar{R}$	propojení prvků systému
S	systém

u_O	okolí systému
V	množina vstupních proměnných systému
W	množina výstupních proměnných systému
I	vstupní abeceda systému
O	výstupní abeceda systému
$\sigma_I(u_o)$	vstupní prostor systému
$\sigma_O(u_o)$	výstupní prostor systému
Q_I	struktura vstupních vazeb okolí se systémem
Q_O	struktura výstupních vazeb okolí se systémem
$Q = Q_I \cup Q_O$	propojení systému s okolím
χ	chování systému
$R \Big\vert \; \bar{A}$	
$R \Big\vert \; \bar{B}$	podmnožiny binární relace R
$R \Big\vert \; \bar{B}$ $\quad\; \bar{A}$	
$S(U^*)$	podsystém systému S vzhledem k univerzu U^*
Π	rozklad univerza
h,g	zobrazení definující homomorfní struktury
ξ_1,ξ_2	zobrazení popisující chování prvku
$f \bullet g$	kompozice zobrazení (f po g)

2.1 Prvek systému

Prvky systému považujeme obecně za elementární, dále nedělitelné části systému představující jeho rozkladové (dekompoziční) složky. Prostřednictvím prvků popisujeme strukturální vlastnosti systému; chování prvků ve vzájemných interakcích, které vyplývají ze struktury systému, určuje chování celého systému.

Dále budeme formálně specifikovat pojem prvek systému tak, aby splňoval požadavky zahrnuté v tomto pojetí prvku systému.

Předpokládejme, že systém S je tvořen konečnou množinou prvků:

$$U = \{u_1, u_2, ..., u_n\}$$

Množinu U nazýváme *univerzum* systému S. Prvek systému $u_i \in U$ vymezíme tímto souborem atributů:

(1) Množinou $X(u_i) = \{x_1^i, x_2^i, ..., x_{k_i}^i\}$ *vstupních proměnných* prvku u_i.

(2) Množinou $S(u_i) = \{s_1^i, s_2^i, ..., s_{l_i}^i\}$ *stavových proměnných* prvku u_i.

(3) Množinou $Y(u_i) = \{y_1^i, y_2^i, ..., y_{m_i}^i\}$ *výstupních proměnných* prvku u_i.

(4) Množinami určujícími obory hodnot vstupních, stavových a výstupních proměnných — tzv. abecedami vstupních, stavových a výstupních proměnných.

(5) Množinami kombinací hodnot vstupních, stavových, výstupních proměnných *vstupními*, *stavovými*, *výstupními* prostory prvku u_i.

Označme:

$\sigma_I(u_i) = \{(x_1^i, x_2^i, ..., x_{k_i}^i)\}$ vstupní prostor prvku u_i,

$\sigma_S(u_i) = \{(s_1^i, s_2^i, ..., s_{l_i}^i)\}$ stavový prostor prvku u_i,

$\sigma_O(u_i) = \{(y_1^i, y_2^i, ..., y_{m_i}^i)\}$ výstupní prostor prvku u_i,

Prvky těchto prostorů nazýváme *vstupními*, *stavovými*, *výstupními* vektory prvku u_i; jsou reprezentovány uspořádanými q-ticemi ($q = k_i$, resp. l_i, resp. m_i) prvků *abeced* vstupních, stavových, výstupních proměnných prvku u_i.

(6) *Časovou množinou* T_i, což je množina všech časových okamžiků, ve kterých jsou definovány hodnoty vstupních, stavových a výstupních proměnných.

(7) *Chováním prvku* ve formě dvojice zobrazení $z_i = (z_{S_i}, z_{O_i})$ tohoto tvaru ($\mathcal{R}$ je množina všech restrikcí (zúžení) všech zobrazení $T_i \to \sigma_I(u_i)$):

$$z_{S_i} \begin{cases} \mathcal{R} \times \sigma_S(u_i) \times T_i \times T_i \to \sigma_S(u_i) \times T_i \\ (\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) \to (\xi_{S_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2), t_2'), \end{cases}$$

$$z_{O_i} \begin{cases} \mathcal{R} \times \sigma_S(u_i) \times T_i \times T_i \to \sigma_O(u_i) \times T_i \\ (\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) \to (\xi_{O_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2), t_2'), \end{cases}$$

Zobrazení ξ_{S_i} přiřazuje časovému průběhu vstupních proměnných mezi dvěma okamžiky t_1 a t_2 v závislosti na počátečním stavu $s(t_1)$ stav $s(t_2')$, $t_2' \geq t_2 > t_1$. Podobný význam má pro výstupní proměnnou zobrazení ξ_{O_i}.

Příklad

Mějme prvek u, který reprezentuje integrační článek. Nechť $X(u_i) = x$, $Y(u_i) = y$, $T = \langle 0, \infty)$, $I = O = \mathbf{R}$, I je vstupní abeceda, O je výstupní abeceda prvku. Chování integračního článku popisují zobrazení:

$$\begin{aligned} z_{S_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) &:= (s(t_1) + \int_{t_1}^{t_2} \vec{x}(\tau) d\tau, t_2), \\ z_{O_i} &:= z_{S_i}. \end{aligned}$$

Poznámka

Aby zobrazení (z_{S_i}, z_{O_i}) vyjadřovalo chování skutečného prvku systému, musí platit:

$$\xi_{S_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_1) = s(t_1),$$

$$\xi_{S_i}(\vec{x}|_{<t_1,t_3>}, s(t_1), t_1, t_3) = \xi_{S_i}(\vec{x}|_{<t_2,t_3>}, \xi_{S_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2), t_2, t_3).$$

2.2 Prvky se stejným chováním

Uvažujme o podmínkách, za kterých můžeme pokládat chování z_i prvku u_i a chování z_j prvku u_j za stejné.

Vyjdeme-li z definice chování prvku, pak je podmínkou pro shodnost chování dvou prvků u_i a u_j shodnost zobrazení z_i a z_j. Tato podmínka však implikuje shodnost definičních oborů a oborů hodnot zobrazení z_i i z_j. Z hlediska modelování, kdy je volba konkrétních kódování

podnětů, stavů, či reakcí prvků závislá na prostředí, v němž model vytváříme, je tato implikace příliš omezující. Abychom toto omezení odstranili, musíme nejdříve transformovat vstupní, výstupní a stavový prostor prvku u_i tak, aby byl shodný s příslušným prostorem prvku u_j a teprve potom můžeme srovnávat jejich chování.

Označme symbolem $|A|$ počet prvků množiny A a uvažujme dva prvky u_i, u_j systému S, o nichž předpokládáme, že $|X(u_i)| = |X(u_j)|$, $|S(u_i)| = |S(u_j)|$ a $|Y(u_i)| = |Y(u_j)|$, tj., že vstupní, stavové a výstupní vektory prvku u_i mají stejný počet složek jako odpovídající vektory prvku u_j. Dále předpokládejme existenci bijektivních zobrazení α_{ij}, β_{ij}, γ_{ij}, δ_{ij} :

$$\begin{aligned} \alpha_{ij} &: \sigma_I(u_i) \to \sigma_I(u_j), \\ \beta_{ij} &: \sigma_S(u_i) \to \sigma_S(u_j), \\ \gamma_{ij} &: \sigma_O(u_i) \to \sigma_O(u_j), \\ \delta_{ij} &: T_i \to T_j. \end{aligned}$$

Zobrazení α_{ij}, β_{ij}, γ_{ij} lze vytvářet na základě korespondence abeced vstupních, stavových a výstupních proměnných prvků u_i a u_j, jejichž prostřednictvím jsou kódovány podněty, stavy a reakce prvků systému. Zobrazení δ_{ij} zachovává uspořádání, tzn., že pro libovolné $t, t' \in T_i$, pro které $t \leq t'$ je $\delta_{ij}(t) \leq \delta_{ij}(t')$. Nyní můžeme definovat pojem *prvky se stejným chováním*.

Nechť z_i je chování prvku u_i, z_j je chování prvku u_j a nechť zobrazení α_{ij}, β_{ij} a γ_{ij} jsou bijektivní zobrazení definována mezi vstupním, stavovým a výstupním prostorem prvku u_i a prvku u_j a δ_{ij} je prosté zobrazení definované mezi časovými množinami T_i a T_j. Prvky u_i a u_j mají *stejné chování*, což budeme zapisovat $z_i \equiv z_j$, jestliže pro každé $t_1, t_2 \in T_i$ platí:

jestliže

$$z_{S_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) = (s(t'_2), t'_2),$$
$$z_{O_i}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) = (y(t'_2), t'_2),$$

potom také

$$z_{S_j}((\alpha_{ij} \cdot \vec{x})|_{<t''_1,t''_2>}, \beta_{ij}[s(t''_1)], t''_1, t''_2) = (\beta_{ij}[s(t'''_2)], \delta_{ij}(t'_2)),$$
$$z_{O_j}((\alpha_{ij} \cdot \vec{x})|_{<t''_1,t''_2>}, \beta_{ij}[s(t''_1)], t''_1, t''_2) = (\gamma_{ij}[y(t'''_2)], \delta_{ij}(t'_2)),$$

kde jsme označili $t''_1 = \delta_{ij}(t_1)$, $t''_2 = \delta_{ij}(t_2)$, $t'''_2 = \delta_{ij}(t'_2)$.

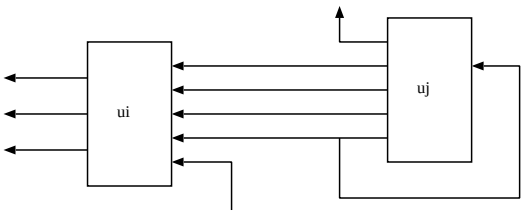
2.3 Charakteristika systému

Uvažujeme systém S obsahující prvky $u_1, u_2, ..., u_n$, které tvoří univerzum U systému S. Jsou-li u_i, u_j dva libovolné prvky systému S, zajímá nás, zda je chování prvku u_i závislé na chování prvku u_j. Nejdříve zodpovíme otázku "přímé závislosti" definováním vazeb mezi prvky systému, realizovaných *propojením* (ztotožněním) některých vstupních proměnných prvku u_i s výstupními proměnnými prvku u_j.

Nechť $X(u_i)$ je množina vstupních proměnných prvku u_i a $Y(u_j)$ je množina výstupních proměnných prvku u_j. Propojení prvku u_j s prvkem u_i lze popsat jako binární relaci

$$\langle R_{ij}, X(u_i), Y(u_j) \rangle,$$

$R_{ij} = \{(x_k^i, y_l^j) |$ vstupní proměnná x_k^i prvku u_i je propojena s výstupní proměnnou y_l^j prvku $u_j\}$.



Obrázek 2.1: Příklad propojení prvků systému

$$\begin{aligned} R_{ij} &= \{(x_1^i, y_2^j), (x_2^i, y_3^j), (x_3^i, y_4^j), (x_4^i, y_5^j)\} \\ R_{ij} &= \emptyset \\ R_{ij} &= \{(x_1^j, y_5^j)\} \end{aligned}$$

Dále předpokládejme, že libovolná vstupní proměnná může být propojena nanejvýš s jedinou výstupní proměnnou. Tento předpoklad vylučuje možnost vzniku konfliktních situací při interakci prvků. Relace R_{ij} potom implikuje existenci funkce $\varphi_{ij} : \bar{X}_j(u_i) \to \bar{Y}_i(u_j)$, kde

$$\bar{X}_j(u_i) \subseteq X(u_i); \quad \bar{X}_j(u_i) = \{x_k^i \mid \exists\, y \in Y(u_j) : (x_k^i, y) \in R_{ij}\},$$

$$\bar{Y}_i(u_j) \subseteq Y(u_j); \quad \bar{Y}_i(u_j) = \{y_j^l \mid \exists\, x \in X(u_i) : (x, y_l^j) \in R_{ij}\},$$

a $\varphi_{ij}(x) = y$ právě když $(x, y) \in R_{ij}$.

Funkce φ_{ij} je funkcí na množinu $\bar{Y}_i(u_j)$. Množina $\bar{X}_j(u_i)$ reprezentuje ty vstupní proměnné prvku u_i, jejichž prostřednictvím chování prvku u_i přímo závisí na chování prvku u_j; množina $\bar{Y}_i(u_j)$ reprezentuje ty výstupní proměnné prvku u_j, které přímo ovlivňují chování prvku u_i.

Pro počet prvků množiny $\bar{X}_j(u_i)$ a $\bar{Y}_i(u_j)$ platí

$$|\bar{X}_j(u_i)| \geq |\bar{Y}_i(u_j)|.$$

Je-li $\bar{Y}_i(u_j) = \emptyset$, pak také $\bar{X}_j(u_i) = \emptyset$, což znamená, že prvek u_i není propojen (není přímo závislý) s prvkem u_j.

Je-li $i = j$, pak $\bar{Y}_i(u_j) \neq \emptyset$ implikuje propojení prvku u_i se sebou samým. Tento specifický typ propojení prvku u_i nazýváme jednoduchou zpětnou vazbou prvku u_i (na *obr. 2.1* je to vazba (x_1^j, y_5^j)). Pojem propojení dvou prvků snadno rozšíříme na propojení všech prvků systému.

Nechť

$X = \bigcup_{i=1}^{n} X(u_i)$ je množina všech vstupních proměnných prvků systému S a

$Y = \bigcup_{i=1}^{n} Y(u_i)$ je množina všech výstupních proměnných prvků systému S.

Relace $\langle R, X, Y\rangle$, $R = \{(x, y) \mid \exists\, i, j \in \{1, ..., n\} : (x, y) \in R_{ij}\}$ definuje propojení vstupů a výstupů prvků univerza U. Jinak vyjádřeno

$$R = \bigcup_{i=1,j=1}^{n} R_{ij}.$$

Relaci $R = \langle R, X, Y \rangle$ nazýváme *charakteristika systému* S. Charakteristika systému popisuje strukturu systému na nejnižší úrovni jeho dekompozice. Často je charakteristika udávána ve tvaru incidenční matice, která představuje maticovou reprezentaci binární relace R.

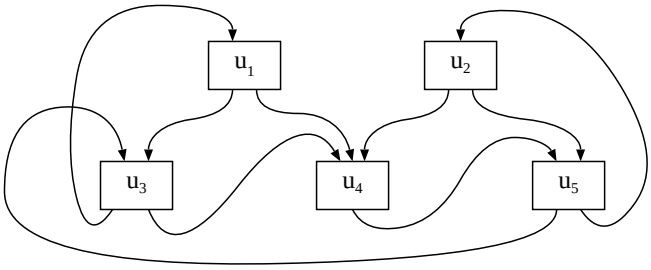
Příklad 2.1

Uvažujme systém S, jehož propojení je graficky znázorněno na *obr. 2.2*. Předpokládejme, že vstupní i výstupní proměnné jsou indexovány zleva doprava.

$U = \{u_1, u_2, u_3, u_4, u_5\}$,

$X(u_1) = \{x_1^1\}$, $Y(u_1) = \{y_1^1, y_2^1\}$,
$X(u_2) = \{x_1^2\}$, $Y(u_2) = \{y_1^2, y_2^2\}$,
$X(u_3) = \{x_1^3, x_2^3\}$, $Y(u_3) = \{y_1^3, y_2^3\}$,
$X(u_4) = \{x_1^4, x_2^4, x_3^4\}$, $Y(u_4) = \{y_1^4\}$,
$X(u_5) = \{x_1^5, x_2^5\}$, $Y(u_5) = \{y_1^5\}$,

$R = \{(x_1^1, y_1^3), (x_1^2, y_1^5), (x_1^3, y_1^5), (x_2^3, y_1^1), (x_1^4, y_2^3), (x_2^4, y_2^1), (x_3^4, y_1^2), (x_1^5, y_1^4), (x_2^5, y_2^5)\}$.



Obrázek 2.2: Příklad propojení prvků systému

Dále se budeme zabývat otázkami strukturální podobnosti dvou systémů. Tyto úvahy někdy vedou pouze k vyšetřování existence, resp. neexistence přímé (strukturální) vazby mezi prvky systému. Uvedeme proto abstraktnější vymezení pojmu propojení prvku systému, které nebude specifikovat prostředníky uvažovaných vazeb, t.j. vstupní a výstupní proměnné prvků.

Nechť je relace $\bar{R}$ na univerzu U definována takto: $\bar{R} = \{(u_i, u_j) \mid u_i, u_j \in U$ a existuje prvek $r \in R$ takový, že $r = (x_k^i, y_l^j)$, kde $x_k^i \in X(u_i)$,$y_l^j \in Y(u_i)\}$.

Relace $\bar{R}$ tedy určuje ty dvojice prvků systému S, mezi nimiž existuje vazba.

Příklad 2.2

Pro strukturu systému z *obr. 2.2* platí:

$$\bar{R} = \{(u_1, u_3), (u_2, u_5), (u_3, u_5), (u_3, u_1), (u_4, u_3), (u_4, u_1), (u_4, u_2), (u_5, u_4), (u_5, u_2)\}.$$

2.4 Systém

Systémem S rozumíme dvojici $S = \langle U, R \rangle$, kde $U = \{u_1, ..., u_n\}$ je univerzum systému S a $R = \langle R, X, Y \rangle$ je charakteristika systému S. Atributy prvků univerza systému určují tyto důležité atributy celého systému:

$X = \bigcup_{i=1}^{n} X(u_i)$ množinu vstupních proměnných prvků systému S,

$Y = \bigcup_{i=1}^{n} Y(u_i)$ množinu výstupních proměnných prvků systému S,

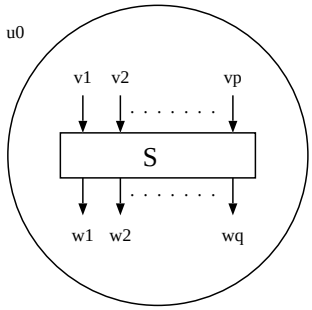
$S = \bigcup_{i=1}^{n} S(u_i)$ množinu stavových proměnných prvků systému S,

$T = \bigcup_{i=1}^{n} T_i$ časovou množinu systému S,

$Z = \{z_i \mid i = 1, ..., n\}$ chování prvků systému S.

2.5 Okolí systému

Nechť $S = \langle U, R \rangle$ je systém. Okolím systému budeme rozumět entitu, která je zdrojem podnětů na systém S a která přijímá reakce systému S na podněty. V některých případech je výhodné uvažovat okolí systému jako speciální prvek. Okolí systému S označíme symbolem u_o a budeme specifikovat jeho vlastnosti.



Obrázek 2.3: Systém a jeho okolí

1. K okolí systému u_o přísluší množina proměnných $V = \{v_1, ..., v_p\}$, která definuje *množinu vstupních proměnných* systému S.
2. K okolí u přísluší množina proměnných $W = \{w_1, ..., w_q\}$, jež definuje *množinu výstupních proměnných* systému S.
3. Proměnné $v_i \in V$ nabývají hodnot z množiny, kterou budeme označovat písmenem I a nazývá se *vstupní abecedou* systému S.
4. Proměnné $w_i \in W$ nabývají hodnot z množiny, kterou budeme označovat písmenem O. Množina O se nazývá *výstupní abeceda* systému S.

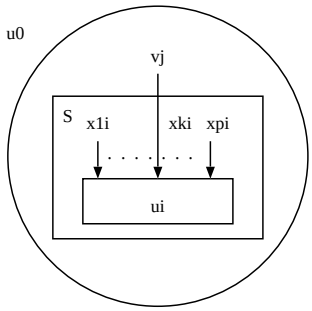
5. Vstupním vektorem systému S budeme rozumět uspořádanou p-tici, vektor $(v_1, ..., v_p)$ hodnot ze vstupní abecedy I; i-tá složka vektoru odpovídá hodnotě vstupní proměnné v_i systému S. Množinu
$$\sigma_I(u_o) = \{(v_1, ..., v_p)\}$$
všech vstupních vektorů nazýváme *vstupním prostorem* systému S.

6. Výstupním vektorem systému S budeme rozumět uspořádanou q-tici $(w_1, ..., w_g)$ hodnot z výstupní abecedy systému S; i-tá složka vektoru $(w_1, ..., w_q)$ odpovídá hodnotě výstupní proměnné w_i systému S. Množinu
$$\sigma_O(u_o) = \{(w_1, ..., w_q)\}$$
všech různých reakcí nazýváme *výstupním prostorem* systému S.

Dále uvedeme formální popis struktury "propojení" okolí se systémem a systému s okolím. Nechť X je množina vstupních proměnných prvků systému S a Y je množina výstupních proměnných prvků systému S.

Relace $Q_I \subseteq X \times V$ popisuje *strukturu* vstupní vazby mezi okolím a systémem prostřednictvím vstupních proměnných systému. Prvek $r = (x, v_j), r \in Q_I$ interpretujeme takto: vstupní proměnná $x \in X(u_i)$ prvku u_i systému S je ztotožněna se vstupní proměnnou v_j systému S.

Analogicky binární relace $Q_O \subseteq W \times Y$ popisuje výstupní vazby mezi systémem a okolím prostřednictvím výstupních proměnných. Prvek $r = (w_i, y)$, $r \in Q_O$ interpretujeme takto: výstupní proměnná w_i systému S je ztotožněna s výstupní proměnnou $y \in Y(u_j)$ prvku u_j systému S.



Obrázek 2.4: Interpretace prvku $R = (x_k^i, v_j), \quad r \in Q_I$

Označme $Q = Q_I \cup Q_O$. Okolí u_o systému S je pak určeno:

1. Množinou V vstupních proměnných a množinou W výstupních proměnných systému.
2. Vstupní abecedou I a výstupní abecedou O.
3. Vstupním prostorem $\sigma_I(u_o)$ a výstupním prostorem $\sigma_O(u_o)$.
4. Strukturou propojení okolí se systémem, která je popsána relací Q.

Na základě relace Q, jež detailně popisuje strukturální vazby mezi systémem a okolím, můžeme definovat abstraktnější relaci $\bar{Q}$ analogicky s definicí relace $\bar{R}$ z odstavce 2.3.

Relace $\langle \bar{Q}, U \cup \{u_o\}, U \cup \{u_o\} \rangle$ je definována takto:
$\bar{Q} = \{(u_i, u_j) | u_i \in U \cup \{u_o\}, u_j \in U \cup \{u_o\}$, existuje prvek $r \in Q$ takový, že

1. $r = (x_k^i, v_j)$, kde $x_k^i \in X(u_i)$ a $v_j \in V$; tj. $u_j = u_o$, nebo

2. $r = (w_i, y_k^j)$ kde $w_i \in W$ a $y_k^j \in Y(u_j)$; tj. $u_i = u_o$ }.

2.6 Izomorfní charakteristiky, izomorfní systémy

Uvažujeme dva systémy $S_1 = \langle U_1, R_1 \rangle$ a $S_2 = \langle U_2, R_2 \rangle$. Charakteristika R_1 systému S_1 a charakteristika R_2 systému S_2 jsou izomorfní (systémy S_1 a S_2 mají izomorfní charakteristiky), existují-li bijektivní zobrazení

$$\begin{aligned} \psi &: U_1 \to U_2, \\ \omega_I &: X_1 \to X_2, \\ \omega_O &: Y_1 \to Y_2 \end{aligned}$$

taková, že platí:

$$\forall u_i, u_j \in U_1 \quad \forall x \in U(u_i) \quad \forall y \in Y(u_j) :$$

$$\omega_I(x) \in X(\psi(u_i)) \wedge \omega_O(y) \in Y(\psi(u_j)) \wedge (x, y) \in R_1 \;\Leftrightarrow\; (\omega_I(x), \omega_O(y)) \in R_2.$$

Systémy S_1 a S_2 mají tedy izomorfní charakteristiky, lze-li prvkům univerza U_1 jednoznačně přiřadit prvky univerza U_2 tak, že pro každý prvek $u \in U_1$ a jeho obraz $u' \in U_2$ platí:

1. Existuje jednojednoznačné přiřazení vstupních, resp. výstupních proměnných prvku u na vstupní, resp. výstupní proměnné prvku u'.

2. Zobrazení prvků systému S_1 na prvky systému S_2 a zobrazení vstupních, resp. výstupních proměnných všech prvků systému S_1 na vstupní, resp. výstupníĺ proměnné prvků systému S_2 implikuje stejné vazby mezi prvky systému S_2 jako mezi prvky systému S_1.

Na základě pojmu izomorfní charakteristiky dvou systémů budeme definovat vztah ekvivalence mezi dvěma systémy — tzv. *izomorfní systémy.*

Uvažujeme dva systémy $S_1 = \langle U_1, R_1 \rangle$ a $S_2 = \langle U_2, R_2 \rangle$, které mají izomorfní charakteristiky. Nechť ψ je bijektivní zobrazení $U_1 \to U_2$ z definice izomorfních charakteristik. Říkáme, že systémy S_1 a S_2 jsou izomorfní, je-li chování každého prvku $u \in U_1$ systému S_1 a odpovídajícího prvku $u' \in U_2$ systému S_2, $u' = \psi(u)$, stejné.

Z definice izomorfismu systému vyplývá, že izomorfní vztah je reflexivní, symetrický i tranzitivní. Tento vztah je tedy určitým typem ekvivalence systémů, která je pro vytváření modelů velmi potřebná. Poněvadž však dalším důsledkem definice izomorfních systémů je skutečnost, že izomorfní systémy mají stejný počet prvků a vazeb, vytváření izomorfních modelů je pro většinu tříd reálných systémů prakticky nedostižné. Může se však uplatnit při vytváření ekvivalentních modelů, např. abstraktní model — simulační model.

2.7 Chování systému

Uvažujeme systém $S = \langle U, R \rangle$ s okolím u_o, které má se systémem S definované strukturální vazby prostřednictvím relace Q. Chováním systému zpravidla rozumíme závislost reakcí systému na podněty z okolí, přičemž je tato závislost uvažována v čase.

Nechť T je časová množina systému S. Označíme-li B^A množinu všech zobrazení $A \to B$, pak chování systému S můžeme definovat jako zobrazení

$$\chi : [\sigma_I(u_O)]^T \to [\sigma_O(u_o)]^T$$

Jinými slovy, chování systému S je funkcionál, který každému časovému průběhu vstupní veličiny přiřadí časový průběh vstupních veličin systému S. Zobrazení χ musí splňovat tuto podmínku:

jsou-li $\vec{x}_1$, $\vec{x}_2$ zobrazení $T \to \sigma_I(u_o)$ a existuje $t_o \in T$, že

$$\forall t \in T \quad t < t_o \Rightarrow \vec{x}_1(t) = \vec{x}_2(t),$$

potom existuje $t_1 \in T$, $t_1 \geq t_o$, že platí:

$$\forall t \in T \quad t < t_1 \Rightarrow (\chi(\vec{x}_1))(t) = (\chi(\vec{x}_2))(t),$$

tj. jestliže na vstup systému přivedeme jednou vstupní veličiny s průběhem $\vec{x}_1$ a podruhé s průběhem $\vec{x}_2$ a platí, že tyto průběhy jsou alespoň do okamžiku t_o shodné, potom odpovídající odezvy na výstupu $\chi(\vec{x}_1)$ a $\chi(\vec{x}_2)$ jsou shodné alespoň do okamžiku $t_1 \geq t_o$, tj. změna na výstupu nenastane dříve, než změna na vstupu.

Proč přesto popisujeme chování prvku a systému formálně jinými prostředky?

1. Definici chování prvku se snažíme co nejvíce přiblížit definici Mealyho automatu, což souvisí s modelováním prvků na počítači. Nevýhodou tohoto pojetí je to, že dva prvky mají stejné chování jen tehdy, mají-li množiny stavů o stejné mohutnosti. To může vést ke stavu, kdy oba prvky reagují stejně na průběhy vstupních proměnných na výstupu, ale jejich stavy se mění jiným způsobem. Takové prvky pak nemají stejné chování.

2. Tuto nevýhodu odstraňuje definice chování systému, která zase neříká nic o reprezentaci systému programovými prostředky. To ostatně ani není nutné, protože tato informace je dána charakteristikou systému a prostřednictvím chování jednotlivých prvků systému.

2.8 Systémy se stejným chováním

Uvažujeme systém $S_1 = \langle U_1, R_1 \rangle$ s časovou množinou T_1 a chováním χ_1, a systém $S_2 = \langle U_2, R_2 \rangle$ s časovou množinou T_2 a chováním χ_2. Nechť $\sigma_{I_i}, \sigma_{O_i}$ jsou vstupní a výstupní prostory i-tého systému, $i \in \{1, 2\}$. Systémy S_1 a S_2 mají stejné chování právě když existují bijektivní zobrazení:

$$\begin{aligned}
\mu &: \sigma_{I_1} \to \sigma_{I_2}, \\
\nu &: \sigma_{O_1} \to \sigma_{O_2}, \\
\tau &: T_1 \to T_2 \quad \forall t_1, t_2 \in T_1 \quad t_1 < t_2 \Rightarrow \tau(t_1) < \tau(t_2)
\end{aligned}$$

taková, že platí:

$$\forall \vec{x} \in \sigma_{I_1}^{T_1} \quad \chi_1(\vec{x}) = \vec{y} \;\Rightarrow\; \chi_2(\mu \bullet \vec{x} \bullet \tau^{-1}) = \nu \bullet \vec{y} \bullet \tau^{-1}, \quad \vec{y} \in \sigma_{O_1}^{T_1}.$$

2.9 Definice podsystému

Dříve, než budeme definovat podsystém, zavedeme označení pro speciální podmnožiny relace.

Konvence: Nechť A,B jsou množiny a nechť $R \subseteq A \times B$ je binární relace z A do B. Jsou-li $\bar{A}$, resp. $\bar{B}$ podmnožiny množiny A, resp. B, pak můžeme definovat restrikce relace R vzhledem k množině $\bar{A}$ nebo $\bar{B}$ nebo k oběma. V dalším textu je budeme označovat takto:

$$\begin{aligned} R\Big|_{\bar{A}} &= \{(a,b) \mid a \in \bar{A} \wedge (a,b) \in R\}, \\ R\Big|^{\bar{B}} &= \{(a,b) \mid a \in \bar{B} \wedge (a,b) \in R\}, \\ R\Big|^{\bar{B}}_{\bar{A}} &= \{(a,b) \mid a \in \bar{A} \wedge b \in \bar{B} \wedge (a,b) \in R\}. \end{aligned}$$

Nechť $S = \langle U, R \rangle$ je systém. *Podsystémem* systému S je systém $S^* = \langle U^*, R^* \rangle$, kde

(1) $U^* \subseteq U$, $U^* = \{u_{i_1}, u_{i_2}, ..., u_{i_m}\}$, $m \leq n$, je-li n počet prvků systému S,

(2) $R^* \subseteq R$, $R^* = R\Big|^{Y^*}_{X^*}$, kde $X^* = \bigcup_{k=1}^{m} X(u_{i_k})$, $Y^* = \bigcup_{k=1}^{m} Y(u_{i_k})$

a $X(u_{i_k})$ resp. $Y(u_{i_k})$ jsou vstupní, resp. výstupní proměnné prvku $u_{i_k} \in U^*$.

Podle definice je podsystém S systémem a má tedy jisté chování. Toto chování můžeme analogicky s chováním systému S popsat zobrazením

$$\chi^* : [\sigma_I^*(u_o^*)]^T \to [\sigma_O^*(u_o^*)]^T.$$

Dále vyšetříme, čím je tvořeno okolí u_o^* systému S^*. Podle odstavce 2.5 je okolí systému určeno těmito atributy:

1. Množinou V vstupních a množinou W výstupních proměnných systému
2. Vstupní abecedou I a výstupní abecedou O.
3. Vstupním prostorem $\sigma_I(u_o)$ a výstupním prostorem $\sigma_o(u_o)$.
4. Strukturou propojení okolí se systémem, která je popsána relací $Q = Q_I \cup Q_O$.

Vyšetříme atributy okolí podsystému S^*. Okolí u_o^* podsystému S^* je tvořeno částí okolí u_o systému S a těmi prvky systému S, které neleží v univerzu U^* podsystému S^*.

Množinu vstupních, resp. výstupních proměnných označíme $V^* = \{v_1, ..., v_p\}$, resp. $W^* = \{w_1, ..., w_q\}$. Množina V^* je definována takto:

$$V^* = \{v \mid \exists\, x \in X^*(x,v) \in Q_I\} \cup \{y \mid \exists\, x \in X^*(x,y) \in R - R^*\}$$

V druhém případě je vstupní proměnná podsystému S^* identická s výstupní proměnnou y prvku $u_j \in U - U^*$. Analogicky množina W^* je definována takto:

$$W^* = \{w \mid \exists\, y \in Y^*(w,y) \in Q_O\} \cup \{x \mid \exists\, y \in Y^*(x,y) \in R - R^*\}$$

Na základě takto popsaných vstupních a výstupních proměnných je nyní snadné definovat relaci $Q^* = Q_I^* \cup Q_O^*$, která popisuje strukturu propojení okolí u_o^* s podsystémem S^*. Relace Q_I^* a Q_O^* jsou definovány sjednocením:

$$Q_I^* = Q_I\Big|_{X^*} \cup R\Big|^{Y - Y^*}_{X^*},$$

$$Q_O^* = Q_O \left| \begin{array}{ll} Y^* & \cup \ R \\ & \end{array} \right| \begin{array}{l} Y^* \\ X - X^* \end{array} \quad .$$

Časová množina T^* podsystému S^* je tvořena sjednocením časové množiny T systému S a časových množin těch prvků systému S, které leží v okolí systému S^*:

$$T^* = T \cup \bar{T},$$

kde $T = \{t \mid t \in T_i$ pro všechna i, pro která $u_i \in U - U^*\}$.

Konvence: Protože podsystém S^* systému S je jednoznačně určen podmnožinou U^* univerza U systému S, budeme v případech, kdy není nutné explicitně vyjadřovat množinu R^*, zapisovat podsystém $S^* = \langle U^*, R^* \rangle$ krátce jako $S(U^*)$.

Důležitým případem podsystémů systému $S = \langle U, R \rangle$ jsou triviální podsystémy $S(\{u_i\})$, $i = 1, 2, ..., n$, n je počet prvků systému S, jejichž univerzum tvoří jediný prvek systému S. V těchto podsystémech je

(1) $U^* = \{u_i\}$,

(2) $R^* = \emptyset$.

Množina vstupních, resp. výstupních proměnných podsystémů S^* je identická s množinou proměnných $X(u_i)$, resp. výstupních proměnných $Y(u_i)$ prvku u_i.

Chování χ^* podsystémů S^*

$$\chi^* : [\sigma_I^*(u_o^*)]^{T^*} \rightarrow [\sigma_O^*(u_o^*)]^{T^*}$$

(kde $\sigma_I^*(u_o^*)$, resp. $\sigma_O^*(u_o^*)$ je vstupní, resp. výstupní prostor prvku u_i a T^* je časová množina prvku u_i), je jednoznačně určeno chováním z_i prvku u_i

$$z_i \left\{ \begin{array}{l} \mathcal{R} \times \sigma_S(u_i) \times T_i \times T_i \rightarrow \sigma_S(u_i) \times T_i \\ \mathcal{R} \times \sigma_S(u_i) \times T_i \times T_i \rightarrow \sigma_O(u_i) \times T_i \end{array} \right. ,$$

kde

$\mathcal{R}$ je množina všech restrikcí všech zobrazení $T_i \rightarrow \sigma_I(u_i)$,

$\sigma_S(u_i)$ je stavový prostor prvku u_i,

$\sigma_I(u_i) = \sigma_I(u_i^*)$, $\sigma_O(u_i) = \sigma_O(u_o^*)$, $T_i = T^*$.

2.10 Homomorfní systémy

Uvažujeme dva systémy $S_1 = \langle U_1, R_1 \rangle$ a $S_2 = \langle U_2, R_2 \rangle$. Říkáme, že charakteristika R_2 systému S_2 je *homomorfní* s charakteristikou R_1 systému S_1, existují-li zobrazení h a g s těmito vlastnostmi:

$$h : U_1 \rightarrow U_2$$

je zobrazení univerza U_1 na univerzum U_2. Zobrazení h definuje na množině U_1 rozklad $\Pi = (\Pi_1, ..., \Pi_{n_2})$, n_2 je počet prvků univerza U_2, kde Π_i obsahuje právě ty prvky $u \in U_1$, pro které $h(u) = u_i$, $u_i \in U_2$. Zobrazení h tedy popisuje korespondenci jisté neprázdné podmnožiny $\Pi_i \subseteq U_1$ prvků systému S_1 s jediným prvkem $u_i \in U_2$ systému S_2.

Nechť X_1, resp. Y_1 je množina vstupních, resp. výstupních proměnných všech prvků systému S_1. Označme $X_1[\Pi_i] \subseteq X_1$ množinu vstupních proměnných těch prvků $u \in U_1$, které tvoří třídu Π_i rozkladu Π. Podobně $Y_1[\Pi_i] \subseteq Y_1$ nechť je množina výstupních proměnných těch prvků $u \in U_1$, které tvoří třídu Π_i rozkladu Π, tj.

$$X_1[\Pi_i] = \cup_{u \in \Pi_i} X_1(u),$$

$$Y_1[\Pi_i] = \cup_{u \in \Pi_i} Y_1(u).$$

Uvažujeme nyní relaci $R_{1\Pi} \subseteq R_1$, která obsahuje pouze ty vazby reprezentují propojení prvků nepatřících do stejné třídy rozkladu Π :

$$R_{1\Pi} = \{(x, y) \in R_1 \mid \exists\, u_i, u_j \in U_1 \quad x \in X_1(u_i) \wedge y \in Y_1(u_j) \wedge h(u_i) \neq h(u_j)\}.$$

Nyní můžeme definovat zobrazení g, popisující, které vazby charakteristiky R_1 se přenášejí do homomorfní charakteristiky R_2.

$g \begin{cases} R_1 \to R_2 \\ (x, y) \to (x', y') \end{cases}$ je zobrazení, pro které platí:

$x \in X_1(u_i) \wedge y \in Y_1(u_j) \Rightarrow x' \in X_2(h(u_i)) \wedge y' \in Y_2(h(u_j))$,

kde $(x', y') = g(x, y)$.

Příklad homomorfních charakteristik

Uvažujeme systém $S_1 = \langle U_1, R_1 \rangle$ (viz *obr. 2.2*).

$$U_1 = \{u_1, u_2, u_3, u_4, u_5\},$$

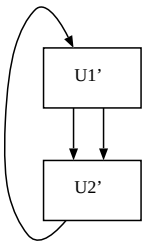
$$R_1 = \{(x_1^1, y_1^3), (x_1^2, y_1^5), (x_1^3, y_1^5), (x_2^3, y_1^1), (x_1^4, y_2^3), (x_2^4, y_2^1), (x_3^4, y_1^2), (x_1^5, y_1^4), (x_2^5, y_2^2)\}$$

a ukažme, že systém $S_2 = \langle U_2, R_2 \rangle$,

$$U_2 = \{u_1', u_2'\},$$

$$R_2 = \{(x'^1_1, y'^2_1), (x'^2_1, y'^1_1), (x'^2_2, y'^1_2)\}$$

má homomorfní charakteristiku s charakteristikou systému S_1.



Obrázek 2.5: Struktura systému S_2

Nechť $h : U_1 \to U_2$ je definována takto:

$$h(u_1) = u_1', \quad h(u_2) = u_2', \quad h(u_3) = u_1',$$

$$h(u_4) = u_2', \quad h(u_5) = u_2'.$$

Zobrazení h definuje na univerzu U_1 systému S_1 rozklad Π (viz *obr. 2.6*):

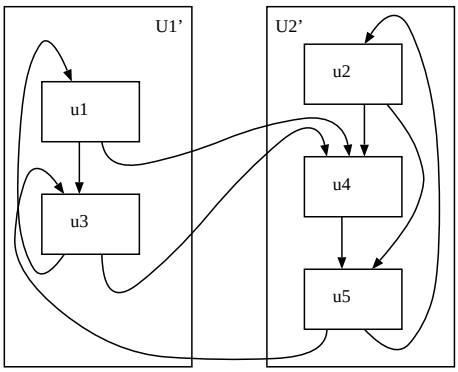
$\Pi = \{\Pi_1, \Pi_2\}$ kde $\Pi_1 = \{u_1, u_3\}$, $\Pi_2 = \{u_2, u_4, u_5\}$

Vytvořme nyní množinu $R_{1\Pi} \subseteq R_1$:

$$R_1 \left| \begin{array}{l} Y_1 - Y_1[\Pi_1] \\ X_1[\Pi_1] \end{array} \right. = \{(x_1^3, y_1^5)\} \quad \text{a}$$

$$R_1 \left| \begin{array}{l} Y_1 - Y_1[\Pi_2] \\ X_1[\Pi_2] \end{array} \right. = \{(x_1^4, y_2^3), (x_2^4, y_2^1)\},$$

tj. $R_{1\Pi} = \{(x_1^3, y_1^5), (x_1^4, y_2^3), (x_2^4, y_2^2)\}$.



Obrázek 2.6: Rozklad na U_2 definovaný zobrazením h

Nyní uvažujeme zobrazení $g : R_{1\Pi} \to R_2$

$$g((x_1^3, y_1^5)) = (x'^1_1, y'^2_1),$$
$$g((x_1^4, y_2^3)) = (x'^2_1, y'^1_1),$$
$$g((x_2^4, y_2^1)) = (x'^2_2, y'^1_2).$$

Protože uvedené zobrazení g splňuje vlastnosti z definice homomorfních charakteristik, je skutečně charakteristika R_2 systému S_2 homomorfní vzhledem k charakteristice R_1 systému S_1.

Nechť charakteristika systému $S_2 = \langle U_2, R_2 \rangle$ je homomorfní s charakteristikou systému $S = \langle U_1, R_1 \rangle$ a nechť $\Pi = (\Pi_1, ..., \Pi_{n_2})$ je rozklad univerza U_1, definovaný homomorfismem charakteristik systémů S_1 a S_2. Systém S_2 je *homomorfní* se systémem S_1 (vzhledem k systému S_1), má-li podsystém $S_1(\Pi_i)$ stejné chování jako podsystém $S_2(\{u_i\})$ pro všechna $i = 1, 2, ..., n_2$, přičemž prvek $u_i \in U_2$ je obrazem všech prvků $u \in \Pi_i \subseteq U_i$ v zobrazení h z definice homomorfních struktur.

Z definice homomorfismu plyne, že systém S_1 má více prvků než systém S_2. Homomorfní vztah systémů není, na rozdíl od izomorfního vztahu, ani reflexivní, ani symetrický; je pouze tranzitivní. Uplatňuje se zvláště při hledání abstraktního modelu reálného systému.

2.11 Klasifikace prvků systému a systémů

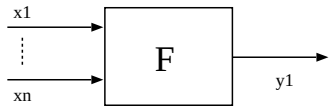
Podle charakteru časové množiny prvku sytému a podle tvaru funkcí, popisujících chování prvku, budeme definovat pojmy *spojité* a *diskrétní* prvky a prvky s *deterministickým* a *nedeterministickým* chováním.

2.11.1 Spojité a diskrétní prvky

Nechť u_i je prvek systému, T_i je jeho časová množina. Prvek u_i nazveme prvkem se *spojitým chováním*, je-li časová množina $T_i = I - E$, kde $I \in \langle 0, \infty)$ je interval a E je konečná nebo spočetná množina reálných čísel.

Příklady spojitých prvků

1. Funkční měnič

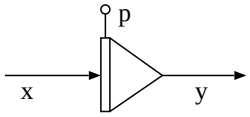


   Funkce F je spojitá. Časovou množinou T je interval, neobsahující body, ve kterých není funkce F definována. Chování prvku je jednoznačně popsáno funkcí $\bar{y} = F(\bar{x})$, kterou můžeme psát také ve tvaru:

$$z_{Si}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) = (s(t_1), t_2),$$

$$z_{Oi}(\vec{x}|_{<t_1,t_2>}, s(t_1), t_1, t_2) = (F(\vec{x}(t_2)), t_2).$$

2. Integrátor



   Popis chování viz příklad v odstavci 2.1. Prvek u_i nazveme prvkem s diskrétním chováním, je-li časová množina T_i tvaru

$$T_i = \{t_n | n \in \mathbf{N} \wedge \forall k \in \mathbf{N}(t_{k+1} \geq t_k)\}.$$

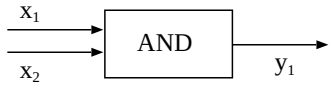
   Chování diskrétních prvků můžeme popisovat také ve tvaru

$$z_i \begin{cases} \sigma_I(u_i) \times \sigma_S(u_i) \times T_i \rightarrow \sigma_O(u_i) \times T_i \\ \sigma_I(u_i) \times \sigma_S(u_i) \times T_i \rightarrow \sigma_S(u_i) \times T_i \end{cases},$$

   který vznikne z tvaru podle definice chování prvku (odst. 2.1) když položíme $t_1 = t_2$.

Příklady diskrétních prvků

1. Logický prvek realizující konjunkci



Vstupní a výstupní abeceda tohoto prvku je binární abeceda $\{0, 1\}$. Časovou množinu $T_i = \{t_i | i = 1, 2...\}$ tvoří diskrétní časové okamžiky t_i. Tento prvek nemá stavové proměnné, a proto má zobrazení z_i tvar

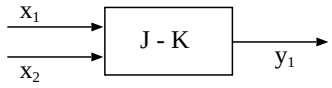
$$\vec{y}(t_i) = \xi_1(\vec{x}(t_i), t_i),$$

kde $\vec{y} = (y_1)$, $\vec{x} = (x_1, x_2)$ a ξ_1 je logická funkce zapsaná tabulkou 2.1.

Tabulka 2.1: Zápis logické funkce

x_1	x_2	y_1
0	0	0
0	1	0
1	0	0
1	1	1

2. Paměťový prvek typu J-K



Tento prvek má jednobitovou paměť, reprezentovanou jednoprvkovým stavovým vektorem $\bar{s} = (s_1)$. Chování prvku J-K může být popsáno touto dvojicí zobrazení ξ_1 a ξ_2:

$$\vec{y}(t') = \xi_1(\vec{s}(t'), t'),$$

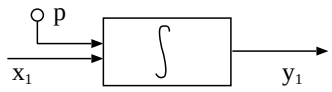
$$\vec{s}(t') = \xi_2(\vec{x}(t'), \vec{s}(t), t'), \quad t' > t; \quad t, t' \in T_1.$$

Zobrazení ξ_2 definuje tabulka 2.2. Zobrazení ξ_1 je identické zobrazení $y_1(t') = s_1(t')$.

Tabulka 2.2: Definice zobrazení ξ_2

x_1	x_2	$s_1(t)$	$s_1(t')$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

3. Diskrétní integrátor



Je-li časovou množinou T_i tohoto prvku ekvidistantní dělení intervalu $\langle a, b\rangle$ s krokem k a použijeme-li pro aproximaci výstupu y_i např. Eulerovy formule, pak lze popsat chování diskrétního integrátoru dvojici zobrazení

$$\vec{y}(t') = \xi_1(\vec{x}(t), \vec{s}(t)),$$

$$\vec{s}(t') = \xi_2(\vec{x}(t), \vec{s}(t)),$$

$t, t' \in T,\ t' = t + k$ a zobrazení

$$\xi_1 \equiv \xi_2 \equiv s_1(t) + kx_1(t), \quad s_1(a) = p.$$

Stavová proměnná s_1 slouží k uchování výstupu z předcházejícího kroku.

2.11.2 Prvky s deterministickým a nedeterministickým chováním

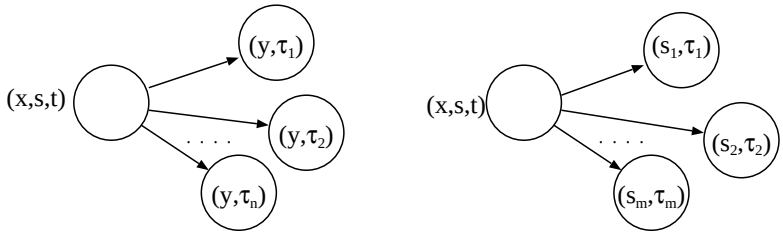
Uvažujeme opět chování prvku u_i ve tvaru:

$$\vec{y}(t') = \xi_1(\vec{x}(t), \vec{s}(t), t),$$

$$\vec{s}(t') = \xi_2(\vec{x}(t), \vec{s}(t), t),$$

$$\vec{x} \in \sigma_I(u_i),\ \bar{y} \in \sigma_O(u_i),\ \vec{s} \in \sigma_S(u_i),\ t, \tau \in T_i,\ t \leq \tau.$$

Prvek u_i má *deterministické* chování, jsou-li zobrazení ξ_1 a ξ_2 jednoznačná. *Nedeterministické* chování prvku u_i je obecně dáno víceznačností zobrazení ξ_1 nebo ξ_2. To znamená (viz *obr. 2.7*), že obrazem některého bodu $(\vec{x}, \vec{s}, t)$ definičního oboru zobrazení ξ_1 nebo ξ_2 není jediná dvojice $(\vec{y}, \tau)$ v zobrazení ξ_1 a jediná dvojice $(\vec{s}, \tau)$ v zobrazení ξ_2, ale množina všech možných obrazů $(\vec{y}, \tau)$, resp. $(\vec{s}, \tau)$.



Obrázek 2.7: Nedeterministické chování prvku v případě konečných oborů hodnot zobrazení ξ_1 a ξ_2

Zvláštním případem nedeterministických prvků jsou tzv. prvky se *stochastickým* chováním, jejichž chování lze popsat stochastickými závislostmi. V tomto případě je vstupní, resp. výstupní, resp. stavový prostor $\sigma_I(u_i)$, resp. $\sigma_O(u_i)$, resp, $\sigma_S(u_i)$ tvořen náhodnými vektory — tzv. vícerozměrnými náhodnými veličinami. Funkce ξ_1 a ξ_2 se nazývají náhodné funkce

nebo stochastické procesy. Podle tvaru časové množiny pak rozlišujeme diskrétní nebo spojité stochastické prvky. O diskrétních stochastických prvcích mluvíme, je-li časová množina T_i ve tvaru, který ilustruje *obr. 2.7.* Přechod z jednoho stavu prvku do druhého, případně výstup příslušející k danému stavu, je ohodnocen určitou pravděpodobností. O spojitých stochastických prvcích mluvíme tehdy, je-li časová množina T_i intervalem.

2.11.3 Systémy se spojitým chováním

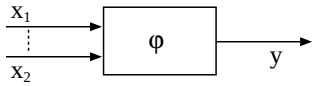
Systém S nazveme systémem se spojitým chováním nebo krátce spojitým systémem, mají-li všechny jeho prvky spojité chování. Spojité systémy jsou popisovány soustavami diferenciálních rovnic a algebraických rovnic tvaru

$$\begin{aligned} w_1' &= f_1(w_1, ..., w_n, x_1, ..., x_m, t) \\ &\vdots \\ w_1' &= f_n(w_1, ..., w_n, x_1, ..., x_m, t) \\ x_1 &= g_1(w_1, ..., w_n, x_1, ..., x_m, t) \\ &\vdots \\ x_1 &= g_m(w_1, ..., w_n, x_1, ..., x_m, t) \end{aligned}$$

s počátečními podmínkami

$$\begin{aligned} w_1(0) &= w_1^0 \\ &\vdots \\ w_n(0) &= w_n^0. \end{aligned}$$

Označíme-li graficky



prvek realizující spojitou funkci $y = \varphi(x_1, ..., x_n)$ a integrátor $y(t) = \int x(t)dt$, $y(0) = p$, pak můžeme tuto soustavu diferenciálních rovnic prvního řádu a algebraických rovnic chápat jako spojitý systém na *obr. 2.8.*

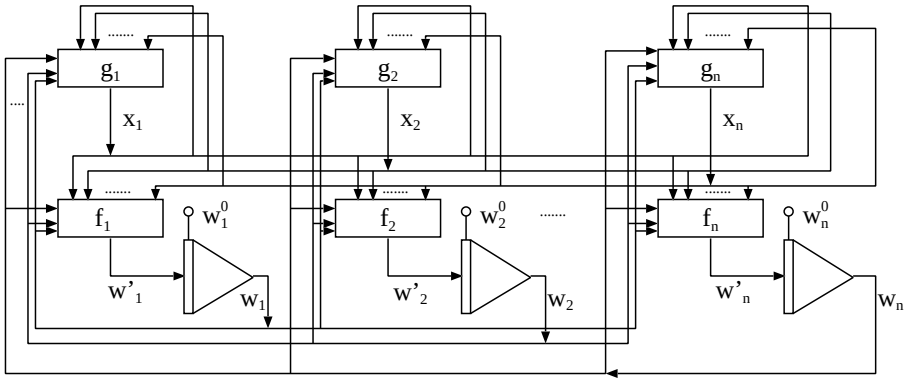
Příklad spojitého systému:

Uvažujeme systém tlumení kola automobilu (*obr. 2.9*), jenž je popsán diferenciální rovnicí druhého řádu

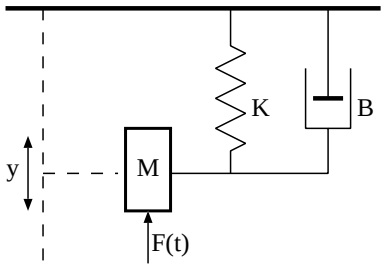
$$My'' + By' + Ky = F(t), \quad y'(0) = y(0) = 0 \tag{2.1}$$

kde

M je hmotnost kola,
K je tuhost pružiny,
B je tlumící faktor a
$F(t)$ je budicí síla.



Obrázek 2.8: Spojitý systém



Obrázek 2.9: Tlumení kola automobilu

Označme proměnnou z zrychlení, v rychlost, y dráhu kola automobilu. Diferenciální rovnici (2.1) převedeme na soustavu diferenciálních rovnic prvního řádu:

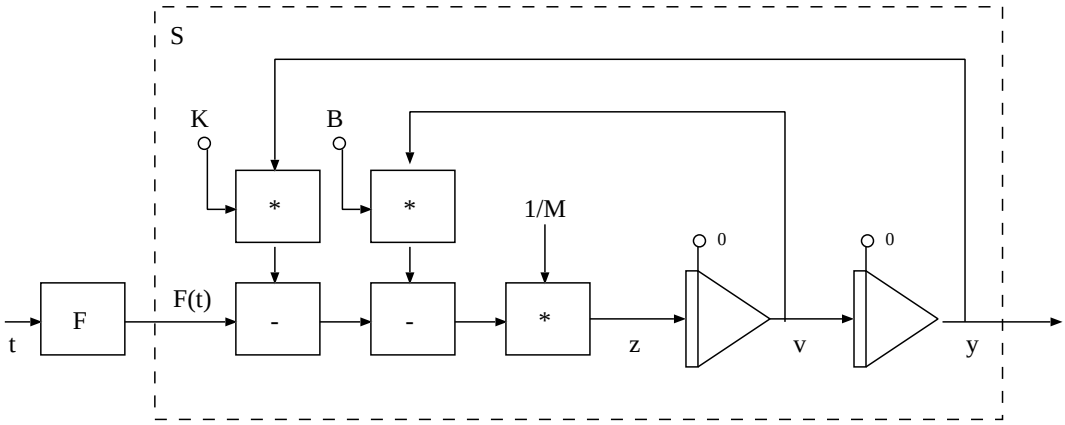
$$\begin{aligned} y' &= v \\ v' &= z \\ z &= -\frac{1}{M}(F(t) - Ky - Dv) \end{aligned}$$

$$y(0) = v(0) = 0$$

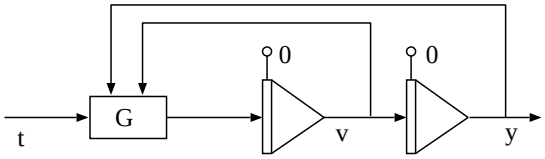
Prvek, realizující násobení, reprezentujeme symbolicky značkou $y = x_1 x_2$.
Prvek, realizující odečítání, reprezentujeme symbolicky značkou $y = x_1 - x_2$.

Uvažovaný spojitý systém pak vyjádříme schématem na *obr. 2.10.*

Takto uvedený systém chápeme jako systém otevřený, kde zdroj budicí síly $F(t)$ je součástí okolí systému. Budeme-li uvažovat budicí sílu $F(t)$ jako součást systému, pak lze systém popsat jednodušším homomorfním systémem na *obr. 2.11.*



Obrázek 2.10: Schéma spojitého systému — tlumení kola automobilu



Obrázek 2.11: Schéma systému tlumení kola automobilu, kde prvek G realizuje funkci

$$z = G(t, y, v) = \frac{1}{M}(F(t) - Ky - Bv)$$

2.11.4 Systémy s diskrétním chováním

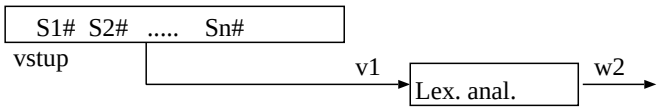
Systém S nazveme *systémem s diskrétním chováním* nebo diskrétním systémem, mají-li všechny jeho prvky diskrétní chování.

Příklad diskrétního systému

Jako příklad diskrétního systému uvažujeme lexikální analyzátor, jehož úkolem je rozeznávat "slova" určitého textu. Předpokládejme, že na vstupní pásce jsou umístěny řetězce znaků nad danou abecedou a že jsou oddělovány znakem # (*obr. 2.12*). Podněty uvažovaného systému budou reprezentovány znaky, tvořícími řetězce včetně koncového znaku #, reakcemi systému bude informace o tom, zda zpracovaná posloupnost vstupních znaků je slovem (výstup "1") nebo není slovem (výstup "0"). Časovou množinu T uvažovaného systému tvoří diskrétní okamžiky, v nichž jsou čteny jednotlivé znaky ze vstupní pásky.

Z teorie formálních jazyků je známo, že po transformaci vstupních znaků do tzv. abecedy terminálních symbolů, může být "jazyk slov" popsán regulární gramatikou a reprezentován obecně nedeterministickým konečným automatem. Na základě těchto poznatků můžeme tedy navrhnout lexikální analyzátor jako systém, jehož univerzum tvoří dva prvky (*obr. 2.12*) — prvek realizující transformaci vstupních znaků (prvek K) a konečný automat realizující

vlastní lexikální analýzu (prvek A).



Obrázek 2.12: Lexikální analyzátor

Ilustrujme nyní postup návrhu lexikálního analyzátoru na příkladě konkrétních slov — identifikátorů a celých čísel bez znaménka:

Vstupní abeceda systému $I = L \cup D \cup \{\#\}$ je dána sjednocením latinské abecedy, množiny arabských číslic a koncového znaku slov #.

Výstupní abecedou systému je binární abeceda $\{0, 1\}$ s významem:

0 — "není slovem"

1 — "je slovem"

Chování prvku K bude popsáno zobrazením

$$y_1 = \xi(x_1(t)) = \begin{cases} d & \text{je} - \text{li} \quad x_1(t) \in D \\ 1 & \qquad\quad x_1(t) \in L \\ \# & \qquad\quad x_1(t) = \# \end{cases}$$

kde $t \in T$.

Protože jazyk identifikátorů a celých čísel může být popsán regulární gramatikou $G = (\{d, 1, \#\}, \{S, N, I\}, P, S)$, kde množina přepisovacích pravidel P má tvar

$$S ::= I\#|N\#$$

$$I ::= 1|I1|Id$$

$$N ::= d|Nd$$

je snadné nalézt zobrazení ξ_1 a ξ_2 určující chování prvku A. Zobrazení ξ_1 je dáno maticí přechodů (tabulka 2.3)

Tabulka 2.3: Definice zobrazeni ξ_1

$\sigma_S(A)$ / $\sigma_I(A)$	S	I	N	E
d	N	I	N	E
1	I	I	E	E
#	E	S	S	S

$$\sigma_S(A) = \{S, I, N, E\},$$
$$\sigma_I(A) = \{d, 1, \#\},$$

kde E značí chybový stav. Zobrazení ξ_2 je neúplné zobrazení

$$y_2(t) = w_1(t) = \xi_2(x_2(t), s(t)), \quad x_2(t) \in \sigma_I(A), \quad s(t) \in \sigma_S(A) :$$

$$\xi_2(\#, I) = \xi_2(\#, N) = 1,$$

$$\xi_2(\#, E) = 0.$$

2.11.5 Systémy s kombinovaným chováním

Spojité a diskrétní systémy představují disjunktní třídy obecných systémů. Často je potřebné uvažovat systémy, které připouštějí koexistenci spojitých i diskrétních prvků. Tyto systémy budeme dále označovat jako *systémy kombinované*.

2.11.6 Systémy s deterministickým, nedeterministickým a stochastickým chováním

Systém označíme jako systém s deterministickým chováním, resp. deterministický systém, mají-li všechny jeho prvky deterministické chování. Systém, jenž obsahuje alespoň jeden nedeterministický prvek, označíme jako systém *nedeterministický*. Systém, jehož všechny nedeterministické prvky jsou stochastické, nazýváme systémem *stochastickým*.

Systémy spojité i diskrétní mohou mít deterministické, nedeterministické i stochastické chování. Z hlediska modelování je tato klasifikace systémů účelná, poněvadž v mnoha případech odráží určité etapy, kterými prochází vytvářený model systému v závislosti na množství informací, jež o modelovaném systému máme.

2.12 Citlivost systému na parametry

Při vytváření matematického modelu systému používáme různé *parametry*. Pokud vytváříme model existujícího systému, může jít o parametry zjištěné měřením, modelujeme-li systém, který se bude teprve realizovat, může jít o parametry předpokládané nebo odhadnuté. V obou případech je zapotřebí vědět, do jaké míry nepřesnost měření nebo odhad parametru ovlivňuje chování systému. Ke kvalitativnímu hodnocení vlivu změn parametrů systému se používá pojmu *citlivost*.

Označuje-li y_i nějakou výstupní proměnnou systému S a p_j parametr, jehož vliv vyšetřujeme, můžeme definovat *citlivost proměnné* y_i na *parametr* p_j *(koeficient citlivosti)* v určitém bodu x_0 vstupního prostoru a hodnotě parametru p_{j0} jako parciální derivaci

$$S_{y_i p_j} = \left.\frac{\partial y_i}{\partial p_j}\right|_{x_0, p_{j0}}$$

Problematiku citlivostní analýzy budeme ilustrovat na příkladě velice jednoduchého spojitého systému se vstupní veličinou x a výstupní y, jehož chování je popsáno diferenciální rovnicí

$$y' + ay = bx \tag{2.2}$$

s počáteční podmínkou $y(0) = y_0$.

V rovnici vystupují dva parametry a a b. Předpokládejme, že nás zajímá citlivost výstupní veličiny y na parametr a pro jednotkový skok vstupní veličiny x v okolí hodnoty parametru $a = a_0$. Rovnice

$$y' + a_0 y = b \qquad\qquad y(0) = y_0 \tag{2.3}$$

se nazývá *nominální rovnice.*

Nyní předpokládejme, že se změní hodnota parametru a o Δa. Následkem toho se změní i výstup y. *Odchylku(perturbaci)* označíme Δy. Není to obecně konstanta, nýbrž funkce času. Můžeme zapsat tzv. *rovnici přírůstkovou nebo rovnici odchylek (perturbací)*

$$(y + \Delta y)' + (a_0 + \Delta a)(y + \Delta y) = b \qquad \Delta y(0) = \Delta y_0$$

a řešit ji pro neznámou odchylku Δy. Dostaneme opět diferenciální rovnici 1.řádu

$$(\Delta y)' + a_0 \Delta y + \Delta a^y + \Delta a \Delta y + y' + a_0 y = b_0$$

odkud po dosazení z nominální rovnice (2.3) a linearizaci (zanedbání členu $\Delta a \Delta y$) obdržíme

$$(\Delta y)' + a_0 \Delta y = -\Delta a y \qquad \Delta y(0) = \Delta y_0 \tag{2.4}$$

Jejím řešením je časový průběh odchylky výstupu y při změně parametru a o Δa.

Jinou možností je zjišťovat průběh koeficientu citlivosti $S_{ya}(t)$. Mezi odchylkou Δy a koeficientem citlivosti S_{ya} platí pro malé hodnoty Δa vztah:

$$\Delta y(t, a) = S_{ya}(t, a) \Delta a \tag{2.5}$$

Derivováním rovnice (2.2) pro $x = 1$ podle parametru a získáme rovnici tvaru

$$\frac{\partial}{\partial a} y' + \frac{\partial}{\partial a}(ay) = 0 \tag{2.6}$$

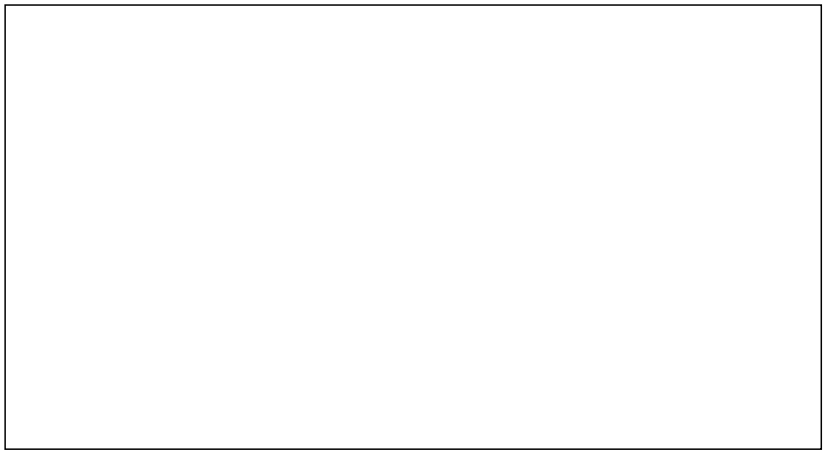
Protože parciální derivace výstupu y podle a značí koeficient citlivosti S_{ya} a parametr a není časově proměnný, můžeme rovnici (2.6) upravit na tzv. *rovnici citlivosti*, která má pro $a = a_0$ tvar

$$S'_{ya} + a_0 S_{ya} = -y \tag{2.7}$$

s počáteční podmínkou $S_{ya}(0) = S_{ya0}$.

Z rovnic (2.4) a (2.7) je vidět, že výstupní veličina y je zde ve funkci vstupu. Lze proto vytvořit simulační model, který bude současně řešit průběh výstupu y i koeficientu citlivosti S_{ya}, resp. odchylky Δy. Průběh odchylky a koeficientu citlivosti našeho jednoduchého příkladu pro $b = 1, a = 2$ a $\Delta a = 0.2$ jsou na *obr. 2.13.*

Na tomto příkladě jsme ukázali pouze některé základní pojmy, s nimiž se v oblasti citlivostní analýzy setkáváme. Lze samozřejmě provádět i složitější analýzy, jejichž cílem je zjištění citlivosti systému na několik parametrů současně, na vlastnosti podsystémů apod. Analýzu citlivosti lze provádět jak u spojitých, tak diskrétních systémů.



Obrázek 2.13: Průběh odchylky a koeficientu citlivosti rovnice $y' + 2y = 1$

Kapitola 3

Modelování spojitých systémů

3.1 Úvod

S modelováním spojitých systémů se setkáváme všude, kde je účelné reprezentovat změny stavových, vstupních a výstupních proměnných systému spojitými funkcemi; nejčastěji v teorii řízení, při modelování elektrických obvodů, ale také v netechnických oborech — při modelování biologických nebo ekologických problémů apod.

Pro popis dynamiky spojitých systémů používáme soustav diferenciálních rovnic. Poněvadž je aplikace analytických metod na řešení soustav diferenciálních rovnic, které popisují reálné dynamické systémy, velmi omezena, je použití počítačů prakticky jedinou cestou k jejich řešení.

Klasickým prostředkem pro modelování spojitých systémů byl analogový počítač. Potíže s přesností analogových výpočtů, jejich dokumentací a reprodukovatelností řešení, realizací nelinearit apod. vedly i v této oblasti k orientaci na počítače číslicové.

3.2 Popis spojitých dynamických systémů

V 2.11.3 jsme viděli, že spojitý dynamický systém lze popsat soustavou obyčejných diferenciálních a algebraických rovnic. Tento popis tvoří abstraktní matematický model vyjadřující vztah mezi vstupními, stavovými a výstupními veličinami. V závislosti na počtu vstupních proměnných a vlastnostech funkcí f_1 až f_n a g_1 až g_k soustavy můžeme klasifikovat různé třídy systémů. Pokusíme se přepsat soustavu do vektorového tvaru. Použijeme k tomu maticové operátory. Splňuje-li operátor $\mathbf{A}$ podmínku homogenity a distribučního zákona, tj. pro nějaké vektory $\mathbf{y}_1$ a $\mathbf{y}_2$ a konstanty a a b platí

$$\mathbf{A}(a\mathbf{y}_1 + b\mathbf{y}_2) = a\mathbf{A}\mathbf{y}_1 + b\mathbf{A}\mathbf{y}_2$$

nazývá se operátor lineární.

Systém, který lze popsat použitím lineárních operátorů, se nazývá *lineární*, v opačném případě *nelineární*. Lineární systémy jsou v praxi popsány lineárními a nelineární nelineárními diferenciálními rovnicemi. Systém, který má jeden vstup, se nazývá *jednoparametrový*, má-li vstupů více, *víceparametrový*.

Má-li systém m vstupů, k výstupů a n stavových veličin, můžeme přepsat soustavu z *kap. 2.11.3* pro lineární systém do tvaru vektorových rovnic

$$\frac{d}{dt}\mathbf{w}(t) + \mathbf{A}(t)\mathbf{w}(t) = \mathbf{B}(t)\mathbf{x}(t)$$

$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{w}(t) + \mathbf{D}(t)\mathbf{x}(t) \tag{3.1}$$

s vektorem počátečních podmínek $\mathbf{w}(0) = \mathbf{w}_0$, kde

$\mathbf{x}(t)$ je m-rozměrný vektor vstupních veličin,

$\mathbf{y}(t)$ je k-rozměrný vektor výstupních veličin,

$\mathbf{w}(t)$ je n-rozměrný vektor stavových veličin a

$\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ jsou lineární operátory typu (n, n), (n, m), (k, n), (k, m).

Jsou-li prvky operátorů $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ konstanty, jde o systém popsaný diferenciálními rovnicemi s konstantními koeficienty. Takový systém se nazývá *stacionární*.

Uvažujme jednoduchý příklad elektrického obvodu podle *obr.* **??**. Zajímá nás časový průběh napětí u_R. Jde o jednoparametrový spojitý systém se vstupní veličinou u a výstupní u_R. Za předpokladu, že hodnoty R_1, R_2, L_1, L_2 jsou konstantní, půjde zřejmě o systém lineární a stacionární.

Neuvažujeme-li vazbu mezi cívkami, lze na základě Ohmova a Kirchhoffových zákonů napsat rovnice

$$\begin{aligned} u &= R_1 i_1 + R_1 i_2 + L_1 \frac{di_1}{dt} \\ u &= R_1 i_1 + R_1 i_2 + R_2 i_2 + L_2 \frac{di_2}{dt} \\ u_R &= R_1 i_1 \end{aligned}$$

a po úpravě

$$\begin{aligned} \frac{d}{dt} i_1 + \frac{R_1}{L_1} i_1 + \frac{R_1}{L_1} i_2 &= \frac{1}{L_1} u \\ \frac{d}{dt} i_2 + \frac{R_1}{L_2} i_1 + \frac{R_1 + R_2}{L_2} i_2 &= \frac{1}{L_2} u \\ u_R &= R_1 i_1 \end{aligned} \tag{3.2}$$

Proudy i_1 a i_2 představují stavové veličiny s počátečními hodnotami $i_1(0) = i_2(0) = 0$. Soustavu (3.2) můžeme přepsat do vektorového tvaru

$$\frac{d}{dt}\mathbf{i} + \mathbf{A}\mathbf{i} = \mathbf{B}\mathbf{u}$$

$$u_R = \mathbf{C}\mathbf{i}$$

kde

$$\mathbf{A} = \begin{bmatrix} \frac{R_1}{L_1} & \frac{R_1}{L_1} \\ \frac{R_1}{L_2} & \frac{R_1+R_2}{L_2} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \frac{1}{L_1} \\ \frac{1}{L_2} \end{bmatrix} \qquad \mathbf{C} = [R_1\ 0]$$

Na základě soustavy (3.1) jsme schopni snadno vytvořit simulační model, protože soustava obsahuje pouze obyčejné diferenciální rovnice 1. řádu a algebraické rovnice, které jsme schopni řešit numerickými metodami. Původní matematický popis systému však může být v jiném tvaru (diferenciální rovnice vyšších řádů, parciální diferenciální rovnice apod.), proto musíme být schopni matematický model vždy do požadovaného tvaru upravit.

Metodika úprav matematických modelů do tvaru soustavy obyčejných diferenciálních rovnic 1. řádu byla významně ovlivněna metodikou modelování na analogových počítačích, protože i zde je k dispozici operační jednotka (integrátor) schopná řešit obyčejnou diferenciální rovnici 1. řádu. U analogových počítačů však bylo třeba navíc řešit v rámci přípravy modelu některé další úlohy jako je normalizace (transformace intervalu hodnot veličin do intervalu daného operačními jednotkami) a transformace času (vynucená omezenými frekvenčními vlastnostmi operačních jednotek).

3.2.1 Modelování stacionárních lineárních systémů

Uvažujme jednoparametrový stacionární lineární systém popsaný obyčejnou diferenciální rovnicí n-tého řádu s konstantními koeficienty tvaru

$$y^{(n)}(t) + a_{n-1}y^{(n-1)}(t) + ... + a_1y'(t) + a_0y(t) = b_mx^{(m)}(t) + ... + b_0x(t) \tag{3.3}$$

s počátečními podmínkami $y^{(i)}(0) = y_{i0}$ pro $i = 0, 1, ..., n-1$

Je vidět, že diferenciální rovnice může obecně mít na pravé straně derivace vstupní veličiny. Pro každý realizovatelný systém musí platit $n \geq m$. V obecném případě, kdy neznáme analytické vyjádření vstupní veličiny, neznáme ani její derivace. Typickým příkladem jsou regulační obvody, kdy jsou regulovaná soustava, případně její části, regulátor apod. popsány ve tvaru přenosu. Rovnice (3.3) by odpovídala přenosu, který má v Laplaceově transformaci tvar:

$$F(p) = \frac{b_mp^m + b_{m-1}p^{m-1} + ... + b_0}{p^n + a_{n-1}p^{n-1} + ... + a_0} \tag{3.4}$$

Ukážeme si dvě metody převodu diferenciální rovnice (3.3) na soustavu (3.1). Pro zjednodušení zápisu budeme používat operátor p, značící derivaci podle času a jeho mocniny p^2 pro druhou, p^3 třetí derivaci atd. Analogicky operátor $1/p$ bude značit integraci. Tento způsob zápisu byl rovněž často používán při programování analogových počítačů. Operátor p v tomto smyslu odpovídá Laplaceovu operátoru p při nulových počátečních podmínkách. V případech, kdy bude mít symbol p význam Laplaceova operátoru, explicitně na tuto skutečnost upozorníme. Rovnici (3.3) můžeme přepsat do tvaru:

$$p^ny + a_{n-1}p^{n-1}y + ... + a_1y + a_0y = b^mx + ... + b_0x \tag{3.5}$$

a) Metoda postupné integrace

Princip metody spočívá v osamostatnění nejvyšší derivace výstupní veličiny a postupné integraci takto vytvořené rovnice. Přitom postupně zavádíme stavové proměnné tak, aby byly řešením diferenciální rovnice 1. řádu. Uvažujme nejprve, že $n = m$, tj. že řád pravé i levé strany je stejný. Rovnici (3.5) můžeme přepsat do tvaru:

$$p^ny = b_np^nx + p^{n-1}(b_{m-1}x - a_{n-1}y) + ... + p(b_1x - a_1y) + (b_0x - a_0y) \tag{3.6}$$

Po integraci (aplikaci operátoru $1/p$) obdržíme:

$$p^{n-1}y = b_np^{n-1}x + p^{n-2}(b_{n-1}x - a_{n-1}y) + ... + (b_1x - a_1y) + \frac{1}{p}(b_0x - a_0y) \tag{3.7}$$

Nyní zavedeme stavovou proměnnou

$$w_1 = \frac{1}{p}(b_0x - a_0y) \tag{3.8}$$

Stavová proměnná w_1 je tedy řešením diferenciální rovnice

$$\frac{d}{dt}w_1 + a_0 y = b_0 x \quad \text{s počáteční podmínkou} \quad w_{10} \tag{3.9}$$

Po substituci rovnice (3.8) integrujeme rovnici (3.7) znovu a dostaneme:

$$p^{n-2}y = b_n p^{n-2}x + p^{n-3}(b_{n-1}x - a_{n-1}y) + ... + \frac{1}{p}(b_1 x - a_1 y + w_1) \tag{3.10}$$

Zavedeme stavovou proměnnou w_2

$$w_2 = \frac{1}{p}(b_1 x - a_1 y - w_1) \tag{3.11}$$

a analogicky postupujeme dál, až po poslední integraci obdržíme

$$y = b_n x + \frac{1}{p}(b_{n-1}x - a_{n-1}y + w_{n-1}) = b_n x + w_n \tag{3.12}$$

kde

$$w_n = \frac{1}{p}(b_{n-1}x - a_{n-1}y + w_{n-1}). \tag{3.13}$$

Výsledná soustava má tvar

$$\begin{aligned} w_1 &= \frac{1}{p}(b_0 x - a_0 y) \\ w_2 &= \frac{1}{p}(b_1 x - a_1 y + w_1) \\ &\vdots \\ w_n &= \frac{1}{p}(b_{n-1}x - a_{n-1}y + w_{n-1}) \\ y &= b_n x + w_n \end{aligned} \tag{3.14}$$

a po dosazení za y v diferenciálních rovnicích bychom dostali soustavu z *kap. 2.11.3*. Snadno bychom zapsali i vektorové rovnice a našli maticové operátory pro tvar podle (3.1). Blokové schéma, vyjadřující propojení bloků realizujících integraci, je pro metodu postupné integrace a podmínku $m = n$ na *obr.* **??**.
Uvažujme příklad systému, jehož přenos má v Laplaceově transformaci tvar:

$$F(p) = \frac{p^2 + 2p + 1}{p^2 + 3p + 2}$$

Odpovídající diferenciální rovnice zapsaná pomocí operátoru p :

$$p^2 y + 3py + 2y = p^2 x + 2px + x$$

Aplikace postupné integrace:

$$\begin{aligned} p^2 y &= p^2 x + p(3x - 2y) + (2x - y) \\ py &= px + (3x - 2y) + \frac{1}{p}(2x - y), \quad w_1 = \frac{1}{p}(2x - y) \\ y &= x + \frac{1}{p}(3x - 2y + w_1), \quad w_2 = \frac{1}{p}(3x - 2y + w_1) \end{aligned}$$

Výsledná soustava proto bude:

$$\begin{aligned} w_1 &= \frac{1}{p}(2x - y) \\ w_2 &= \frac{1}{p}(3x - 2y + w_1) \\ y &= x + w_2 \end{aligned}$$

Uvedený postup je použitelný i pro $m < n$. V takovém případě však lze postupovat i tak, že aplikujeme postupnou integraci do okamžiku, kdy odstraníme všechny derivace vstupní veličiny x a dál použijeme substituci, která je základem tzv. *metody snižování řádu derivace*. Vychází ze vztahu:

$$p^{i-1}y = \frac{1}{p}(p^i y) \tag{3.15}$$

Jestliže jsme postupnou integrací dostali poslední rovnici ve tvaru

$$p^{n-m}y = w_m, \tag{3.16}$$

budou mít další rovnice tvar:

$$\begin{aligned} w_{m+1} &= \frac{1}{p}(w_m) \\ w_{m+2} &= \frac{1}{p}(w_{m+1}) \\ &\vdots \\ w_n &= \frac{1}{p}(w_{n-1}) \\ y &= w_n \end{aligned} \tag{3.17}$$

Blokové schéma pro metodu snižování řádu derivace aplikovanou na rovnici tvaru

$$y^{(n)}(t) + a_{n-1}y^{(n-1)}(t) + ... + a_1 y'(t) + a_0 y(t) = b_0 x(t)$$

s počátečními podmínkami $y^{(i)}(0) = y_{i0}$ pro $i = 0, 1, ..., n-1$ je na *obr.* **??**.

Významnou vlastností metody snižování řádu derivace je, že stavové veličiny jsou totožné s tzv. *fázovými veličinami*, které odpovídají derivacím výstupu y. Z rovnic (3.17) plyne, že w_{n-1} odpovídá prvé derivaci y, w_{n-2} druhé až w_m $(n-m)$-té derivaci. Skutečnost, že fázové a stavové veličiny jsou totožné, má značné výhody. Patří mezi ně znalost průběhů derivací, které nás často také zajímají, navíc odpadají problémy při nenulových počátečních podmínkách, jak uvidíme později.

V případě, že diferenciální rovnice (3.3) nemá derivace na pravé straně, je nejjednodušší převod na soustavu diferenciálních rovnic 1. řádu právě použitím metody snižování řádu derivace.

b) Metoda snižování řádu derivace se zavedením pomocné proměnné

Viděli jsme, že metoda snižování řádu derivace používá pro převod na soustavu rovnic 1. řádu velmi jednoduchou substituci. Je však použitelná pouze za situace, kdy není v rovnici derivace vstupní veličiny. V obecném případě ji musíme kombinovat s další substitucí. Uvažujme popis systému ve tvaru přenosu (3.4). Vyjadřuje vztah mezi obrazy výstupu a vstupu v Laplaceově transformaci:

$$F(p) = \frac{y(p)}{x(p)} \tag{3.18}$$

Zavedeme-li pomocnou proměnnou z tak, že platí

$$F(p) = \frac{y(p)}{z(p)} \frac{z(p)}{x(p)} \tag{3.19}$$

můžeme přenos (3.4) rozepsat na dva přenosy:

$$\frac{z(p)}{x(p)} = \frac{1}{p^n + a_{n-1}p^{n-1} + ... + a_0} \tag{3.20}$$

$$\frac{y(p)}{z(p)} = b_m p^m + b_{m-1}p^{m-1} + ... + b_0 \tag{3.21}$$

Přenos (3.20) odpovídá diferenciální rovnici, jejíž zápis v operátorovém tvaru je

$$p^n z + a_{n-1}p^{n-1}z + ... + a_0 z = x \tag{3.22}$$

Protože jde o diferenciální rovnici bez derivací nezávisle proměnné, rozložíme ji snadno na n diferenciálních rovnic 1. řádu metodou snižování řádu derivace:

$$\begin{aligned} w_n &= p^n z = x - a_{n-1}p^{n-1}z - ... - a_0 z \\ w_{n-1} &= p^{n-1}z = \frac{1}{p}w_n \\ &\vdots \\ w_0 &= z = \frac{1}{p}w_1 \end{aligned} \tag{3.23}$$

Hodnota výstupu y je potom podle (3.21) určena algebraickou rovnicí

$$y = b_m w_m + b_{m-1}w_{m-1} + ... + b_0 w_0 \tag{3.24}$$

Diferenciální rovnice (3.23) a algebraická rovnice (3.24) již představují tvar podle *kap. 2.11.3.* Odpovídající blokové schéma pro předpoklad $m = n$ je na *obr.* **??**.

Poznámka:

V soustavě (3.23) jsme volili sestupnou indexaci stavových proměnných, aby hodnota indexu odpovídala řádu derivace, který představuje. Proměnná w_n ve skutečnosti není stavovou proměnnou, protože není řešením diferenciální, nýbrž algebraické rovnice. Zavedli jsme ji pro zjednodušení zápisu.

Pro ilustraci vyřešíme metodou snižování řádu derivace se zavedením pomocné proměnné příklad, který jsme řešili metodou postupné integrace. Přenos měl tvar:

$$F(p) = \frac{p^2 + 2p + 1}{p^2 + 3p + 2}$$

Zavedeme pomocnou proměnnou z, pro kterou řešíme rovnici:

$$\begin{aligned} p^2 z + 3pz + 2z &= x \\ w_2 &= x - 3w_1 - 2w_0 \\ w_1 &= \frac{1}{p}(w_2) \\ w_0 &= \frac{1}{p}(w_1) \end{aligned} \tag{3.25}$$

Dosazením za $p^2 z = w_2$, $pz = w_1$ a $z = w_0$ dostáváme pro výstup:

$$y = w_2 + 2w_1 + w_0$$

Tyto dvě metody jsou obecně použitelné pro náhradu obyčejné lineární diferenciální rovnice s konstantními koeficienty na soustavu n diferenciálních rovnic 1. řádu. Existují však i další metody, z nichž některé lze nalézt v [6].

Dosud jsme se blíže nezabývali otázkou počátečních hodnot. Systém, popsaný diferenciální rovnicí n-tého řádu, má pro hledané řešení zadáno n počátečních hodnot pro výstupní veličinu a její derivace po řád $n-1$. Počáteční podmínky představují počáteční podmínky pro fázové veličiny. Pokud při převodu na soustavu rovnic prvního řádu použijeme stavové proměnné, které nejsou totožné s fázovými, musíme mezi nimi najít vztah, abychom byli schopni určit počáteční hodnoty stavových proměnných, jak je zřejmé z blokových schémat na *obr.* **??** a **??**. Tato problematika však překračuje rámec skript. Některé převodní vztahy lze nalézt v [6].

Při úpravě abstraktního modelu lineárního stacionárního systému bychom měli dodržovat tyto zásady:

- Neobsahuje-li diferenciální rovnice derivace vstupní veličiny, použijeme metodu snižování řádu derivace.
- V opačném případě zavedeme pomocnou proměnnou nebo použijeme metodu postupné integrace. Při nenulových počátečních podmínkách je nutno nalézt vztah mezi stavovými a fázovými veličinami pro nastavení počátečních podmínek.

3.2.2 Modelování nestacionárních a nelineárních systémů

Nestacionární systémy jsou popsány diferenciálními rovnicemi s časově proměnnými koeficienty. Pokud jde o rovnici bez derivací vstupní veličiny, můžeme beze změn používat metodu snižování řádu derivace.

Uvažujme systém popsaný diferenciální rovnicí

$$\frac{d^2y}{dt^2} + \frac{1}{t}\frac{dy}{dt} + y = 0 \tag{3.26}$$

s počátečními podmínkami $y(0.02) = 0.99$, $y'(0) = -0.01$.

Rovnici přepíšeme použitím operátoru p a vyřešíme metodou snižování řádu derivace:

$$\begin{aligned} p^2y + \frac{1}{t}py + y &= 0 \\ w_1 = \frac{1}{p}(-\frac{1}{t}w_1 - y) & \\ y = \frac{1}{p}(w_1) & \end{aligned} \tag{3.27}$$

Pokud je systém popsán rovnicí s derivacemi vstupní veličiny, nelze použít metodu postupné integrace ani snižování řádu derivace v té podobě, jak jsme si je uváděli. Důvodem je skutečnost, že záleží na pořadí operací násobení a integrace, resp. derivování. Obecně platí

$$a(t)\frac{dy(t)}{dt} \neq \frac{d}{dt}(a(t)y(t)).$$

V takových případech se pro převod na soustavu diferenciálních rovnic 1. řádu používají speciální metody, které lze považovat za zobecněné metody postupné integrace a snižování řádu derivace. Jsou uvedeny např. v [7].

Podobná situace jako u nestacionárních systémů je i u nelineárních systémů. Na rozdíl od modelování na analogových počítačích má simulace na číslicovém počítači obrovskou výhodu v tom, že nečiní problémy modelovat nelineární závislosti vyjádřené analyticky nebo vhodně aproximovatelné.

Uvažujme náš jednoduchý elektrický obvod z *obr.* **??** s tím, že indukčnosti cívek L_1 a L_2 nebudeme považovat za konstantní, nýbrž za závislé na protékajícím proudu. Abstraktní model by měl tvar

$$u = R_1 i_1 + R_1 i_2 + \frac{d}{dt}\left(L_1(i) i_1\right) = R_1 i_1 + R_1 i_2 + \left(\frac{dL_1(i)}{di} i_1 + L_1(i)\right)\frac{di_1}{dt}$$

$$u = R_1 i_1 + R_1 i_2 + R_2 i_2 + \frac{d}{dt}\left(L_2(i).i_2\right) = R_1 i_1 + R_1 i_2 + R_2 i_2 + \left(\frac{dL_2(i)}{di} i_2 + L_2(i)\right)\frac{di_2}{dt}$$

$$u_R = R_1 i_1 \tag{3.28}$$

a po úpravě

$$\frac{di_1}{dt} + \frac{R_1}{\left(\frac{dL_1(i)}{di} i_1 + L_1(i)\right)} i_1 + \frac{R_2}{\left(\frac{dL_1(i)}{di} i_1 + L_1(i)\right)} i_2 = \frac{u}{\left(\frac{dL_1(i)}{di} i_1 + L_1(i)\right)}$$

$$\frac{di_2}{dt} + \frac{R_1}{\left(\frac{dL_2(i)}{di} i_2 + L_2(i)\right)} i_1 + \frac{R_1 + R_2}{\left(\frac{dL_2(i)}{di} i_2 + L_2(i)\right)} i_2 = \frac{u}{\left(\frac{dL_2(i)}{di} i_2 + L_2(i)\right)}$$

$$u_R = R_1 i_1 \tag{3.29}$$

Je zřejmé, že při přepisu do vektorových rovnic nedostaneme lineární maticový operátor, jde proto o systém nelineární. Pokud bychom aproximovali derivaci indukčnosti podle proudu vhodným vztahem, dostaneme soustavu nelineárních diferenciálních rovnic 1. řádu.

Pokud je systém popsán nelineární diferenciální rovnicí vyššího řádu bez derivací vstupní veličiny, provedeme převod na soustavu 1. řádu metodou snižování řádu derivace.

V některých případech lze u nelineárních systémů s výhodou využít dekompozice na lineární a nelineární část. Příkladem jsou systémy simulované v teorii řízení. Uvažujme jednoduchý regulační obvod podle *obr.* **??**.

$F_1(p)$ a $F_2(p)$ značí přenosy lineárních částí vyjádřené v Laplaceově transformaci a N značí nějakou nelinearitu. Nalezení analytického vyjádření vztahu mezi výstupem y a vstupem x by mohlo být značně obtížné, navíc výsledná diferenciální rovnice by byla nelineární. Podstatně jednodušší je využít dekompozice a upravovat samostatně rovnice pro přenos F_1 a F_2. K modelování nelinearit bývají simulační systémy zpravidla vybaveny odpovídajícími prostředky. V našem příkladě by byl výchozí tvar matematického modelu:

$$\begin{aligned} e &= x - y \\ \frac{g_1(p)}{e(p)} &= F_1(p) \\ g_2 &= N(g_1) \\ \frac{y(p)}{g_2(p)} &= F_2(p) \end{aligned} \tag{3.30}$$

3.2.3 Řešení soustav diferenciálních rovnic

Abstraktní matematický model systému popsaného soustavou diferenciálních rovnic vyššího řádu budeme upravovat použitím dříve uvedených metod. Postup ukážeme na soustavě

$$p^2y_1 + a_1py_1 + a_0y_1 + b_2p^2y_2 + b_1py_2 + b_0y_2 = k_0z$$

$$c_2p^2y_1 + c_1py_1 + c_0y_1^+p^2y_2 + d_1py_2 + d_0y_2 = 0 \tag{3.31}$$

s počátečními podmínkami $y_1(0) = y_{10}$, $y_2(0) = y_{20}$, $y_1'(0) = y_{10}'$, $y_2'(0) = y_{20}'$.
K úpravě použijeme metodu snižování řádu derivace tak, že prvou rovnici řešíme pro proměnnou y_1 a druhou pro y_2. Obdržíme soustavu

$$\begin{aligned} w_2 &= p^2y_1 = k_0z - a_1w_1 - a_0y_1 - b_2v_2 - b_1v_1 - b_0y_2 \qquad (3.32)\\ w_1 &= py_1 = \frac{1}{p}w_2 \\ y_1 &= \frac{1}{p}w_1 \\ v_2 &= p^2y_2 = -c_2w_2 - c_1w_1 - c_0y_1 - d_1v_1 - d_0y_2 \qquad (3.33)\\ v_1 &= py_2 = \frac{1}{p}v_2 \\ y_1 &= \frac{1}{p}v_1 \end{aligned}$$

Blokové schéma je na *obr.* **??**. Při řešení soustav diferenciálních rovnic může dojít k vytvoření tzv. *algebraických* nebo také *rychlých* smyček. Taková situace nastala i v našem příkladě u bloků pro rovnice (3.32) a (3.33) — smyčka je v obrázku zvýrazněna. Jde o smyčku, která neobsahuje žádný blok realizující integraci. Algebraické smyčky vyžadují při výstavbě simulačního modelu určité ošetření. Blíže si této problematiky všimneme v *kap. 3.5.*

3.2.4 Systémy popsané parciálními diferenciálními rovnicemi

Některé spojité systémy nelze popsat obyčejnými diferenciálními rovnicemi, nýbrž je nutno použít parciální diferenciální rovnice. Také tyto rovnice však potřebujeme pro řešení na počítači upravit do tvaru soustavy z *kap. 2.11.3.* Především je potřeba nahradit parciální diferenciální rovnici rovnicemi obyčejnými. K tomu se používá *metoda přímek*, jejíž princip spočívá v tom, že jedna nezávisle proměnná je chápána jako spojitá, zatímco ostatní jsou diskretizovány. Parciální derivace podle spojité proměnné tak přejde v bodech konstantních hodnot ostatních proměnných v obyčejnou derivaci a derivace podle ostatních proměnných jsou nahrazeny diferenčními vztahy. Metody přímek se dělí podle toho, která z nezávisle proměnných zůstává spojitá. Ukážeme si použití metody DSCT (*Discrete Space Continuous Time*).

Uvažujme vlnovou parciální diferenciální rovnici popisující kmitající strunu upevněnou na obou koncích podle *obr.* **??**. Předpokládejme, že sledujeme výchylku struny pouze ve směru osy y. Rovnice má tvar

$$\frac{\partial^2 y}{\partial t^2} = a\frac{\partial^2 y}{\partial x^2} \tag{3.34}$$

s počátečními podmínkami $y(x,0) = -\frac{4}{l^2}x^2 + \frac{4}{l}x$ a $y'(x,0) = 0$ a okrajovými podmínkami $y(0,t) = y(l,t) = 0$ — struna je na koncích upevněna.

V rovnici vystupují dvě nezávisle proměnné — čas a souřadnice x. Čas ponecháme spojitý a prostorovou souřadnici x diskretizujeme tak, že budeme hledat řešení pouze v $n+1$ bodech

intervalu $\langle 0, l \rangle$, jak je naznačeno na *obr.* **??**. Tyto body interval rovnoměrně rozdělí, přičemž vzdálenost dvojice bodů ve směru osy x bude Δx. Výsledkem řešení budou časové průběhy výchylky y v bodech x_i pro $i = 0, 1, ..., n$. Zavedeme si označení $y(x_i) = y_i$ pro $i = 0, 1, ..., n$. V bodech x_i je již souřadnice x konstantní, proto můžeme parciální derivaci podle času nahradit obyčejnou derivací podle času a parciální derivaci podle x nahradíme vhodným diferenčním vztahem, např.

$$\left.\frac{\partial^2 y}{\partial x^2}\right|_{x_i} = \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} \tag{3.35}$$

Vztah (3.35) lze získat z Taylorova rozvoje funkce y v bodech $x_i + \Delta x$ a $x_i - \Delta x$. Po dosazení do rovnice (3.34) obdržíme soustavu obyčejných diferenciálních rovnic 2. řádu

$$\frac{d^2 y_i}{dt^2} = \frac{a}{\Delta x^2}(y_{i+1} - 2y_i + y_{i-1}) \tag{3.36}$$

pro $i = 1, ..., n-1$, $y_0 = 0$, $y_n = 0$ (okrajové podmínky) s počátečními podmínkami $y_i(0) = -\frac{4}{l^2}x_i^2 + \frac{4}{l}x_i$ pro $i = 1, ..., n-1$.

Tuto soustavu již umíme upravit metodou postupné integrace na soustavu 1. řádu. Úpravu, případně nakreslení blokového schématu již necháváme na čtenáři.

3.3 Spojité simulační jazyky

Již v roce 1955 byl na počítači IBM 701 realizován program, jenž zde imitoval funkci analogového počítače. Po svém autorovi byl nazván SELFRIDGE. Od šedesátých let pak vznikala řada podobných simulačních systémů, v nichž se modely zapisovaly formou zakódovaného analogového schématu — propojením standardních funkčních bloků. V pozdějších spojitých simulačních systémech se stává jejich podmnožinou některý z obecných programovacích jazyků, nejčastěji FORTRAN. Na rozdíl od simulátorů analogového počítače je soubor standardních funkčních bloků rozšířen o logické operace, generátory náhodných čísel a prostředky pro vytváření nových funkčních bloků. Z této třídy simulačních jazyků jsou nejznámější: systém MIMIC (1965), vytvořený na počítači IBM 7090 a 7094, vývojová řada systémů DSL (Digital Simulation Language), zvláště DSL 10 (1965) na IBM 7010, vývojová řada systémů CSMP (Continuous System Modeling Program), nejrozšířenější CSMP 360 (1967) na počítačích IBM 360 a CSMP 1130 na počítačích IBM 1130.

Ve vývoji spojitých simulačních jazyků zaujímá důležité místo jazyk CSSL (Continuous System Simulation Language), jehož definice zahrnula přesná syntaktická pravidla pro zápis simulačního programu (v BNF) a sémantickou část, odrážející moderní trendy, které se ve vývoji spojitých simulačních jazyků objevily.

Také u nás byla od konce šedesátých let vytvořena řada spojitých simulačních jazyků od simulátorů analogového počítače až po jazyky srovnatelné se světovým standardem.

V poslední době je věnována pozornost spojitým simulačním jazykům, implementovaným na osobních počítačích třídy IBM PC XT/AT. Zde patří k nejzajímavějším systémům ACSL (Advanced Continuous Simulation Language), jenž byl původně implementován na střediskových počítačích a od r. 1986 je k dispozici i na osobních počítačích. Dalším důležitým systémem je Enhanced DESIRE, jehož autorem je prof. Korn z univerzity v Arizoně, jeden z předních světových odborníků ve spojité simulaci.V těchto systémech se objevují nové přístupy, umožňující programovat složité modely, efektivně plánovat simulační experimenty a důmyslně používat nejrůznější metody numerické integrace.

Přístupy moderních spojitých simulačních jazyků ukážeme v těchto skriptech prostřednictvím simulační knihovny SIMLIB v jazyce C++.

3.4 Numerické metody pro řešení spojitých systémů

Při vytváření modelů spojitých systémů na počítači se můžeme setkat s většinou numerických metod, které tvoří základní kurz numerické matamatiky. Poněvadž je abstraktním modelem spojitého systému soustava diferenciálních a algebraických rovnic, setkáme se vždy s numerickými metodami pro jejich řešení. Proto jsou tyto metody také součástí spojitých simulačních systémů.

Pro řešení algebraických a transcendentních rovnic je v simulačním jazyce zpravidla postačující jediná metoda, nejčastěji metoda Newtonova nebo iterační. Se základními numerickými metodami pro řešení algebraických a transcendentních rovnic jsme se v dostatečném rozsahu seznámili v předmětu numerické metody a matematická statistika, proto se zde nebudeme k těmto metodám samostatně vracet. Na druhé straně však různorodost modelovaných systémů vyžaduje, aby byl spojitý simulační jazyk vybaven řadou univerzálních i speciálních metod pro numerickou integraci. Kvalifikované využívání moderního spojitého simulačního jazyka zahrnuje proto také výběr vhodné numerické metody řešení diferenciálních rovnic a vyžaduje hlubší znalost těchto metod [14].

V této kapitole budeme diskutovat a srovnávat ty numerické metody, které jsou obvykle v simulačních jazycích začleněny a zaměříme se na ty aspekty, jež nemusí být významné při jednorázovém použití metody, ale v rámci komplikovaných simulačních výpočtů v simulačním jazyce podstatně ovlivňují efektivnost simulace. Seznámíme se také s metodami pro řešení diferenciálních rovnic se značným rozptylem časových konstant.

3.4.1 Základní pojmy

Zabývejme se soustavou m obyčejných diferenciálních rovnic

$$\begin{aligned} y_1'(t) &= f_1(t, y_1, ..., y_m) \\ y_2'(t) &= f_2(t, y_1, ..., y_m) \\ &\vdots \\ y_m'(t) &= f_m(t, y_1, ..., y_m) \end{aligned}$$

s počátečními podmínkami

$$\begin{aligned} y_1(t_0) &= y_0^1 \\ y_2(t_0) &= y_0^2 \\ &\vdots \\ y_m(t_0) &= y_0^m \end{aligned}$$

Soustavu diferenciálních rovnic budeme zkráceně zapisovat ve tvaru

$$y' = f(t, y) \tag{3.37}$$

$$y(t_0) = y_0 \tag{3.38}$$

kde y, y', y_0, f značí m-rozměrné vektory.

Předpokládejme, že jsou splněny *podmínky existence* a *jednoznačnosti* řešení v $m+1$-rozměrném oboru

$$M = I \times D$$

kde I je interval reálných čísel a D je otevřená oblast m-rozměrného euklidovského prostoru E^m o souřadnicích $y_1, ..., y_m$.

Numerickým řešením soustavy (3.37), (3.38) rozumíme posloupnost y_i hodnot

$$y(t_0), y(t_1), ..., y(t_k)$$

které odpovídají hodnotám $t_i, i = 0, 1, ..., k$ nezávisle proměnné t v intervalu $\langle t_0, t_k \rangle$. Výraz

$$t_{i+1} - t_i = h$$

je *integrační krok*; hodnoty $y(t_i)$ numerického řešení budeme značit y_i na rozdíl od hodnot *exaktního řešení* $Y_i = Y(t_i)$.

Cílem numerických metod řešení diferenciálních rovnic je nalezení numerického řešení soustavy (3.36), (3.37).

Má-li být numerická metoda použitelná, je nutné, aby posloupnost $\{y_i\}$ konvergovala pro $h \to 0$ k exaktnímu řešení $Y(t)$. *Konvergencí* zde rozumíme existenci limity posloupnosti $\{y_i\}$ pro $h \to 0$, $i \to \infty$, ale $i.h = t$ zůstává pevné.

Rozlišujeme dva základní typy metod numerického řešení soustavy diferenciálních rovnic (3.37), (3.38):

1. Metody, kde hodnoty funkce $f(t, y)$ se počítají jen v bodech $[t_i, y_i]$, (kde y_i je hodnota numerického řešení v bodě $t = t_i$), jsou zastoupeny *vícekrokovými metodami*.

2. Metody, kde se hodnoty funkce $f(t, y)$ počítají i mezi jednotlivými body $[t_i, y_i]$. Jsou zastoupeny *jednokrokovými metodami* — metody typu Runge-Kutta.

Poznámka:

Oba typy metod používají pouze první derivace řešení y. Existují metody, které používají i vyšších derivací. Hodnotu první derivace obdržíme prostým dosazením $[t_i, y_i]$ do (3.37). Vyšší derivace však nelze jednoduchým způsobem získat, neboť předpokládáme, že funkce $f(t, y)$ není obecně analyticky vyjádřena.

V obou případech je posloupnost hodnot $\{y_i\}$ výsledkem postupné extrapolace, přičemž již samotné výchozí body jsou zatíženy lokální chybou E (location error).

Lokální chyba — chyba jednoho kroku E

$$E = ET + ER$$

kde

ET je *chyba metody* (tj. chyba zanedbávací — truncation error), způsobená respektováním pouze konečného počtu členů Taylorova rozvoje,

ER je *chyba zaokrouhlovací* (rounding error), je dána omezenou délkou slova v počítači.

Velikost chyby zanedbávací bývá často udávána řádově v porovnání s mocninou kroku integrace h. Výrazem $o(h^i)$ označujeme, že chyba ET je řádu h na i-tou (z angličtiny **o**rder = řád).

Chyba jednoho kroku však ovlivňuje také výsledky kroků následujících — tento jev se studuje obvykle pod hlavičkou *stability*.

Metoda se nazývá *absolutně stabilní* pro daný krok h a danou diferenciální rovnici, jestliže chyba vzniklá při výpočtu y_n, se nezvětší v následujících hodnotách řešení y_k, $k > n$.

K vyšetřování absolutní stability se používá "testovací rovnice" $y' = \lambda y$, λ =konstanta. Množinu hodnot $\tilde{h} = h\lambda$, pro něž je metoda absolutně stabilní, nazýváme *oborem absolutní*

stability. (Poznamenejme, že λ může být i komplexní číslo a pak jde o obor rovinný.) Je možné najít téměř tolik definicí stability, kolik je prací o metodách numerické integrace.

Nepřesnost výsledku po n krocích je charakterizována akumulovanou chybou. *Akumulovaná chyba* po n krocích je definována výrazem

$$\epsilon_n = Y_n - y_n.$$

Zhodnocení metod provedeme z hlediska těchto kritérií:

1. Přesnosti metody — chyba lokální a prostředky jejího vyhodnocení.
2. Stability metody.
3. Časové náročnosti — spotřeby strojového času.
4. Nároku na paměť počítače.
5. Využitelnosti.

Pořadí kritérií nemusí odpovídat jejich důležitosti, např. někdy může být požadována velká rychlost i za cenu menší přesnosti metody.

Máme k dispozici *dvě kvalitativně odlišné skupiny metod*, které mají své specifické výhody i nevýhody.

3.4.2 Základní problémy implementace metody

Pod pojmem implementace metody rozumíme nejen implementaci jejího aproximačního vzorce, ale také některých dalších algoritmů, např. algoritmu výběru délky příštího integračního kroku, rozhodovacího kritéria, apod. *Implementovaná metoda* je tedy kompletním programem, který má tyto části:

1. *Aproximační vzorec*, jenž tvoří prostředky metody pro výpočet numerického řešení v bodě t_{n+1} v závislosti na integračním kroku h a předchozích aproximacích.
2. *Odhad lokální chyby*, na který navazuje rozhodovací kritérium. Často se používá odhadu lokální chyby podle Richardsona, který spočívá v porovnání řešení a krokem h a $2h$.
3. *Rozhodovací kritérium*, jehož prostřednictvím se rozhoduje, zda se řešení přijme, nebo zda se bude integrační krok opakovat. Rozhodovací kritérium může být založeno např. na podmínce:
$$|\text{odhad lokální chyby}| < \frac{3}{4}h$$
4. *Strategie volby integračního kroku* h, jež představuje způsob výběru velikosti příštího integračního kroku v závislosti na výsledcích rozhodovacího kritteria. Velmi často používanou strategií je půlení a dvojnásobení integračního kroku. Musíme však zabezpečit, aby nevznikly oscilace velikosti kroku typu $...h, 2h, h, 2h...$ Další aspekty, které bývají v implementované metodě zahrnuty, jsou uvedeny podrobně dále.

Ve speciálním případě tato definice zahrnuje i metody, pracující s pevným krokem integrace. V tomto případě jde o triviální strategii výběru (krok se nemění) a rozhodovací kritérium přijme každé řešení.

Rozebereme nyní podrobněji body 2 – 4:

Odhad lokální chyby

V úvodu jsme definovali lokální chybu jako součet chyby zanedbávací a chyby zaokrouhlovací

$$E = ET + ER$$

Při odhadech lokální chyby se většinou omezujeme na odhad chyby zanedbávací. Toto omezení je vzhledem k délkám paměťového slova existujících počítačů a vzhledem k možnosti počítání v dvojnásobné aritmetice oprávněné. Pokusy kompenzovat zaokrouhlovací chyby, jak je tomu např. v Gillově modifikaci metody Runge-Kutta, jsou řídkým zjevem.

Metody prediktor-korektor používají k odhadu lokální chyby vztahu

$$E = K\,|p_{n+1} - c_{n+1}| \tag{3.39}$$

Pokud slouží odhad velikosti chyby pouze pro řízení délky integračního kroku, nikoliv k opravě numerické aproximace, volí se běžně $K = 1$.

U metod jednokrokových je situace s odhadem lokální chyby poněkud horší. S výjimkou Mersonovy modifikace metod Runge-Kutta, která obsahuje předpis pro výpočet odhadu chyby, je jedině možný, i když časově náročnější, odhad chyby podle Richardsona. (Tohoto odhadu lze samozřejmě použít i u metod vícekrokových.)

Tyto způsoby odhadu chyby postihují, jak již bylo řečeno, chybu zanedbávací. V případě, že bychom volili extrémně malý krok, mohla by zaokrouhlovací chyba převýšit chybu zanedbávací. Taková situace však může vzniknout jen tehdy, jestliže vzhledem k řádu metody žádáme příliš vysokou přesnost.

Rozhodovací kritérium

Pro rozhodovací kritérium je obtížné vytvořit obecný předpis. V podstatě se jedná o sestavení algoritmu, který rozhoduje o tom, zda se řešení přijme, nebo se bude řešení opakovat a současně rozhodne, jakým způsobem se bude opakovat. Jádrem tohoto algoritmu je podmínka, kladená na velikost lokální chyby.

Při řešení soustav diferenciálních rovnic oceňujeme lokální chybu normou vektoru E. Prakticky se používá dvou vektorových norem:

$$||E|| = \max_i |E_j|, \quad j = 1, 2, ..., m \tag{3.40}$$

$$||E|| = \sum_{j=1}^{m} C_j |E_j|, \quad C_j > 0 \tag{3.41}$$

kde m je počet rovnic soustavy.
(Druhý vzorec je vhodný v případě, že chceme zvýraznit vliv některých složek chyby. Platí-li $\sum_{j=1}^{m} C_j = 1$, nazýváme C_j váhové koeficienty.)
Pro ilustraci uveďme často používaný tvar jednoduchého rozhodovacího kritéria:

a) Jsou dány dvě konstanty m_1, m_2. (Obecně by to mohly být funkce závislé na velikosti kroku h.) Nechť $m_1 < m_2$.

b) Je-li $m_2 < ||E||$, řešení se nepřijme. Integrační krok se opakuje s menším krokem h.

c) Je-li $m_1 \leq ||E|| \leq m_2$, řešení se přijme a výpočet pokračuje s nezměněným krokem.

d) Je-li $||E|| < m_1$, řešení se přijme a je možno zvětšit krok h.

Strategie výběru integračního kroku h

V této části výpočtu je třeba vybrat vhodnou velikost kroku h, se kterou se bude určovat další aproximace. Přitom je třeba brát v úvahu některé omezující podmínky; např. během výpočtu je třeba tisknout hodnoty řešení s konkrétní délkou intervalu tisku. Pak musíme brát ohled na tuto skutečnost a krok řídit tak, aby se v bodech tisku aproximace skutečně počítaly — tj. *krok integrace je shora omezen velikostí intervalu tisku.*

Nejčastější strategie výběru nového integračního kroku h je půlení a dvojnásobení. Krok h se půlí pokaždé, když podle rozhodovacího kritéria nelze spočítanou aproximaci řešení pokládat za dostatečně přesnou. Na druhé straně, podmínka pro malou lokální chybu $||E|| < m_1$ bývá zřídka kdy dostatečná pro okamžité zdvojení kroku. Snahou je nepřipustit příliš častou změnu kroku. V každém případě by se totiž mohlo stát, že časová úspora, vzniklá počítáním s delším krokem, by mohla být menší, než ztráta, která vznikne příliš častou manipulací s krokem. (Zejména u vícekrokových metod, kde každé půlení kroku vyžaduje nové odstartování výpočtu.) V extrémním případě by mohla vzniknout oscilace typu $..., h, 2h, h, 2h, ...$, při které výpočet zůstává stále v témže bodě.

Těmto nežádoucím případům se dá zabránit jednak dostatečně velkým poměrem m_2/m_1 mezi pro odhad lokální chyby, jednak podmínkou, že ke zdvojení kroku dojde pouze tehdy, platila-li relace $||E|| < m_1$ v k předcházejících krocích po sobě.

Strategie výběru kroku h bude dále obsahovat způsob odvození počátečního kroku, dosažení bodu tisku bez zvláštní manipulace s krokem a požadavek nepřekročení maximální a minimální velikosti kroku.

Pro zmenšení zaokrouhlovací chyby je u počítačů s dvojkovou aritmetikou výhodné volit počáteční hodnoty h jako celá záporné mocniny základu 2:

$$h = 2^{-r}, \quad r = 0, 1, 2, ... \tag{3.42}$$

tedy např. hodnoty $h = 1, 0.5, 0.25, 0.125$ atd. Mantisa takových čísel má při zobrazení v pohyblivé řádové čárce v normalizovaném tvaru jedničku jenom v nejvyšším bitu, kdežto všechny bity napravo jsou nulové.

3.4.3 Jednokrokové metody

Svůj název dostaly podle toho, že k výpočtu hodnoty y_{n+1} stačí znát hodnotu y_n. (Základní jednokrokovou metodou je Taylorův rozvoj řešení y v bodě y_n, který je pro naše účely zcela nevhodný.)

Základem všech Runge-Kuttových metod je vyjádření rozdílů mezi hodnotami řešení y v bodech t_{n+1} a t_n ve tvaru

$$y_{n+1} - y_n = \sum_{i=1}^{p} w_i k_i \tag{3.43}$$

kde w_i jsou konstanty a

$$k_i = h.f(t_n + a_i h, y_n + \sum_{j=1}^{i-1} b_{ij} k_j), \quad i = 1, ..., p \tag{3.44}$$

kde $h = t_{n+1} - t_n$ a a_i, b_{ij} jsou konstanty, přičemž $a_1 = 0$.
Metoda se nazývá p-hodnotová (používá p hodnot funkce $f(t, y)$).

Konstanty w_i, a_i, b_{ij} jsou spočteny tak, aby získané řešení souhlasilo s Taylorovým rozvojem v bodě $[t_n, y_n]$ až do P-té mocniny kroku h včetně. Metodu pak nazýváme *Runge-Kuttovou metodou řádu P.* Lze dokázat, že obecně platí $P \leq p$. Pro $p \leq 4$ platí $P = p$, tj.

p-hodnotová metoda je metodou p-tého řádu. Budeme se zabývat jen případy $P = 2, 3, 4$, proto nadále lze užívat označení p místo P.

Obor absolutní stability metod Runge-Kutta je určen nerovností

$$\left|1 + \tilde{h} + \frac{\tilde{h}^2}{2!} + ... + \frac{\tilde{h}^p}{p!}\right| < 1 \tag{3.45}$$

kde $\tilde{h} = h\lambda$, λ je komplexní číslo.

Chyba Runge-Kuttových metod

Odhad chyb u těchto metod je složitější, než u metod vícekrokových. Zanedbávací chyba metody druhého řádu je $ET = o(h^3)$, u metody čtvrtého řádu $ET = o(h^5)$. Pro odhad lokální chyby nelze použít Milneova principu. Jednou z možností odhadu přesnosti výpočtu je dvojí výpočet s krokem h a $2h$ — tj. *metoda polovičního kroku*. Pro odhad chyby při kroku $2h$ dostaneme

$$^{(2)}ET = \frac{^{(2)}y - ^{(1)}y}{2^p - 1} \tag{3.46}$$

kde $^{(2)}y$ resp. $^{(1)}y$ značí aproximaci řešení s krokem $2h$ resp. h a p je řád metody. (Odhadu (3.46) lze použít i pro vícekrokové metody. Z hlediska rychlosti výpočtu je použití vztahu (3.46) značnou ztrátou.)

Ve snaze dosáhnout vhodného odhadu chyby byly odvozeny speciální metody typu Runge-Kutta. Například *Mersonova metoda*, která je 4. řádu. Tento odhad vyžaduje další výpočet funkce $f(t, y)$. Merson přidal k soustavě rovnic (3.43) další rovnici, která umožňuje odhad chyby. Odvozený odhad je však správný pouze pro rovnice typu $y' = at + by + c$; kde a, b, c jsou konstanty.

Stabilita metod Runge-Kutta

Metody Runge-Kutta mají všechny ohraničený obor absolutní stability, definovaný nerovností (3.45). Obor absolutní stability se zvětšuje s rostoucím řádem. Uveďme si pro názornost obrázek, který ilustruje oblasti absolutní stability metod Runge-Kutta i metod vícekrokových (*obr. 3.1*). Křivky jsou zakresleny pro $h = 1$.

Příklady metod typu Runge-Kutta

Pro jednoduchost označme formálně

$$f_n = f(t_n, y_n).$$

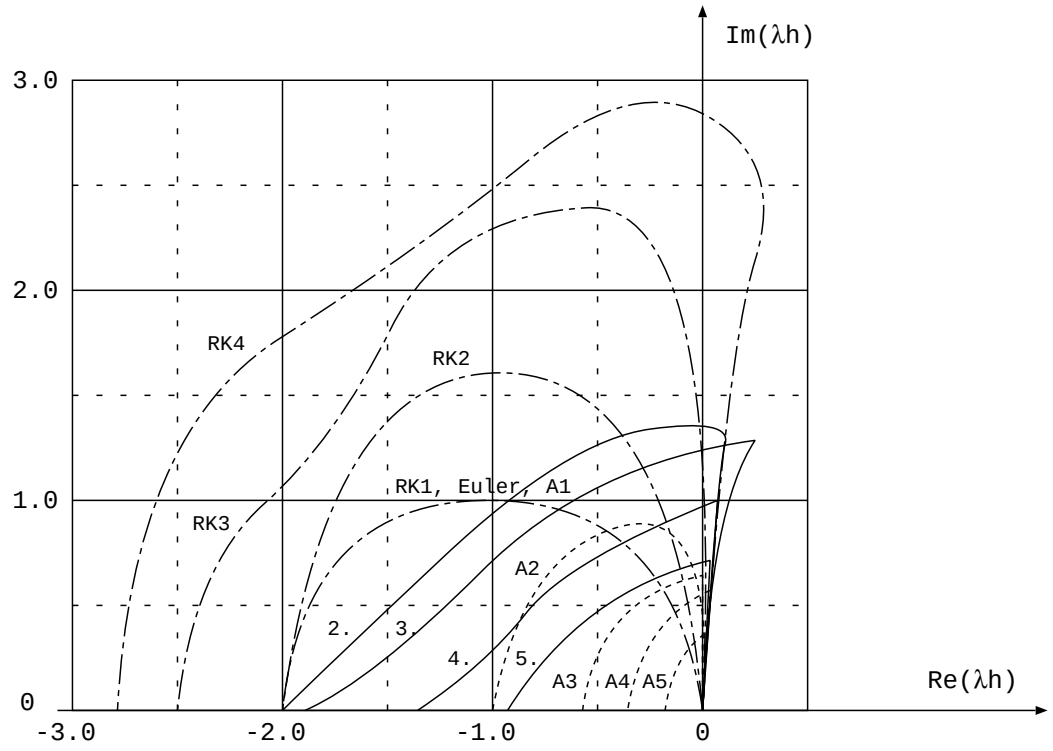
Metody druhého řádu

$$y_{n+1} = y_n + h.f(t_n + \frac{h}{2}, y_n + \frac{h}{2}f_n) \tag{3.47}$$

$$y_{n+1} = y_n + \frac{h}{2}(f_n + f(t_n + h, y_n + hf_n)) \tag{3.48}$$

Je-li $f(t, y)$ pouze funkcí času t, je metoda (3.48) lichoběžníkovým pravidlem.

Z *metod třetího řádu* uveďme pouze



Obrázek 3.1: Oblasti absolutní stability

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}(k_1 + 4k_2 + k_3) \\ k_1 &= h.f(t_n, y_n) \\ k_2 &= h.f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= h.f(t_n + h, y_n - k_1 + 2k_2) \end{aligned} \tag{3.49}$$

Je-li $f(t, y)$ pouze funkcí proměnné t, pak je uvedená metoda Simpsonovým pravidlem.

Pro *metody čtvrtého řádu* použijeme pro zvýšení přehlednosti raději formy tabulky (*tab. 3.1*).

Vztahy (3.43), (3.44) přepíšeme na tvar:

$$\begin{aligned} y_{n+1} &= y_n + w_1k_1 + w_2k_2 + w_3k_3 + w_4k_4 \\ k_1 &= h.f(t_n, y_n) \\ k_2 &= h.f(t_n + a_2h, y_n + b_{21}k_1) \\ k_3 &= h.f(t_n + a_3h, y_n + b_{31}k_1 + b_{32}k_2) \\ k_4 &= h.f(t_n + a_4h, y_n + b_{41}k_1 + b_{42}k_2 + b_{43}k_3) \end{aligned}$$

Tabulka 3.1: Odhad chyby pro metody Runge-Kutta

	standardní	tříosminová	Gill	Merson
w_1	1/6	1/8	1/6	1/2
w_2	1/3	3/8	$(1-\frac{1}{2}\sqrt{2})/3$	0
w_3	1/3	3/8	$(1+\frac{1}{2}\sqrt{2})/3$	$-3/2$
w_4	1/6	1/8	1/6	2
a_2	1/2	1/3	1/2	1/3
a_3	1/2	1/3	1/2	1/3
a_4	1	1	1	1/2
b_{21}	1/2	1/3	1/2	1/3
b_{31}	0	$-1/3$	$(\sqrt{2}-1)/2$	1/6
b_{32}	1/2	1	$1-\frac{1}{2}\sqrt{2}$	1/6
b_{41}	0	1	0	1/8
b_{42}	0	-1	$-\frac{1}{2}\sqrt{2}$	0
b_{43}	1	1	$1+\frac{1}{2}\sqrt{2}$	3/8

Všechny formule čtvrtého řádu dávají lokální chybu řádu h^5 a každá z uvedených má své specifické výhody. Např. standardní formule umožňuje díky řadě nulových koeficientů zjednodušený výpočet. Často uváděná modifikace Gillovy formule pro potlačení vlivu zaokrouhlovacích chyb má význam při krátké mantise zobrazení čísel a pro běžnou délku slova nemá její použití velké opodstatnění.

Mersonův odhad chyby předpokládá, že vedle hodnoty y_{n+1}, získané dosazením do (3.43), (3.44) dle *tab 3.1*, se provede ještě zpřesňující výpočet dle vzorce

$$\tilde{y}_{n+1} = y_n + \frac{1}{6}k_1 + \frac{2}{3}k_4 + \frac{1}{6}hf(t_n + h, y_{n+1})$$

a pro chybu ET pak platí

$$ET \approx \frac{1}{5}(y_{n+1} - \tilde{y}_{n+1}).$$

Zhodnocení metod Runge - Kutta

1. Metody Runge-Kutta nevyžadují dodatečných počátečních hodnot, krok integrace lze libovolně měnit a jejich použití na počítači je obecně velmi jednoduché.

2. Mají přibližně stejnou přesnost, často jsou přesnější, než metody prediktor-korektor stejného řádu. Přes velké množství publikovaných algoritmů se u metod Runge-Kutta

pro odhad chyby při proměnném kroku integrace nejvíce používá Richardsonova principu — přepočtu řešení s dvojnásobným krokem, který vyžaduje provedení alespoň o třetinu více strojových operací, než je třeba k vlastnímu výpočtu. Přepočet úlohy s dvojnásobným krokem je ovšem na druhé straně velmi účinným způsobem kontroly výpočtu.

3. Metody Runge-Kutta mají velký význam pro získání výchozích hodnot vícekrokových metod jak při záhájení výpočtu, tak při změně integračního kroku vícekrokových metod. Pro řešení na celém intervalu jsou někdy považovány za méně vhodné než metody vícekrokové — potřebují totiž v každém kroku tolik výpočtů funkční hodnoty $f(t, y)$, jaký je řád metody.

3.4.4 Vícekrokové metody

Tyto metody lze definovat vztahem

$$y_{n+1} = \sum_{i=0}^{r} a_i y_{n-i} + h \sum_{i=-1}^{r} b_i y'_{n-i} \tag{3.50}$$

kde a_i, b_i jsou konstanty.

Vztah (3.50) nazýváme *k-krokovou metodou*, jestliže k výpočtu hodnoty y_{n+1} používá k hodnot předchozích. Ve vztahu (3.50) je $k = r + 1$.

Je zřejmé, že těchto metod nelze použít k výpočtu prvého kroku, jsou tedy *nesamostartující*. K zahájení řešení je nutno prvých k hodnot spočítat některou z metod Runge-Kutta, nebo jinou jednokrokovou metodou.

Říkáme, že metoda (3.50) je řádu p, je-li přesná pro polynomy stupně p. V tom případě je nutné, aby bylo splněno následujících $p + 1$ podmínek:

$$\sum_{i=1}^{r} (-1)^s a_i + s \sum_{i=-1}^{r} (-1)^{s-1} b_i = 1, \quad s = 0, 1, ..., p \tag{3.51}$$

Splnění vzorce (3.51) pro $s = 0, 1$ zajišťuje *konzistenci metody*, tj., že metoda je přesná pro lineární polynomy. Vždy vyžadujeme, aby $p \geq 2$, tj. aby metoda byla konzistentní.

V každé zvláštní specializaci vzorce (3.50) mohou být některé z koeficientů a_i nebo b_i rovny nule, ale předpokládejme, že buď a_r nebo b_r není rovno nule. Dále se nebudeme zabývat konstrukcí jednotlivých metod, ale všimneme si jejich základních vlastností.

Je-li $b_{-1} = 0$, pak y_{n+1} je vyjádřeno jako lineární kombinace známých (z předchozího výpočtu) hodnot y_n a snadno se tedy vypočítá. Metody(3.50) pro $b_{-1} = 0$ se nazývají *explicitní metody* (také dopředné, přímé, prediktorového typu).

Je-li $b_{-1} \neq 0$, pak (3.50) je implicitní rovnicí pro y_{n-1}, protože $y'_{n+1} = f(t_{n+1}, y_{n+1})$ a obecně ji lze řešit jen iteračním postupem. Metody (3.50) s $b_{-1} \neq 0$ se nazývají *implicitní metody* (také iterační, nepřímé, korektorového typu).

Z velkého množství vícekrokových metod uveďme některé ve tvaru tabulky — *tab. 3.2.*

Názvy metod nejsou podstatné, bývá však zvykem odvolávat se na jednotlivé metody jménem, aniž by přesné znění formule bylo uvedeno.

Formule 7. – 10. představují známé Adamsovy prediktory (někdy také Adams-Bashforthovy formule), 15. – 18. Adamsovy korektory. Prostřednictvím řádu p a konstanty ve sloupci C lze snadno nalézt výraz pro odhad chyby metody:

$$ET \approx C h^{p+1} Y^{(p+1)}(\xi), \quad \xi \in \langle t_n, t_{n+1} \rangle. \tag{3.52}$$

Tabulka 3.2: Vícekrokové metody

číslo	a_0	a_1	a_2	a_3	a_4	a_5	b_{-1}	b_0	b_1	b_2	b_3	b_4	řád	C	název
1.	1	0	0	0	0	0	0	1	0	0	0	0	1	$\frac{1}{2}$	Eulerova metoda
2.	0	1	0	0	0	0	0	2	0	0	0	0	2	$\frac{1}{3}$	Milneho 2. řádu, Nyrstrômova
3.	0	0	1	0	0	0	0	$\frac{3}{2}$	$\frac{3}{2}$	0	0	0	2	$\frac{3}{4}$	
4.	0	0	0	1	0	0	0	$\frac{8}{3}$	$\frac{-4}{3}$	$\frac{8}{3}$	0	0	4	$\frac{14}{15}$	Milneho prediktor
5.	0	0	0	0	1	0	0	$\frac{55}{24}$	$\frac{5}{24}$	$\frac{5}{24}$	$\frac{55}{24}$	0	4	$\frac{95}{144}$	
6.	0	0	0	0	0	1	0	$\frac{33}{10}$	$\frac{-42}{10}$	$\frac{78}{10}$	$\frac{-42}{10}$	$\frac{33}{10}$	6	$\frac{41}{140}$	
7.	1	0	0	0	0	0	0	$\frac{3}{2}$	$\frac{-1}{2}$	0	0	0	2	$\frac{5}{12}$	Adams-Bashforthovy prediktory
8.	1	0	0	0	0	0	0	$\frac{23}{12}$	$\frac{-16}{12}$	$\frac{5}{12}$	0	0	3	$\frac{3}{8}$	
9.	1	0	0	0	0	0	0	$\frac{55}{24}$	$\frac{-59}{24}$	$\frac{37}{24}$	$\frac{-9}{24}$	0	4	$\frac{251}{720}$	
10.	1	0	0	0	0	0	0	$\frac{1900}{720}$	$\frac{-2774}{720}$	$\frac{2616}{720}$	$\frac{1274}{720}$	$\frac{251}{720}$	5	$\frac{95}{288}$	
11.	0	0	1	0	0	0	0	$\frac{21}{8}$	$\frac{-9}{8}$	$\frac{15}{8}$	$\frac{-3}{8}$	0	4	$\frac{243}{720}$	
12.	0	1	0	0	0	0	0	$\frac{8}{3}$	$\frac{-5}{3}$	$\frac{4}{3}$	$\frac{-1}{3}$	0	4	$\frac{232}{720}$	
13.	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0	$\frac{91}{36}$	$\frac{-63}{36}$	$\frac{57}{36}$	$\frac{-13}{36}$	0	4	$\frac{242}{720}$	
14.	-9	9	1	0	0	0	0	6	6	0	0	0	4	$\frac{72}{720}$	
15.	1	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0	2	$\frac{-1}{12}$	Lichoběžníkové pravidlo
16.	1	0	0	0	0	0	$\frac{5}{12}$	$\frac{8}{12}$	$\frac{-1}{12}$	0	0	0	3	$\frac{-1}{24}$	Adamsovy korektory
17.	1	0	0	0	0	0	$\frac{9}{24}$	$\frac{19}{24}$	$\frac{-5}{24}$	$\frac{1}{24}$	0	0	3	$\frac{-19}{720}$	
18.	1	0	0	0	0	0	$\frac{251}{720}$	$\frac{646}{720}$	$\frac{-264}{720}$	$\frac{106}{720}$	$\frac{-19}{720}$	0	5	$\frac{-3}{160}$	
19.	0	1	0	0	0	0	$\frac{1}{3}$	$\frac{4}{3}$	$\frac{1}{3}$	0	0	0	4	$\frac{-1}{90}$	Milneho korektor
20.	$\frac{9}{17}$	$\frac{9}{17}$	$\frac{-1}{17}$	0	0	0	$\frac{6}{17}$	$\frac{18}{17}$	0	0	0	0	4	$\frac{-3}{170}$	
21.	1	$\frac{1}{9}$	$\frac{-1}{9}$	0	0	0	$\frac{10}{27}$	$\frac{22}{27}$	$\frac{-8}{27}$	0	0	0	4	$\frac{-19}{810}$	
22.	$\frac{9}{8}$	0	$\frac{-1}{8}$	0	0	0	$\frac{3}{8}$	$\frac{3}{4}$	$\frac{-3}{8}$	0	0	0	4	$\frac{-1}{40}$	Hammingův korektor
23.	$\frac{9}{7}$	$\frac{-1}{7}$	$\frac{-1}{7}$	0	0	0	$\frac{8}{21}$	$\frac{2}{3}$	$\frac{-10}{21}$	0	0	0	4	$\frac{-17}{630}$	
24.	$\frac{9}{5}$	$\frac{-3}{5}$	$\frac{-1}{5}$	0	0	0	$\frac{2}{5}$	$\frac{2}{5}$	$\frac{-4}{5}$	0	0	0	4	$\frac{-1}{30}$	
25.	1	0	0	0	0	0	1	0	0	0	0	0	1	$\frac{-1}{2}$	Implicitní Eulerova metoda
26.	$\frac{4}{3}$	$\frac{-1}{3}$	0	0	0	0	$\frac{2}{3}$	0	0	0	0	0	2	$\frac{-2}{9}$	Curtiss-Hirschfelder
27.	$\frac{18}{11}$	$\frac{-9}{11}$	$\frac{2}{11}$	0	0	0	$\frac{6}{11}$	0	0	0	0	0	3	$\frac{-1}{44}$	
28.	$\frac{48}{25}$	$\frac{-36}{25}$	$\frac{16}{25}$	$\frac{-3}{25}$	0	0	$\frac{12}{25}$	0	0	0	0	0	4	$\frac{-7}{750}$	

Při praktickém použití určité metody vzniká otázka, jak vhodně volit krok h, abychom získali co největší přesnost. Na tuto otázku nelze dát uspokojivou odpověď. Částečně je tento problém řešen absolutní stabilitou metody.

V *tab. 3.3* jsou uvedeny *obory absolutní stability* některých formulí z *tab. 3.2.* Obory absolutní stability Adamsových prediktorů (metody č. 1, 7, 8, 9) a metod prediktor-korektor jsou znázorněny na *obr.* **??**.

Lichoběžníková metoda (č.15) má obor stability (při komplexním λ) celou polorovinu

Tabulka 3.3: Obory absolutní stability

formule číslo	1	7	8	9	15	16
obor stability	$(-2,0)$	$(-1,0)$	$(-0.556,0)$	$(-0.3,0)$	$(-\infty,0)$	$(-1.8,0)$

$Re\lambda < 0$. Takové metody nazýváme A-stabilní. (A-stabilita metody je požadována při řešení tzv. tuhých soustav — viz. dále.)

Milneho korektor (č. 19) není stabilní v případě $\partial f/\partial y < 0$. Proto nelze metodu doporučit, pokud není předem známo, že derivace $\partial f/\partial y$ je kladná. (Poznamenejme, že pro řád $p \geq 3$ je nutno k vyčíslení oblasti stability řešit charakteristickou rovnici p-tého řádu.)

Je zřejmé, že implicitní metody jsou pracnější, než explicitní (větší spotřeba strojového času pro stejný integrační krok). Na druhé straně jsou však přesnější a mají lepší vlastnosti pokud se týká stability (viz *obr.* **??**).

Kombinací explicitních a implicitních metod dostáváme metody prediktor-korektor. Pro první výpočet (predikci) y^p_{n+1} se užije formule explicitní a určíme derivaci $y'_{n+1} = f(t_{n+1}, y^p_{n+1})$. Implicitní formulí pak řešení zpřesňujeme, korigujeme — spočteme y^C_{n+1}. Dvojice metod se volí obvykle tek, aby měly stejný řád.

U metod prediktor-korektor vyžadujeme, aby zvolený prediktor měl malou lokální chybu a necitlivost na zaokrouhlení, U korektoru pak vyžadujeme co největší oblast stability (neboť ta určuje stabilitu celé metody) při zachování co nejmenší lokální chyby. Navíc je zde požadavek na dobrou konvergenci iterací, při kterých opakujeme výpočet korektoru tak dlouho, pokud nedosáhneme požadované vlastnosti. (Konvergence je zaručena za předpokladu $|b_{-1}hL| < 1$, kde L je Lipschitzova konstanta funkce $f(t,y)$. Příkladem je *Milneho metoda 4. řádu*, která používá prediktoru č. 4 a korektoru č. 19 z *tab. 3.2.* a *Hammingova metoda 4. řádu*, která používá prediktoru č.4 a korektoru č. 22 z *tab. 3.2.*

Milneho metoda je přesnější, ale je nestabilní pro $\partial f/\partial y < 0$. Hammingova metoda je sice méně přesná, ale stabilní pro $\partial f/\partial y$ kladné i záporné (oblast stability korektoru $h\partial f/\partial y \in (-8/3, 0)$.)

Nezbytnost iteračních výpočtů u metod prediktor-korektor způsobuje značné zvýšení spotřeby strojového času. Tento nedostatek odstraňuje metoda *prediktor-modifikátor-korektor.*

Za předpokladu, že prediktor a korektor mají řádově stejnou chybu ET, můžeme nalézt funkce $\varphi_1(p_n - C_n)$ resp. $\varphi_2(p_{n+1} - C_{n+1})$, kde $p_k - C_k$ je rozdíl predikované a korigované hodnoty. Tyto funkce explicitně vyjadřují odhad chyby ET prediktoru resp. korektoru. Vzorec

$$m_{n+1} = p_{n+1} + \varphi_1(p_n - C_n)$$

opravující predikovanou hodnotu řešení se nazývá *modifikátor.* Korektorem vypočítána aproximace je potom opravena hodnotou $\varphi_2(p_{n+1} - C_{n+1})$, takže výsledná aproximace je dána vztahem

$$y_{n+1} = C_{n+1} + \varphi_2(p_{n+1} - C_{n+1}).$$

Úprava tedy spočívá v tom, že iterace jsou nahrazeny modifikací prediktoru na základě znalosti chyby ET. Jako příklad uveďme *Hammingovu metodu* prediktor-modifikátor-korektor 4.řádu, která používá prediktoru č. 4 a korektoru č. 22 z *tab. 3.2*:

Prediktor: $p_{n+1} = y_{n-3} + \frac{4}{3}h(2y'_n - y'_{n-1} + 2y'_{n-2})$

Modifikátor: $m_{n+1} = p_{n+1} - \frac{112}{121}(p_n - C_n)$

$$m'_{n+1} = f(t_{n+1}, m_{n+1}) \tag{3.53}$$

Korektor: $C_{n+1} = \frac{9}{8}y_n - \frac{1}{8}y_{n-2} + \frac{3}{8}h(m'_{n+1} + 2y'_n - y'_{n-1})$

Výsledná aproximace řešení: $y_{n+1} = C_{n+1} + \frac{9}{121}(p_{n+1} - C_{n+1})$

Metody prediktor-korektor umožňují snadný odhad lokální chyby řešení v daném bodě t_n prostřednictvím rozdílu $(p_n - C_n)$. Na každém kroku vyčíslujeme hodnotu funkce $f(t, y)$ nejvýše dvakrát, nezávisle na řádu metody. Metody prediktor-korektor jsou nesamostartující — tato vlastnost je zvláště nepříjemná při změnách integračního kroku h.

3.4.5 Metody pro řešení tuhých ("stiff") soustav obyčejných diferenciálních rovnic

"Tuhou" soustavou diferenciálních rovnic rozumíme soustavu se značně rozdílnými časovými konstantami, tj. soustavu se značně rozdílnými vlastními čísly Jacobiho matice soustavy diferenciálních rovnic.

Označme J Jacobiho matici soustavy (3.37), jejíž prvky jsou $\partial f_i/\partial y_j$, $i, j = 1, ..., m$. Vlastní čísla matice J závisejí na čase t a jejich velikost se během integrace mění.

Tuhá a *stabilní* soustava (3.37) má tyto vlastnosti:

a) $Re\lambda_j < 0, \quad j = 1, ..., m$ pro všechna $t \in I$

b) $\max\limits_j |Re\lambda_j| \geq \min\limits_j |Re\lambda_j|, \quad j = 1, ..., m$

Zde λ značí vlastní číslo Jacobiho matice soustavy a $Re\lambda$ jeho reálnou část. Tuhost soustavy lze charakterizovat tuhostním poměrem $\max\limits_j |Re\lambda_j| : \min\limits_j |Re\lambda_j|$.

U většiny numerických metod vyžaduje stabilita metody zpravidla omezení na integrační krok h ve tvaru $|h\lambda| < K$, kde λ je maximální vlastní číslo (v absolutní hodnotě) Jacobiho matice a K je pevná konstanta. (Např. pro metodu Runge-Kutta čtvrtého řádu je $K \approx 2.8$, viz *obr.* **??**). U takových metod pak velké hodnoty vynucují velmi malý integrační krok.

Je tedy vhodné řešit tuhé soustavy takovými metodami, jejichž oblastí absolutní stability je celá levá polorovina — tj. $Re\lambda h < 0$. Takové metody se nazývají *A-stabilní*. Platí:

a) Explicitní vícekroková metoda nemůže být A-stabilní.

b) A-stabilní implicitní vícekroková metoda může být nejvýše druhého řádu a nejmenší lokální chybu má lichoběžníková metoda.

Implicitní metodou prvého řádu, která je A-stabilní, je metoda č. 25 z *tab. 3.2*

$$y_{n+1} = y_n + hf_{n+1}$$

tj.implicitní Eulerova metoda. Oblastí absolutní stability je celá rovina $h\lambda$ mimo jednotkový kruh se středem v bodě $[1, 0]$.

Implicitní metoda 2. řádu, která je A-stabilní, je *lichoběžníková metoda* č. 15 z *tab. 3.2*:

$$y_{n+1} = y_n + \frac{h}{2}(f_{n+1} + f_n).$$

Vzhledem k A-stabilitě lze volit krok integrace libovolně, ale složky řešení, odpovídající největšímu vlastnímu číslu (v abs. hodnotě), mohou pak být aproximovány nepřesně.

Řešíme-li tedy lichoběžníkovou metodou rovnici $y' = \lambda y$ $(Re\lambda < 0)$, dostaneme

$$y_{n+1} = \frac{y_n(2 + \lambda h)}{(2 - \lambda h)},$$

zatímco pro přesné řešení $Y(t)$ platí

$$Y(t_{n+1}) = Y(t_n)e^{\lambda h}.$$

Pak

$$\lim_{\lambda h \to \infty} e^{\lambda h} = 0 \quad \text{a} \quad \lim_{\lambda h \to \infty} \frac{(2+\lambda h)}{(2-\lambda h)} = -1.$$

To znamená, že $|y_n|$ bude konvergovat k nule velmi pomalu pro velmi tuhé soustavy.

Implicitní metoda Eulerova tento nedostatek nemá, poněvadž řešení testovací rovnice $y' = \lambda y$ $(Re\lambda < 0)$ je tvaru

$$y_{n+1} = \frac{y_n}{(1-\lambda h)} \quad \text{a} \quad \lim_{\lambda h \to \infty} (1-\lambda h)^{-1} = 0$$

Abychom mohli konstruovat pro řešení tuhých soustav diferenciálních rovnic vícekrokové metody řádu vyššího než dvě a používat je, byl silný požadavek A-stability metody nahrazen slabším požadavkem a vznikly tzv. *tuhostně stabilní metody.*
Metoda se nazývá tuhostně stabilní, existují-li kladná čísla a, b, c tak, že

- v oboru R_1 je absolutně stabilní
- v oboru R_2 dává přesné řešení rovnice $y' = \lambda y$ $(Re\lambda < 0)$.

Obory R_1, R_2 jsou zakresleny na *obr.* **??**.

Příkladem je *Gearova metoda*, která je však časově náročná v důsledku Jacobiho matice a inverze matice. Selhává např. při výpočtu inverzní matice pro její špatnou podmíněnost.

Dalším příkladem tuhostně stabilních metod jsou *metody Curtise a Hirschelfeldera.*
Metoda druhého řádu

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2}{3}hf_{n+1}$$

je uvedena v *tab. 3.2* pod číslem 26. Metody třetího a čtvrtého řádu jsou zde uvedeny pod čísly 27 a 28.

Obory absolutní nestability metod Curtis-Hirschfeldera ilustruje *obr.* **??**.

Pro tyto metody je konstanta a z definice tuhostní stability menší než 0.7, tj. $a < 0.7$.

V souvislosti s řešením tuhých soustav byly vypracovány také implicitní *metody Runge-Kutta*, které jsou A-stabilní. Jako příklad uveďme implicitní metodu třetího řádu:

$$y_{n+1} = y_n + \frac{h}{4}(K_1 + 3K_2)$$

$$K_1 = f(t_n, y_n + \frac{1}{4}hK_1 - \frac{1}{4}hK_2)$$

$$K_2 = f(t_n + \frac{2}{3}h, y_n + \frac{1}{4}hK_1 + \frac{5}{12}hK_2)$$

Tyto metody jsou méně výhodné než vícekrokové implicitní metody, neboť vyžadují na každém kroku řešení jisté soustavy nelineárních rovnic — jsou tedy časově náročné. (V uvedeném příkladě dvě rovnice pro neznámé K_1, K_2. Jsou-li ovšem K_1, K_2 m-rozměrné vektory, pak musíme řešit soustavu $2m$ nelineárních rovnic.)

V roce 1967 publikovali autoři Fowler a Warten speciální metodu pro řešení tuhých soustav diferenciálních rovnic. Metoda *Fowler-Wartenova* je vícekroková samostartující metoda s automatickou volbou integračního kroku. Nad ostatní známé metody pro řešení tuhých soustav diferenciálních rovnic vyniká tím, že nevyžaduje iterační výpočty.

Základní myšlenka, z níž autoři při odvozování integrační formule vycházeli, je tato: Řešení y_C diferenciální rovnice $y' = f(t, y)$ v bodě $(t + \xi)$ je dáno součtem dvou složek

$$y_C(t + \xi) = y_A(t + \xi) + y_p(t + \xi)$$

kde

$$y_A(t + \xi) = y_A(t) + \xi y'_A(t)$$

představuje asymptonickou část řešení $y_C(t + \xi)$ a

$$y_p(t + \xi) = y_p(t) + \frac{e^{\lambda\xi} - 1}{\lambda} y'_p(t)$$

je odchylka od asymptoty v důsledku lokálního chování řešení. Formule se shoduje s Taylorovým rozvojem řešení až do druhého řádu včetně, lze tedy její lokální chybu charakterizovat výrazem $o(h^3)$.

Označme y aproximaci řešení získanou v minulém kroku. Dále označme $h = t_{n+1} - t_n$ a $h_0 = t_n - t_{n-1}$. Algoritmus pro vyčíslení numerického řešení $y_{n+1_C} = y_C(t + h)$ je dán těmito vzorci:

1. $y'_A(t) = (y(t - h_0))/h_0$ aby byla metoda samostartující, klademe na počátku $y'_A = 0$.
2. $d_1 = y'(t) - y'_A(t)$
3. Integrace Eulerovou metodou, kde krok δ nepřevýší $h/4$: $y_p(t + \delta) = y(t) + \delta y'(t)$, kde $\delta \leq h/4$
4. $y'_p(t + \delta) = f(t + \delta, y_p(t + \delta))$
5. Vypočteme aproximaci 2. derivace řešení $y''(t)$ $d_2 = (y'_p(t + \delta) - y'(t))/\delta$
6. $$\lambda = \begin{cases} \frac{d_2}{d_1} & \ldots \text{ pro } d_1 \neq 0 \\ 0 & \ldots \text{ pro } d_1 = 0 \end{cases} \tag{3.54}$$
$$C_1 = \begin{cases} \frac{(e^{\lambda h} - 1)}{\lambda h} & \ldots \text{ pro } \lambda < 0 \\ 1 + \frac{\lambda h}{2} & \ldots \text{ pro } \lambda \geq 0 \end{cases} \tag{3.55}$$
$$C_0 = \begin{cases} e^{\lambda h} & \ldots \text{ pro } \lambda < 0 \\ 1 + \lambda h & \ldots \text{ pro } \lambda \geq 0 \end{cases} \tag{3.56}$$
7. Řešení $y_C(t + h)$ je nyní dáno vztahem
$$Y_C(t + h) = y(t) + h y'_A(t) + h C_1 d_1$$
8. Derivace řešení na konci integračního kroku
$$y'_C(t + h) = f(t + h, y_c(t + h))$$
Pokud se řešení y_C přijme, bude i derivace y'_C přijata za derivaci řešení v novém integračním kroku.
9. Odhad zanedbávací chyby řešení y_C
$$E = h(y'(t + h) - (y'_A(t) + C_0 d_1))$$

10. Odhad zanedbávací chyby při integraci Eulerovou metodou (krok 3):

$$E_p = \delta(y'_p(t+\delta) - y'(t))/2$$

Zobecnění vzorců pro soustavu diferenciálních rovnic je snadné. Pak y, d_1, d_2, λ, C_1 atd. značí vektory.

Pro řízení velikosti kroků h a δ doporučují autoři tuto strategii:
Změna integračního kroku h je závislá na velikosti — v absolutní hodnotě — maximální a minimální složky vektoru E.

Buďte α_1, α_2 konstanty, $\alpha_1 < \alpha_2$. Nechť $E_{max} = \max_K(|E_K|)$, $E_{min} = \min_K(|E_K|)$, $K = 1, 2, ..., m$, kde m značí počet rovnic soustavy (3.37).

Jestliže $E_{max} > \frac{3}{2}\alpha_2$, pak se integrační krok h půlí a řešení se nepřijme. V ostatních případech se řešení přijme, přičemž příští délka kroku je určována tímto způsobem:

- Je-li $\frac{3}{2}\alpha_2 > E_{max} > \frac{3}{4}\alpha_2$, pak se krok h půlí.
- Je-li $E_{max} \leq \frac{\alpha_2}{4}$ a $E_{min} \geq \alpha_1$, pak se délka integračního kroku h nemění.
- Je-li $E_{max} \leq \frac{3}{4}\alpha_2$ a $E_{min} < \alpha_1$, pak přichází zdvojovací signál. Ke skutečnému zdvojení integračního kroku dojde teprve tehdy, přišel-li zdvojovací signál v sedmi po sobě jdoucích krocích. Aby nedocházelo k opakovanému zdvojování a půlení kroku, stačí zachovat poměr $\alpha_2 : \alpha_1 = 1 : 150$.

Změna kroku δ je závislá na velikosti maximální složky vektoru Ep. Označme $Ep_{max} = \max(|Ep_K|)$, $K = 1, 2, ..., m$. Změna δ podléhá podobné strategii jako změna kroku h. Meze β_1, β_2 jsou poloviční ve srovnání s mezemi pro řízení kroku h : $\beta_1 = \alpha_1/2$, $\beta_2 = \alpha_2/2$.

- Krok δ se půlí, je-li $Ep_{max} > \beta_2$.
- Krok δ se zdvojuje, platí-li $Ep_{max} < \beta_1$ v sedmi po sobě jdoucích integračních krocích.
- Krok δ nepřevýší hodnotu $h/4$.

Toto kritérium pro strategii volby kroku není samozřejmě jediné, je však jednoduché a efektivní.

Fowler-Wartenovy metody se týkají tyto závěry:

1. Metoda je vhodná pro řešení lineárních i nelineárních soustav diferenciálních rovnic.
2. U lineárních soustav, kde maximální vlastní číslo $lambda_{max}$ (v absolutní hodnotě) je reálné a tuhostní poměr velký, vede metoda k podstatnému snížení doby výpočtu.
3. Je-li λ_{max} komplexní, je použití Fowler-Wartenovy metody zhruba rovnocenné s použitím metody Runge-Kutta druhého řádu.
4. U nelineárních soustav se dá výsledný efekt stěží odhadnout. U konkrétní testované soustavy byla metoda Fowler-Wartenova 40-50 krát rychlejší, než metoda Runge-Kutta čtvrtého řádu.
5. Analýza stability Fawler-Wartenovy metody, ve smyslu jak ji známe pro vícekrokové metody, zůstává stále otevřeným problémem.

Metoda Fowler-Wartenova byla úspěšně implementována v simulačním jazyce CSMP.

Na závěr tohoto odstavce shrňme poznatky o těch vlastnostech metod numerického řešení soustav diferenciálních rovnic, které tvoří obvykle kritéria pro jejich zařazení do simulačního jazyka a pro výběr metody pro danou aplikaci.

Přesnost

Je zřejmé, že metody vyšších řádů jsou přesnější (mají menší lokální chybu),než metody řádů nižších. U vícekrokových metod jsou metody implicitní přesnější, než metody explicitní.

Stabilita

(viz *obr.* **??**)

Jednokrokové metody Runge-Kutta: s rostoucím řádem metody se zvětšuje oblast absolutní stability.

Metody prediktor-korektor a Adamsovy explicitní metody: s rostoucím řádem se zmenšuje oblast absolutní stability. Metody prediktor-korektor mají větší oblast absolutní stability, než Adamsovy explicitní metody téhož řádu.

Časová náročnost

Je zřejmá platnost těchto tvrzení: Rychlost řešení je pro daný integrační krok h nepřímo úměrná řádu metody. Vícekrokové metody při daném integračním kroku jsou rychlejší než metody jednokrokové.

Vliv chyby: Metody vyšších řádů mají menší lokální chybu, tudíž připouštějí delší integrační krok. Omezíme-li krok integrace shora intervalem tisku, pak při řešení konkrétního problému může metoda vyššího řádu nevyužitím maximální délky integračního kroku zbytečně zpomalit řešení úlohy.

U jednokrokových metod způsobuje odhad chyby podle Richardsona značné časové ztráty oproti odhadu lokální chyby metod prediktor-korektor.

U vícekrokových metod dochází zase k časovým ztrátám při každé změně integračního kroku, neboť jsou nesamostartující.

Vliv stability: Jednokrokové metody nižšího řádu požadují k docílení stability menšího integračního kroku — odtud plyne, že např. Eulerova metoda může být pro řešení dané soustavy pomalejší, než metoda Runge-Kutta určitého řádu.

U vícekrokových metod je situace opačná — metoda nižšího řádu povoluje z hlediska stability delší integrační krok (viz. *obr. 3.1*).

Nároky na paměť

Nároky, které klade metoda na paměť, jsou přímo úměrné řádu metody. Pro metody daného řádu platí:

Metody vícekrokové mají větší nároky na paměť, než metody jednokrokové. U vícekrokových metod mají implicitní metody větší nárok na paměť, než metody explicitní. Dále je zřejmé, že metody s proměnným krokem integrace mají větší nároky na paměť, než metody s pevným krokem integrace.

3.5 Principy výstavby programových prostředků pro modelování spojitých systémů

Jak jsme již uvedli v podkapitole *3.3*, spektrum programových systémů, jež umožňují implementaci spojitých modelů na číslicovém počítači, je velmi široké. Zahrnuje na jedné straně programy, které na základě vstupních dat umožňují řešit konkrétní třídu spojitých modelů,

na druhé straně pak jazyky, které, kromě toho, že umožňují přepis jak matematické, tak funkcionální reprezentace spojitého systému, obsahují prostředky obecného programovacího jazyka.

V tomto odstavci se zaměříme na problémy, spojené s organizací výpočtu spojitého simulačního modelu, zadaného programovým systémem pro spojitou simulaci. Tento problém vyplývá ze základní vlastnosti spojitých simulačních jazyků — z jejich neprocedurálnosti.

3.5.1 Pojem funkční blok, klasifikace funkčních bloků

Výrazové prostředky simulačního jazyka pro zápis abstraktního spojitého modelu mají deskriptivní (neprocedurální) charakter, tj. umožňují bez ohledu na způsob řešení modelu zapsat v jistém tvaru soustavu diferenciálních a algebraických rovnic, nebo propojení funkčních bloků spojitého modelu zadaného ve tvaru blokového schematu. Úkolem překladače simulačního jazyka je pak vyčlenění určitých funkčních prvků (korespondujících obvykle s příkazy jazyka) a jejich uspořádání do posloupnosti, v níž se budou tyto funkční prvky interpretovat v rámci numerického řešení modelu.

Zmíněné funkční prvky, které se tradičně nazývají *bloky*, je účelné klasifikovat do dvou tříd. Uvažujeme nejdříve chéma implementace metody pro řešení soustavy diferenciálních rovnic $y' = f(x, y)$, $y(x_0) = y_0$ pro jednoduchost např. Eulerovu metodu (*obr.* **??**).

Mezi příkazy, označenými v tomto schématu čísly (2) a (3) lze vidět kvalitativní rozdíl. Zatímco příkaz (2) realizuje vyčíslení funkce, příkaz (3) odpovídá numerické formuli metody a podle *kap. 2* jej lze chápat jako numerický integrátor s vnitřní stavovou proměnnou (pamětí). Klasifikaci funkčních prvků — bloků — výpočetního schématu spojitého modelu provedeme právě podle tohoto rozdílu. Bloky, jež mají vnitřní stavovou proměnnou (proměnné), nazýváme *bloky s pamětí* (paměťové bloky nebo pomalé bloky) a bloky, jež nemají vnitřní stavovou proměnnou, nazýváme *bloky bez paměti* (bezpaměťové bloky nebo rychlé bloky).

Nyní můžeme zobecnit předchozí výpočetní schéma tak, jak je na *obr.* **??**.

Povšimneme si počtu bloků příslušejících implementaci uvažované soustavy u diferenciálních rovnic. Zřejmě každé diferenciální rovnici 1. řádu přísluší jeden integrátor, takže celkový počet paměťových bloků je roven n. Počet bezpaměťových bloků pro vyčíslení funkce $f(x, y)$ nelze apriorně stanovit, závisí obecně na tvaru funkce $f(x, y)$ a na tom, kolika aritmetickými výrazy je výpočet této funkce popsán (viz *obr. 2.10* a *2.11*).

Pro korektní numerické řešení spojitého systému je velmi důležité pořadí, ve kterém se jednotlivé bloky provádějí.

Zabývejme se nejdříve problémem, zda pořadí zpracování paměťových bloků může ovlivnit korektnost výsledného řešení. Uvažujeme část modelu na *obr.* **??**.

Dále předpokládejme, že integrátor je realizován funkční procedurou INT(X,Y) se dvěma parametry: X určuje počáteční podmínku a Y vstup integrátoru. Opět předpokládejme, že numerická integrace je prováděna Eulerovou metodou. Pak zápisy

```
 Y1 = INT(A,OMEGA)
 Y0 = INT(B,Y1)

resp.
 Y0 = INT(B,Y1)
 Y1 = INT(A,OMEGA)
```

jež oba popisují propojení z *obr.* **??**., nejsou ekvivalentní, poněvadž první zápis vede k chybnému řešení. Skutečně, po provedení prvního příkazu $Y1$ =INT($A, OMEGA$), má-li

nezávisle proměnná hodnotu $x = x_n$, bude v Y1 hodnota řešení v čase $x_n + h$. To však má za následek, že při výpočtu druhého příkazu $Y0 =$INT$(B, Y1)$ je při vyčíslení Eulerovy formule (při hodnotě nezávisle proměnné $x = x_n$) počítán výraz $h * f(x_n, y_{n+1})$ namísto $h * f(x_n, y_n)$.

Abychom zabránili takovému vedlejšímu efektu procedur, realizujících numerickou integraci a dosáhli nezávislosti na pořadí vyhodnocování integrátorů (případně i jiných paměťových bloků), volí se následující strategie pro začlenění numerických metod řešení diferenciálních rovnic do spojitých simulačních systémů.

Na *obr.* **??** je schéma pro tzv. centralizovanou integraci aplikované na diskutovaný příklad z *obr.* **??**.

Proměnné Y_s1 a Y_s0 představují vnitřní proměnné, které nabývají hodnoty řešení v bodě x_n+h, kdežto proměnné $Y1$ resp. $Y2$ uchovávají hodnotu řešení v bodě x_n. Za tohoto režimu může být pořadí vyhodnocení integrátorů libovolné. Po ukončení vyhodnocení posledního integrátoru je nyní třeba převést hodnoty proměnných Y_s1, resp. Y_s2 do proměnných $Y1$, resp. $Y2$ a může být realizován další ineagrační krok. Toto schéma integrace lze samozřejmě zobecnit i pro metody Runge-Kutta a prediktor-korektor.

Při vyčíslování bezpaměťových bloků nelze aplikací nějaké analogické techniky učinit pořadí jejich zpracování nevýznamným. Vyčíslení bezpaměťových bloků, které odpovídají funkci na pravé straně diferenciální rovnice, musí probíhat v pořadí, jež vyplývá ze strukturalizace této funkce. V terminologii převzaté z analogových výpočetních schémat v pořadí ve směru toku signálu. Další závažnou komplikací při hledání tohoto pořadí je existence zpětných vazeb v propojení bezpaměťových prvků — tzv. rychlých smyček. Uvažujme např. propojení (*obr.* **??**), které odpovídá soustavě rovnic

$$y' = z, \quad y(0) = A$$

$$z^3 + z = y$$

Na rozdíl od zpětnovazebních propojení, obsahujících paměťové bloky, nemůže být rychlá smyčka řešena postupným vyčíslováním jejích bloků, protože nelze nalézt blok, jenž má "správně definovaný" výstup. Z korespondence rychlé smyčky s algebraickou rovnicí plyne její alternativní název algebraická smyčka. Existují-li ve spojitém modelu algebraické smyčky, musí je překladač nejprve identifikovat a posléze převést jejich vyhodnocování na integrační výpočty v rámci nějaké metody řešení algebraických rovnic. Problému nalezení těchto smyček se budeme podrobněji věnovat v *odst. 3.5.2.*

3.5.2 Třídění a uspořádání funkčních bloků

Chceme-li, aby simulační jazyk umožňoval uživateli popsat spojitý systém bez ohledu na pořadí vyčíslování funkčních bloků, případně existenci rychlých smyček, je nutné, aby překladač realizoval také třídění a uspořádání funkčních bloků.

Strukturu propojení funkčních bloků můžeme znázornit orientovaným grafem $G_D = \langle U_D, H_D, f_D \rangle$. Nechť i-tý funkční blok struktury je reprezentován i-tým uzlem $u_i \in U_D$, spojení výstupu i-tého bloku se vstupem j-tého bloku nechť je reprezentováno orientovanou hranou $h_k \in H_D$, $f_D(h_k) = (u_i, u_j)$.

Z hlediska třídění funkčních bloků je důležité poze rozlišení paměťových a bezpaměťových bloků. Protože při vyhodnocování paměťových bloků nezáleží na jejich vzájemném pořadí, není třeba provádět jejich uspořádání. Z toho důvodu vyjmeme z grafu G_D všechny uzly, reprezentující paměťové bloky a všechny hrany, směřující do nich a z nich. Takto vzniklý nový graf označme $G_Q = \langle U_Q, H_Q, f_Q \rangle$, kde

$$U_Q \in U_D$$

$$H_Q \in H_D$$

$$f_Q \in f_D/H_Q$$

Obr. 3.16 znázorňuje příklad grafu G_D a *obr.* **??** znázorňuje odpovídající graf G_Q za předpokladu, že uzly u_2, u_3 *obr.* **??** reprezentovaly paměťové bloky. Nyní ukážeme, jak lze postupovat při hledání uspořádání bezpaměťových bloků.

Nejprve v G_Q nalezneme všechny silné komponenty SK_k. Aby nedošlo k tomu, že dva různé uzly budou označeny stejným indexem, nechť index k silné komponenty SK_k je roven indexu některého uzlu patřícímu k SK_k v grafu G_Q. Připomeňme, že silnou komponentou grafu rozumíme největší silně souvislý podgraf grafu.

Značí-li nyní M množinu uzlů všech silných komponent SK_k, pak sestrojíme graf $G_A = \langle U_A, H_A, f_A \rangle$ takto:

Z grafu G_Q vezměme množinu R uzlů u_i, které nepatří k žádné silné komponentě SK_k, tj. $u_i \in R = U_Q - M$ a množinu všech hran (u_i, u_j), jejichž oba krajní uzly $u_i, u_j \in R$. Každou silnou komponentu SK_k redukujeme na jediný uzel $u_k \in R'$. Do uzlu $u_k \in R'$ nechť směřují všechny hrany z G_Q, které mají za výstupní uzel některý uzel $u_i \in R$ a jejichž vstupním uzlem je některý uzel silné komponenty SK_k. Analogicky z uzlu u_k nechť vycházejí všechny hrany, jejichž počátečním uzlem je některý uzel SK_k a koncovým uzlem některý uzel $u_i \in R$. Pak

$$U_A = R \cup R', \quad R \cap R' = \emptyset$$

kde $R' = \{u_k\}, \quad u_k \in SK_k$

$$H_A = H_Q - \cup H_k$$

kde H_k je množina hran silné komponenty SK_k:

$$h_i \in H_k \Rightarrow f_Q(h_i) = (u_i, u_j) \wedge u_i, u_j \in SK_k.$$

Orientovaná icidence grafu G_A je definována takto:
Pro každou hranu $h_q \in H_A$ je $f_A(h_q) = (u_i, u_j)$ právě tehdy, platí-li jedna z těchto podmínek:

a) $f_Q(h_q) = (u_i, u_j)$ a $u_i, u_j \in R$

b) $f_Q(h_q) = (u_i, u_l)$ a $u_i \in R$ a $u_l, u_j \in SK_m$

c) $f_Q(h_q) = (u_k, u_j)$ a $u_j \in R$ a $u_k, u_i \in SK_n$

d) $f_Q(h_q) = (u_k, u_l)$ a $u_k, u_i \in SK_m$ a $u_l, u_j \in SK_n$ a $m \neq n$

Takto vytvořený graf $G_A = \langle U_A, H_A, f_A \rangle$ je acyklický, neboť neobsahuje žádné cykly, může však být obecně nesouvislý. Graf G_A, odvozený výše uvedeným způsobem z grafu G_Q na *obr.* **??**, je znázorněn na *obr.* **??**.

Každý uzel grafu G_A představuje určitý funkční blok a tudíž mu přísluší akce pro vyhodnocení tohoto funkčního bloku. Akce, jež přísluší uzlu $u_i \in R$ představuje výpočet i-tého funkčního bloku, akce, příslušející uzlu $u_j \in R'$, odpovídá výpočtu všech funkčních bloků v grafu G_Q, tvořících silnou komponentu SK_k, $(u_j \in SK_k)$.

Na základě acyklického grafu G_A můžeme definovat relaci ρ v U_A takto:

$\rho = \{\langle u_i, u_j \rangle |$ existuje orient. cesta u z uzlu u_i do $u_j\}$

Relace ρ je zřejmě částečným uspořádáním (reflexivní, tranzitivní a antisymetrická) a definuje nutné precedence v pořadí vyčíslování jednotlivých bloků. Je-li $u_i \rho u_j$, pak zřejmě blok u_i musí být vyhodnocen dříve než blok u_j; jestliže $\langle u_i, u_j \rangle \notin \rho$, pak na vzájemném pořadí vyčíslení bloků u_i a u_j nezáleží.

Známe-li relaci ρ, pak lze problém seřazení bloků formulovat i realizovat velmi snadno: částečné uspořádání ρ rozšíříme na úplné (lineární) uspořádání na množině U_A.

Jinými slovy, problém seřazení bezpaměťových bloků spočívá v nalezení posloupnosti

$$u_{i_1}, u_{i_2}, ..., u_{i_{n-1}}, u_{i_n}$$

pro jejíž každé dva prvky u_{i_j} a u_{i_k}, kde $j < k$, platí buď $u_{i_j} \rho u_{i_k}$, nebo $\langle u_{i_j}, u_{i_k} \rangle \notin \rho$.
Pro graf z *obr.* **??**. jsou takovými posloupnostmi např. posloupnosti

$$u_1, u_9, u_8, u_4, u_{10}, u_5, u_{11} \quad \text{nebo}$$

$$u_4, u_5, u_8, u_9, u_{10}, u_1, u_{11}$$

avšak nikoli posloupnost

$$u_1, u_{10}, u_{11}, u_9, u_8, u_5, u_4$$

K nalezení posloupnosti uspořádaných bezpaměťových bloků je možno použít následujícího jednoduchého algoritmu, jehož vstupem je graf G_A:

(1) polož $i = 1$

(2) z množiny U_A vybereme uzel, do kterého nevstupují žádné hrany; vzhledem k existenci uspořádání ρ v U_A takový prvek existuje a tvoří i-tý prvek hledané posloupnosti

(3) nyní vyjmeme nalezený "nejmenší" uzel z U_A spolu s hranami, které z něho vycházejí

(4) je-li $i = n$ (n je celkový počet uzlů grafu G_A), pak algoritmus končí nalezením hledaného uspořádání uzlů; v opačném případě polož $i = i + 1$ a vrať se k (2).

Na závěr tohoto odstavce se zabývejme způsobem řešení bloků, které příslušejí silné komponentě grafu G_Q. V grafu G_A této silné komponentě odpovídá jediný uzel. Jemu příslušející akcí je řešení algebraické rovnice (nebo soustavy rovnic), která koresponduje se zpětnovazebním propojením bezpaměťových bloků.

Jako příklad uvažujme silnou komponentu SK_5 z *obr.* **??**. Označme písmenem V_k proměnnou nabývající hodnoty výstupu funkčního bloku reprezentovaného uzlem u_k a písmenem f_k funkci, kterou realizuje funkční blok u_k. Pak silné komponentě SK_5 přímo odpovídá soustava

$$\begin{aligned} V_{13} &= f_{13}(V_7) \\ V_7 &= f_7(V_5) \\ V_5 &= f_5(V_4, V_6, V_{13}) \\ V_6 &= f_6(V_6, V_7) \end{aligned}$$

o čtyřech neznámých V_{13}, V_7, V_5, V_6.

Substitucemi za proměnné V_k můžeme zredukovat počet rovnic soustavy k tvaru

$$\begin{aligned} V_7 &= f_7(f_5(V_4, V_6, f_{13}(V_7))) \\ V_6 &= f_6(V_6, V_7) \end{aligned}$$

což je z hlediska numerického řešení jistě tvar příznivější.

Pro algoritmizaci redukce počtu rovnic soustavy algebraických rovnic, prováděnou překladačem, se používá *metody řídicích bloků* (proměnných).

Označme $\tilde{U}_k$ množinu uzlů tvořících silnou komponentu SK_k. Množina $U_k^R \subset \tilde{U}_k$, jež odpovídá množině řídicích bloků, obsahuje uzly s touto vlastností: Po vyjmutí všech uzlů $u_i \in U_k^R$ z $\tilde{U}_k$ neobsahuje silná komponenta ani cyklus ani smyčku. Vrácením libovolného uzlu $u \in U_k^R$ do $\tilde{U}_k$, vznikne v SK_i alespoň jeden cyklus nebo smyčka.

Máme-li nalezenou množinu řídicích bloků, pak lze redukovat počet rovnic soustavy takto:

Nechť k je počet řídicích bloků a m počet cyklů a smyček silné komponenty SK_i. Pak platí $k \leq m$ a komponentu SK_i můžeme řešit jako soustavu

$$V_j = F_j(V_1, V_2, ..., V_k), \quad j = 1, 2, ..., k$$

kde V_j je hodnota výstupu j-tého řídicího bloku a funkce F_j závisí na funkčních předpisech bloků silné komponenty. V příkladě na *obr.* **??**. je $m = 3$ a $k = 2$. Pro množinu řídících bloků $\{u_6, u_7\}$ jsme výslednou soustavu dvou algebraických rovnic již uvedli. Poznamenejme, že jinou možnou množinou řídicích bloků, jež vede opět k soustavě dvou algebraických rovnic, je množina $\{u_5, u_6\}$.

V rámci řešení rovnic, odpovídajících silné komponentě, není třeba hledat explicitní vyjádření funkcí F_j. Snazší cestou je uspořádání těch bloků, jež nejsou řídicí a jež se podílejí na výpočtu funkčních hodnot funkcí F_j, do správných posloupností. Vzhledem k tomu, že po vyjmutí řídicích bloků je graf silné komponenty acyklický, je tento problém snadný a je obdobou již diskutovaného problému uspořádávání bezpaměťových bloků.

Jako příklad uvažujme opět komponentu z *obr.* **??**. Pro množinu řídicích bloků u_6, u_7 znázorníme výsledný acyklický graf dané komponenty zdvojením uzlů, které odpovídají řídicím blokům (tj. "roztržením" příslušných cyklů nebo smyček) na *obr.* **??**.

Takto získáme acyklický graf, v němž uzly X_7, resp. X_6 odpovídají výstupům řídícího bloku u_7, resp. u_6. Setříděním tohoto acyklického grafu získáme vztahy

$$\begin{aligned} V_7 &= f_7(f_5(V_4, X_6, f_{13}(X_7))) \\ V_6 &= f_6(X_6, X_7) \end{aligned}$$

jež tvoří jádro iteračního řešení soustavy

$$\begin{aligned} V_7 &= f_7(f_5(V_4, V_6, f_{13}(V_7))) \\ V_6 &= f_6(V_6, V_7) \end{aligned}$$

3.5.3 Překladače simulačního jazyka

Pod pojmem překladač simulačního jazyka rozumíme, i když ne zcela správně, celý programový systém, jenž umožňuje zadat spojitý model ve tvaru spojitého simulačního programu, jeho transformaci do procedurálního tvaru v určitém intermediárním jazyce a jeho provádění v případném interaktivním režimu s konverzačními rysy.

Systém s takto vymezenou funkcí lze ve skutečnosti rozdělit do dvou oddělitelných částí:

- vlastní překladač jazyka
- soubor výkonných podprogramů a komunikačních podprogramů

První překladače simulačních jazyků generovaly pro vstupní popis spojitého modelu přímo program ve strojovém jazyce. Jak jsme již uvedli, postupně simulační jazyky přebíraly většinu rysů obecných programovacích jazyků, což přirozeně vedlo k tomu, že překladač simulačního jazyka musel realizovat takřka tytéž funkce, jako překladač obecného jazyka a ještě další, vyplývající za specifik simulačních jazyků.

Vzhledem k minimalizaci množství práce, spojené s výstavbou překladače spojitého simulačního jazyka, se obvykle volí strategie překladu, zobrazená na *obr.* **??**.

Za intermediární jazyk překladače spojitého simulačního jazyka tedy můžeme považovat obecný programovací jazyk. U většiny starších jazyků to byl FORTRAN.

Zaměřme se nyní na to, které funkce realizuje překladač simulačního jazyka. K základním patří zřejmě transformace neprocedurálního simulačního programu na procedurální program, obsahující volání podprogramů nebo procedur, jež odpovídají realizaci funkčních bloků. Tato volání musí být uspořádána do sekvenčních posloupností, které se opakují obvykle se změnou hodnoty modelového času (odpovídají jednomu kroku numerického řešení modelu). To tedy znamená, že součástí překladače je implementace algoritmu seřazování bloků — tzv. *seřazovací program.*

Poznámka:

Některé starší spojité simulační jayzyky neumožňují seřazování a tedy zakazují výskyt algebraických smyček a předpokládají "seřazení" bezpaměťových bloků programátorem.

Některé simulační jazyky umožňují, aby uživatel na základě standardních příkazů (bloků) mohl definovat nové, komplexnější a specializovanější funkční bloky sloužící ke stručnějšímu a pohodlnějšímu popisu daného spojitého systému. Proto, aby byl v rámci takových prostředků snadno řešitelný problém seřazování, bývají nejčastěji stylizovány jako makropříkazy. Pak překladač simulačního jazyka musí zajistit rozvoj volání makropříkazů, to znamená, že je jeho součástí makrogenerátor.

K další funkci překladače pak patří inicializace systémových proměnných podle tzv. "datových" příkazů, jež předepisují počáteční modelový čas, počáteční podmínky, intervaly tisku apod.

Druhá část programového systému pro spojitou simulaci obsahuje jednak soubor výkonných podprogramů, realizujících standardní funkční bloky simulačního programu a jednak komunikační programy, zprostředkující zásahy uživatele do řešení modelu. V této části jsou tedy implementovány také veškeré numerické metody, kterými simulační jazyk disponuje.

V kapitole 7 ukážeme objektový přístup k výstavbě spojitých simulačních jazyků. V rámci knihovny SIMLIB popíšeme simulační systém implementovaný v C++.

Kapitola 4

Systémy hromadné obsluhy a jejich analytické modely

4.1 Systém hromadné obsluhy

Systém hromadné obsluhy charakterizujeme jako systém, v němž k určitému zařízení (lince obsluhy), které poskytuje obsluhu jistého druhu, přicházejí prvky (požadavky) vyžadující obsluhu. Nejsou-li obslouženy ihned, mohou na obsluhu čekat ve frontě. Po skončení obsluhy opouštějí požadavky systém a obsluhu je možno poskytnout dalším požadavkům, které čekají ve frontě, nebo které teprve přijdou. Příklad jednoduchého systému hromadné obsluhy je na *obr.* **??**.

Teorie hromadné obsluhy zkoumá především ty situace, při nichž vznikají v systému zdržení a ztráty. Je to problematika hromadění osob v obchodech, úřadech, vozidel na křižovatkách, v opravnách, u benzínových čerpadel, obrobků v dílnách, hovorů v telefonních ústřednách, zakázek v počítačových systémech apod. Měla by uživateli umožnit nalezení závislosti mezi veličinami charakterizujícími kvalitu obsluhy a veličinami reprezentujícími vstupní tok požadavků, způsob obsluhy a organizaci fronty.

Vstupní požadavky přicházejí do sytému obsluhy ze zdroje požadavků, jenž může být omezený nebo neomezený. U omezeného zdroje kolísá intenzita vstupního toku požadavků v závislosti na počtu požadavků, které jsou právě ve frontě nebo v lince obsluhy; počet požadavků ve zdroji je tedy o tento počet menší. Měnící se počet požadavků ve zdroji má vliv na charakteristiku vstupního toku, zatímco u neomezeného zdroje se tento vliv neprojevuje. Neomezený zdroj požadavků je abstrakcí, které používáme, je-li počet požadavků ve zdroji dostatečně velký a jeho změna v důsledku úbytku o požadavky ve frontě a v obsluze je tak malá, že nemá podstatný vliv na produkovaný tok vstupních požadavků. Ve zvláštních případech může současně vzniknout několik požadavků na obsluhu; v tomto případě se vstupní tok skládá z dávek požadavků. Popis vstupního toku pak musí být rozšířen o popis počtu požadavků v dávce; tento počet může být také náhodný. Charakteristikou vstupního toku je rozdělení pravděpodobností dob mezi příchody požadavků do systému obsluhy. Jedním extrémem je konstantní doba mezi příchody požadavků a druhým extrémem, tzv. nejhorším případem, je exponenciální rozdělení dob mezi příchody požadavků. Systémy hromadné obsluhy, jejichž vstupní toky jsou charakterizovány rovnoměrným nebo exponenciálním rozložením pravděpodobností, lze řešit *analytickými metodami*. V reálných systémech se však vyskytují většinou vstupní toky, jejichž rozložení pravděpodobností leží někde mezi těmito extrémními případy. Pro řešení těchto systémů je zpravidla jedinou použitelnou metodou

simulace. Abychom ověřili správnost výsledků dosažených simulací, můžeme využít skutečnosti, že získané výsledky musí ležet v intervalu daném extrémními případy, které jsme předem vyřešili analyticky.

Fronta čekajících požadavků se může vytvořit vždy, když se doby mezi příchody požadavků nebo doby jejich obsluhy náhodně mění. Pro frontu jsou charakteristické dvě hodnoty; největší možná délka fronty s určitým způsobem výběru z fronty a řazení do fronty — *frontový řád*. Největší možná délka fronty může být i nulová; v tomto případě musí obslužný systém opustit každý požadavek, jenž najde obsluhu obsazenou. Dále může být fronta konečná (omezená velikostí čekacího prostoru) a nekonečná, může-li čekat na obsluhu libovolný počet požadavků. Nejjednodušším frontovým řádem je *řádný frontový řád*, při němž se požadavky řadí do fronty v pořadí, jak přicházejí. K obsluze odchází ten požadavek, který přišel nejdříve. Tento řád se zpravidla označuje FIFO (First In, First Out). Obráceným případem je *inverzní frontový řád*, kdy k obsluze odchází z fronty ten požadavek, jenž přišel do fronty poslední. Je to režim označovaný LIFO (Last In, First Out). Další možností je *náhodný výběr z fronty*. Komplikovanější jsou případy, kdy požadavek po nějakou dobu čeká, ale pak z fronty odchází nebo frontu předběhne. Jiným důležitým případem jsou *fronty s prioritami*, kdy nejsou přicházející požadavky rovnocenné. Některé požadavky mají přednost a předbíhají ve frontě ty požadavky, které přišly před nimi. Prioritní úrovně mohou být dvě, tři nebo jich může být i více. Tak lze u jednoho obslužného zařízení vytvářet i několik front s různými prioritami. Přitom se požadavky ve frontě s nižší prioritou mohou brát k obsluze jen tehdy, když jsou fronty s vyšší prioritou prázdné. Kromě toho může být priorita požadavků také funkcí doby, po kterou požadavek čeká. Po určité době, strávené ve frontě, se tedy může požadavek přeřadit do fronty s vyšší prioritou. Při uplatňování priority může být právě probíhající obsluha požadavků s nižší prioritou dokončena nebo přerušena. Byla-li obsluha přerušena, pokračuje se v ní (až na požadavek přijde řada) tam, kde se skončilo, nebo se začne s obsluhou znova. Přicházejícím požadavkům bývá priorita přidělena před vstupem do obslužného systému. Možnosti různých frontových řádů jsou velmi rozmanité. Komplikovanější případy je opět zpravidla nutné řešit simulací.

Obsluha může být poskytována jednou nebo několika paralelními linkami. Je-li současně více volných obslužných linek, pak přicházející požadavek přejde buď náhodně na některou z nich, nebo se linky obsazují podle nějakého pořadí. přitom vždy předpokládáme, že po ukončení nějaké obsluhy, tj. při uvolnění některé linky, se okamžitě začne obsluhovat další požadavek, pokud je nějaký ve frontě. Jinak musí obsluha čekat na příchod nového požadavku. Vlastní obsluha může trvat pevně stanovenou dobu, nebo je tato doba náhodná, případně je závislá na typu požadavku nebo na použité lince obsluhy. Zpravidla nás zajímá střední doba, po kterou je požadavek obsluhován, případně zákon rozložení pravděpodobností délky doby obsluhy, podobně jako je tomu u doby mezi příchody požadavků.

4.2 Náhodné procesy

4.2.1 Úvod

Matematickou abstrakcí skutečných procesů, které probíhají v systémech hromadné obsluhy, jsou tzv. *náhodné* (*stochastické*, *pravděpodobnostní*) procesy. Dříve než podáme definici náhodného procesu, připomeneme některé základní pojmy z teorie pravděpodobnosti.

1. Uspořádanou dvojici $(\Omega, \mathcal{R})$ nazveme *měřitelným prostorem* a množinu $\mathcal{R}$ *σ-algebrou*, jestliže platí:

   (i) $\Omega \in \mathcal{R}$

(ii) $\forall E \quad (E \in \mathcal{R} \quad \Rightarrow \quad \Omega - E \in \mathcal{R})$

(iii) Je-li $\forall i \in \mathbf{N}$ $(E_i \in \mathcal{R})$ a $E = \bigcup_{i=1}^{\infty} E_i$, pak $E \in \mathcal{R}$.

Z (ii) a (iii) navíc plyne:

(iv) Je-li $\forall i \in \mathbf{N}$ $(E_i \in \mathcal{R})$ a $E = \bigcap_{i=1}^{\infty} E_i$, pak $E \in \mathcal{R}$.

V dalším textu budeme pod symbolem $\mathcal{R}_{\mathbf{R}}$ (resp. $\mathcal{R}_{\mathbf{R}^n}$) rozumět tzv. σ-algebru borelovských množin na $\mathbf{R}$ (na $\mathbf{R}^n$), která je definována jako nejmenší (vzhledem k inkluzi) σ-algebra obsahující všechny otevřené intervaly v $\mathbf{R}$ (v $\mathbf{R}^n$).

2. Uspořádanou trojici $(\Omega, \mathcal{R}, p)$ nazveme *pravděpodobnostním prostorem*, je-li $(\Omega, \mathcal{R})$ měřitelný prostor a $p : \Omega \to \langle 0, 1\rangle$ takové zobrazení, že platí:

(v) $p(\Omega) = 1$

(vi) Je-li $\{E_i | i \in \mathbf{N}\}$ disjunktní spočetný systém prvků $\mathcal{R}$, platí rovnost

$$p(\bigcup_{i=1}^{\infty} E_i) = \sum_{i=1}^{\infty} p(E_i).$$

Zobrazení p se nazývá pravděpodobnostní míra na $(\Omega, \mathcal{R})$, příp. *pravděpodobnostní rozložení* na $(\Omega, \mathcal{R})$. Prvky množiny Ω bývá zvykem nazývat *elementárními jevy.*

3. Pod pojmem *náhodná veličina* budeme rozumět zobrazení $X : \Omega \to \mathbf{R}$, definované na jistém pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$, platí-li

(vii) $\forall E \quad (E \in \mathcal{R}_{\mathbf{R}} \Rightarrow X^{-1}[E] \in \mathcal{R})$.

Podobně řekneme, že X je n-rozměrným náhodným vektorem, jestliže je $X : \Omega \to \mathbf{R}^n$ a platí

(viii) $\forall E \quad (E \in \mathcal{R}_{\mathbf{R}^n} \Rightarrow X^{-1}[E] \in \mathcal{R})$.

Připomeňme, že $X^{-1}[E] = \{\omega \in \Omega | X(\omega) \in E\}$.

Protože pro každé $\omega \in \Omega$ je hodnota $X(\omega)$ n-rozměrného náhodného vektoru X prvkem n-rozměrného lineárního prostoru, můžeme tuto hodnotu vyjádřit ve tvaru $(X_1(\omega), ..., X_n(\omega))$, případně ve tvaru $(X(1)(\omega), ..., X(n)(\omega))$, kde $X(i) : \Omega \to \mathbf{R}$ jsou náhodné veličiny pro $i = 1, 2, ..., n$. Jinak řečeno, každý n-rozměrný náhodný vektor můžeme vyjádřit jako konečnou posloupnost $(X(i))_{i \in \{1,2,...,n\}}$ jistých náhodných veličin. Z tohoto hlediska je pojem náhodného procesu zobecněním pojmu náhodného vektoru.

Náhodný proces můžeme definovat jako soubor $(X(t))_{t \in T}$ náhodných veličin $X(t) : \Omega \to \mathbf{R}$ definovaných na určitém pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$. Je-li T spočetná množina, říkáme, že $(X(t))_{t \in T}$ je náhodný proces *s diskrétním časem*, je-li T nespočetná (např. $T = \mathbf{R}$), říkáme, že $(X(t))_{t \in T}$ je náhodný proces *se spojitým časem*. Poznamenejme, že je-li $T = 1, 2, ..., n$, splývá prakticky náhodný proces $(X(t))_{t \in T}$ s n-rozměrným náhodným vektorem $X : \Omega \to \mathbf{R}^n$.

Množinu $\Im := \bigcup_{t\in T}\{X(t)(\omega)|\omega \in \Omega\}$ nazýváme *množinou stavů*. Je-li množina $\Im$ spočetná, mluvíme o *procesu s diskrétními stavy*.

Přestože pojem náhodného procesu může být definovaný obecněji pro naše další úvahy vystačíme s výše uvedenou definicí a s předpokladem $T \subset \mathbf{R}$.

4. Nechť X je náhodná veličina definována na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$. Pak pravděpodobnostní míra $p \bullet X^{-1} : \mathcal{R}_{\mathbf{R}} \to \langle 0, 1\rangle$, $E \to p(X^{-1}[E])$ se nazývá *pravděpodobnostní rozložení náhodné veličiny* $X : \Omega \to \mathbf{R}$ a funkci $F : \mathbf{R} \to \langle 0, 1\rangle$, $x \to p(X^{-1}[(-\infty, x)])$ nazýváme *distribuční funkcí náhodné veličiny* X. Pro každé $x \in \mathbf{R}$ zřejmě platí:

$$F(x) = p \bullet X^{-1}[(-\infty, x)] = p(X^{-1}[(-\infty, x)]) = p(\{\omega|X(\omega) \in (-\infty, x)\}) =$$
$$= p(\{\omega|X(\omega) < x\}).$$

Předchozí vztah bývá zvykem zapisovat stručněji ve tvaru $F(x) = p(X < x)$. Zkráceného zápisu $p(X < x)$ budeme nadále používat. Nesmíme však zapomenout, že p je pravděpodobnostní míra definována na jistém měřitelném prostoru $(\Omega, \mathcal{R})$, a že tedy pravděpodobnost $p(X < x)$ je rovna pravděpodobnosti množiny elementárních jevů $\{\omega|X(\omega) < x\} \subset \Omega$ a pravděpodobnost $p(X = x)$ je rovná pravděpodobnosti množiny elementárních jevů $\{\omega|X(\omega) = x\} \subset \Omega$ pravděpodobnostního prostoru $(\Omega, \mathcal{R}, p)$.

5. Podobných zkrácených zápisů užíváme též pro vyjádření podmíněných pravděpodobností. Jsou-li $X(t_0), X(t_1), ..., X(t_{n+1})$ náhodné veličiny definované na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$, $(X(t_i) : \Omega \to \mathbf{R}, \omega \to X(t_i)(\omega)$ pro $i = 0, 1, ..., n + 1$, pak např. podmíněnou pravděpodobnost

$$\mathcal{P}(\{\omega|X(t_{n+1})(\omega) < x_{n+1}\}|\{\omega|X(t_0)(\omega) = x_0\} \cap \{\omega|X(t_1)(\omega) = x_1\} \cap ...$$
$$\cap\{\omega|X(t_1)(\omega) = x_n\})$$

budeme zapisovat ve tvaru:

$$\mathcal{P}(X(t_{n+1}) < x_{n+1}|X(t_0) = x_0, X(t_1) = x_1, ..., X(t_n) = x_n).$$

Připomeňme, že pro $A \in \mathcal{R}$, $B \in \mathcal{R}$ je *podmíněná pravděpodobnost* $\mathcal{P}(A|B)(p(B) \neq 0)$ definovaná vztahem:

$$\mathcal{P}(A|B) = \frac{p(A \cap B)}{p(B)}, \tag{4.1}$$
$$p(A \cap B) = \mathcal{P}(A|B).p(B) \tag{4.2}$$

V případě, že např. $B = \{\omega|X(\omega) = x_0\}$ a $p(B) = 0$, kde X je náhodná veličina (resp. náhodný vektor) s distribuční funkcí F, definujeme nejprve, motivováni vztahem (4.2), funkci $\mathbf{R} \to \mathbf{R}(\mathbf{R} \to \mathbf{R}^n)$ zobrazující $x \to \mathcal{P}(A|X = x)$, která vyhovuje následujícímu vztahu:

$$\forall C \in \mathcal{R}_{\mathbf{R}} \text{ (resp. } \mathcal{R}_{\mathbf{R}^n}) : p(A \cap X^{-1}[C]) = \int_C \mathcal{P}(A|X = x)dF(x). \tag{4.3}$$

Existence této funkce vyplývá z Radonovy-Nikodymovy věty a její hodnotu $\mathcal{P}(A|X = x_0)$ nazýváme podmíněnou pravděpodobností náhodného jevu A za podmínky $X = x_0$ na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$, kde $x_0 \in \mathbf{R}$ (resp. $x_0 \in \mathbf{R}^n$).

Poznamenejme, že v případě $p(X^{-1} > [\{x_0\}]) = p(\{\omega \in \Omega | X(\omega) = x_0\}) \neq 0$ platí vztah:

$$\mathcal{P}(A|X = x_0) = \frac{p(A \cap X^{-1}[\{x_0\}])}{p(X^{-1}[\{x\}])},$$

který je klasickým vyjádřením podmíněné pravděpodobnosti.

Příklad 1:

Nechť (X_1, X_2) je náhodný spojitý vektor s hustotou h. Funkce $f(r_1) = \int_{-\infty}^{\infty} h(r_1, r_2)dr_2$ je hustota náhodné veličiny X_1. Vyjádřete $\mathcal{P}(X_2 < x_2 | X_1 = x_1)$.

Řešení: Předpokládejme, že pro všechna $r_1 \in C$, $C \in \mathcal{R}_{\mathbf{R}}$, je $f(r_1) > 0$. Pak

$$p(\{\omega | X_2(\omega) < x_2\} \cap X_1^{-1}[C]) = \int\limits_{(-\infty, x_2) \times C} h(r_1, r_2)dr_1 dr_2 =$$

$$\int_C \left(\int\limits_{(-\infty, x_2)} \frac{h(r_1, r_2)}{f(r_1)} dr_2 \right) f(r_1)dr_1 = \int_C \left(\int\limits_{(-\infty, x_2)} \frac{h(r_1, r_2)}{f(r_1)} dr_2 \right) dF(r_1),$$

kde F je distribuční funkce náhodné veličiny X_1. Odtud porovnáváním s výrazem (4.3) plyne:

$$\mathcal{P}(X_2 < x_2 | X_1 = x_1) = \int\limits_{(-\infty, x_2)} \frac{h(x_1, r_2)}{f(x_1)} dr_2.$$

Závěrem tohoto odstavce ukážeme, že pro náhodné veličiny X_a, X_b, X_c definované na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$, které nabývají jen celočíselných hodnot větších nebo rovných nule, platí pro všechna j, k, $l \in \mathbf{N}$:

$$\mathcal{P}(X_c = k | X_a = j) = \sum_{i=0}^{\infty} \mathcal{P}(X_c = k | X_b = l, X_a = j).\mathcal{P}(X_b = l | X_a = j). \tag{4.4}$$

Důkaz: Za předpokladu $p(B_1 \cap B_2) > 0$ nejprve ukážeme, že platí:

$$\mathcal{P}(A|B_1 \cap B_2).\mathcal{P}(B_1|B_2) = \mathcal{P}(A \cap B_1|B_2). \tag{4.5}$$

$$\mathcal{P}(A|B_1 \cap B_2).\mathcal{P}(B_1|B_2) = \frac{p(A \cap (B_1 \cap B_2))}{p(B_1 \cap B_2)} \cdot \frac{p(B_1 \cap B_2)}{p(B_2)} = \frac{p((A \cap B_1) \cap B_2)}{p(B_2)} =$$
$$= \mathcal{P}(A \cap B_1|B_2),$$

takže

$$\sum_{l=0}^{\infty} \mathcal{P}(X_c = k | X_b = l, X_a = j).\mathcal{P}(X_b = l | X_a = j) = \sum_{i=0}^{\infty} \mathcal{P}(X_c = k, X_b = l | X_a = j) =$$

$$= \sum_{l=0}^{\infty} \mathcal{P}(\{\omega | X_c(\omega) = k\} \cap \{\omega | X_b(\omega) = l | X_a = j\}) =$$

$$= \mathcal{P}(\bigcup_{l=0}^{\infty}(\{\omega|X_c(\omega)=k\}\cap\{\omega|X_b(\omega)=l\})|X_a=j) =$$

$$= \mathcal{P}(\{\omega|X_c(\omega)=k\}\cap\bigcup_{l=0}^{\infty}\{\omega|X_b(\omega)=l\}|X_a=j) =$$

$$= \mathcal{P}(\{\omega|X_c(\omega)=k\}\cap\Omega|X_a=j) = \mathcal{P}(X_c=k|X_a=j).$$

Pro případ $p(X_a=j)=0$ lze vztah (4.5) dokázat pomocí vztahu (4.3) použitím zobecněné Fubiniho věty.

Rovnice (4.3) se při výpočtech často užívá ve zvláštních tvarech. Je-li X spojitá náhodná veličina s funkcí hustoty f, pak lze rovnici (4.3) psát ve tvaru:

$$p(A\cap X^{-1}[C]) = \int_C \mathcal{P}(A|X=x).f(x)dx, \tag{4.6}$$

je-li X diskrétní náhodná veličina, pak (4.3) přejde ve tvar

$$p(A\cap X^{-1}[C]) = \sum_{x\in C\cap D} \mathcal{P}(A|X=x).g(x), \tag{4.7}$$

kde D je množina všech bodů nespojitosti distribuční funkce náhodné veličiny X a g je její frekvenční funkce.

4.2.2 Markovovy procesy, Markovovy řetězce

Při zkoumání dějů v systémech hromadné obsluhy se často předpokládá, že náhodné procesy, které se tu vyskytují, jsou markovovské.

Markovovy procesy jsou jednoduché případy náhodných procesů, jejichž společnou vlastností je, že splňují tzv. *Markovovu vlastnost*:

$$\forall n\in\mathbf{N}: s,s_0,s_1,s_2,...,s_n\in\Im\wedge t,t_0,t_1,...,t_n\in T\wedge t_n<t_{n-1}<...t_1<t_0<t$$

$$\Rightarrow \mathcal{P}(X(t)=s|X(t_n)=s_n,...,X(t_0)=s_0) = \mathcal{P}(X(t)=s|X(t_0)=s_0), \tag{4.8}$$

tj. že stav $X(t)$ v čase t nezávisí na stavech $X(t')$ v časech t', $t'<t_0<t$, $t',t_0,t\in T$. Proto tyto procesy bývají někdy nazývány náhodnými procesy bez následných účinků.

Časová množina T i množina stavů $\Im$ mohou být jak spočetné, tak i nespočetné. U systémů zkoumaných v teorii hromadné obsluhy se však nejčastěji užívají náhodné procesy s diskrétními stavy, takže $\Im=\{s_1,s_2,...\}$.

Definujeme-li *pravděpodobnosti přechodů*

$$p_{ij}(t_1,t_2) := \mathcal{P}(X(t_2)=s_j|X(t_1)=s_i), \quad i,j\in\mathbf{N}, \quad t_1,t_2\in T \tag{4.9}$$

můžeme pro náhodný proces s Markovovou vlastností na základě (6) a (4) odvodit vztah

$$\begin{aligned} p_{ij}(t_1,t_2) &= \sum_{l=0}^{\infty}\mathcal{P}(X(t_2)=s_j|X(t_0)=s_1,X(t_1)=s_i).\mathcal{P}(X(t_0)=s_l|X(t_l)=s_i) = \\ &= \sum_{l=0}^{\infty}\mathcal{P}(X(t_2)=s_j|X(t_0)=s_l).\mathcal{P}(X(t_0)=s_l|X(t_l)=s_i) = \\ &= \sum_{l=0}^{\infty}p_{lj}(t_0,t_2).p_{il}(t_1,t_0), \quad \text{kde } t_1<t_0<t_2. \end{aligned} \tag{4.10}$$

Markovovy řetězce

Náhodný proces $(X(t))_{t\in T}$ s diskrétním časem a s diskrétními stavy, který má Markovovu vlastnost (6), se nazývá *Markovův řetězec*. Protože T je spočetná množina, položíme v dalším $T = \mathbf{N} = \{0, 1, ...\}$. Pravděpodobnost stavu $s_i \in \Im$ v čase $t \in \mathbf{N}$ označíme symbolem $\pi_i(t) := p(X(t) = s_i)$.

Vektor $\pi(t) := (\pi_1(t), \pi_2(t), ..., \pi_n(t), ...)$, $t \in \mathbf{N}$ se nazývá vektor rozložení stavů Markovova řetězce $(X(t))_{t\in\mathbf{N}}$ v čase t, $\pi(0)$ je tzv. *vektor počátečního rozložení stavů*. Pokud existuje $\lim_{t\to\infty} \pi(t)$, pak symbolem $pi(\infty) = \lim_{t\to\infty}(t)$ označujeme *vektor finálního rozložení stavů*. Pro libovolné $t \in \mathbf{N}$ se definuje stochastická *matice přechodů* $Q(t) = (p_{ij}(t))$, kde:

$$p_{ij}(t) := p_{ij}(t, t + 1).$$

Trojicí $(\Im, Q(t)_{t\in\mathbf{N}}, \pi(0))$ je Markovův řetězec zcela charakterizován. Markovův řetězec se nazývá homogenní, jestliže matice přechodů $Q(t)$ nezávisí na t, tj. $\forall t \in \mathbf{N} : Q(t) = Q(t+1)$.

Zabýváme-li se jen homogenními Markovovými řetězci, pak kromě matice přechodů Q má význam definovat *matici přechodů v n krocích* ($n > 0$):

$$Q_n := (p_{ij}^{(n)}), \text{ kde } p_{ij}^{(n)} := p_{ij}(t, t + n).$$

Pro výpočet Q_n se užívá tzv. *Chapmanova-Kolmogorovova rovnice*, která vyplývá ze vztahu (8). Platí totiž

$$p_{ij}^{(n+m)} = p_{ij}(t, t + n + m) = \sum_{l=0}^{\infty} p_{il}(t, t + n).p_{lj}(t + n, t + n + m) = \sum_{l=0}^{\infty} p_{il}^{(n)} p_{lj}^{(m)} \quad (4.11)$$

a speciálně pro $m = 1$ odtud vyplývá $Q_{n+1} = Q_n.Q$, takže vzhledem k rovnosti $Q_1 = Q$ platí $Q_n = Q^n$.

Platí též $\pi(n) = \pi(0).Q^n$ (neboť $\pi_i(1) = \sum_{j=0}^{\infty} \pi_j(0).p_{ij}$ podle (4.7)).

Symbolem $Q_\infty = \lim_{n\to\infty} Q_n$ označujeme *finální matici přechodů*.

Říkáme, že stav s_j je dosažitelný ze stavu s_i, existuje-li takové číslo $n \in \mathbf{N}$, že $p_{ij}^{(n)} > 0$. Dosažitelnost stavů určuje do značné míry chování systému a lze ji zjišťovat ze stochastické matice přechodů Q. Platí:

(1) Je-li Q stochastická matice, potom Q^n, $n = 1, 2, 3, ...$, je rovněž stochastická matice.

(2) Je-li matice Q tvaru

$$\begin{bmatrix} A & 0 \\ 0 & D \end{bmatrix}$$

kde A a D jsou čtvercové submatice, potom systém nalézající se v některém za stavů A nemůže nikdy přejít do stavu D a obráceně. Takovou matici nazýváme rozložitelnou a množiny stavů uzavřené. Patří sem i matice, které lze shodnou permutací řádků i sloupců uvést do výše uvedeného tvaru.

(3) Je-li matice Q tvaru

$$\begin{bmatrix} A & 0 \\ C & D \end{bmatrix}$$

kde A a D jsou čtvercové submatice, potom pravděpodobnost, že se systém bude nacházet v některém ze stavů D se vzrůstajícím n neroste. Přechod z kteréhokoliv stavu

submatice D do některého ze stavů odpovídajícího A je možný, ale nikoliv obráceně. Stavy submatice D jsou nenávratné. Analogií je matice

$$\begin{bmatrix} A & B \\ 0 & D \end{bmatrix}$$

kdy k nenávratným stavům vedou přechody ze stavů submatice A. I v takovém případě bývá matice P označována jako rozložitelná. Z vlastností (2) a (3) plyne: Matice je nerozložitelná, právě když je každý stav dosažitelný z libovolného jiného stavu. I zde platí poznámka, že stejnou vlastnost má stochastická matice, kterou lze shodnou permutací řádků i sloupců uvést do uvedených tvarů.

(4) Lze-li shodnou permutací řádků i sloupců uvést matici Q do tvaru:

$$\begin{bmatrix} 0 & A_1 & 0 & . & 0 \\ 0 & 0 & A_2 & . & 0 \\ . & . & . & . & . \\ 0 & 0 & 0 & . & A_{n-1} \\ A_n & 0 & 0 & . & 0 \end{bmatrix}$$

kde A_i jsou čtvercové submatice, nazýváme takový systém periodickým s periodou n.

V případě, že stochastická matice Q není rozložitelná nebo periodická, platí, že limita vektoru absolutních pravděpodobností pro n

$$\lim_{n\to\infty} \pi(n) = \pi(\infty) = \pi \tag{4.12}$$

se blíží konečným hodnotám $\pi_0, \pi_1, \pi_2, ...$, nezávislým na počátečním rozložení $\pi(0)$. V tomto případě je systém staticky ustálený a matici Q, která je vytvořena z pravděpodobností přechodů, jež vedou systém do ustáleného stavu, říkáme *ergodická*.

Markovovy procesy

O Markovově procesu $(X(t))_{t\in T}$ mluvíme tehdy, jde-li o náhodný proces s Markovovou vlastností (6), se spojitým časem a diskrétními stavy.

Pro pravděpodobnosti přechodů $p_{ij}(t, t+\tau)$ definované vztahem (7) předpokládejme existenci limit:

$$a_{ij}(t) := lim_{\tau\to 0}\frac{p_{ij}(t,t+\tau)}{\tau} \geq 0, \quad i \neq j \tag{4.13}$$

$$a_{ii}(t) := lim_{\tau\to 0}\frac{1-p_{ii}(t,t+\tau)}{\tau} \geq 0.$$

Prvá z těchto limit představuje intenzitu pravděpodobností přechodů ze stavu i do stavu j, druhá limita je intenzitou pravděpodobností výstupu ze stavu i. Intenzity mohou být obecně závislé na čase t a stavu, ve kterém se systém nachází.

Dále se budeme zabývat pouze *homogenními Markovými procesy*, u nichž intenzity $a_{ij} := a_{ij}(t)$ a pravděpodobnosti přechodů $p_{ij}(\tau) := p_{ij}(t, t+\tau)$ nezávisí na čase.

Pro sledování diskrétních změn stavů ve spojitě probíhajícím čase vyjádříme z (8) Chapmanovy-Kolmogorovovy rovnice v tvaru:

$$p_{ij}(t+\tau) = \sum_{k=0}^{\infty} p_{ik}(t).p_{kj}(\tau) = p_{ij}(t).p_{jj}(\tau) + \sum_{k\neq i} p_{ik}(t).p_{kj}(\tau). \tag{4.14}$$

Celou rovnici vydělíme τ:

$$\frac{p_{ij}(t+\tau)-p_{ij}(t)}{\tau} = -p_{ij}(t)\frac{1-p_{ij}(\tau)}{\tau} + \sum_{k\neq j} p_{ik}(t).\frac{p_{kj}(\tau)}{\tau} \tag{4.15}$$

a pro $\tau \to 0$ za předpokladu homogenity náhodného procesu

$$\frac{d}{dt}p_{ij}(t) = -p_{ij}(t).a_{jj} + \sum_{k\neq j} p_{ik}(t).a_{kj}. \tag{4.16}$$

Obdobně dostaneme z (4.7) pro absolutní pravděpodobnosti $\pi_i(t) = p(X(t) = s_i)$ soustavu lineárních diferenciálních rovnic:

$$\frac{d}{dt}\pi_j(t) = -\pi_j(t).a_{jj} + \sum_{k\neq j} p_k(t).a_{kj} \tag{4.17}$$

Tuto soustavu lze vyjádřit maticově:

$$\frac{d}{dt}\pi(t) = \pi(t)A \tag{4.18}$$

kde kvazistochastická *matice intenzit pravděpodobností přechodů* je:

$$A := \begin{bmatrix} -a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & -a_{22} & ... & a_{2n} \\ ... & & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix} \tag{4.19}$$

Je-li matice A ergodická, pak existuje vektor finálního rozložení stavů

$$\pi(\infty) = \lim_{\tau\to\infty} \pi(t) \tag{4.20}$$

nezávislý na počátečních podmínkách. V tomto případě hovoříme o tom, že se Markovovský proces statisticky ustálil.

Zajímá-li nás v chování systému, který je popsán soustavou diferenciálních rovnic (18), resp. (17), pouze statisticky ustálený stacionární stav, potom limitní pravděpodobnosti vzhledem ke vztahu (20) určíme ze vztahu

$$\pi.A = 0, \tag{4.21}$$

kde složky vektoru π jsou vázány podmínkou

$$\sum_j \pi_j = 1.$$

Rozepíšeme-li maticový součin (21) , obdržíme soustavu lineárních rovnic

$$\begin{aligned} a_{11}.\pi_1 + a_{12}.\pi_2 + ... + a_{1n}.\pi_n &= 0 \\ a_{21}.\pi_1 + a_{22}.\pi_2 + ... + a_{2n}.\pi_n &= 0 \\ ... & \\ a_{n1}.\pi_1 + a_{n2}.\pi_2 + ... + a_{nn}.\pi_n &= 0, \end{aligned} \tag{4.22}$$

jejímž řešením určíme π_j.

Pro zápis soustavy lineárních rovnic (22) nemusíme mít sestavenou kvazistochastickou matici. Stačí, máme-li znázorněn orientovaný graf přechodů, jenž je ohodnocený intenzitami přechodů do následných stavů.

4.3 Modely systému hromadné obsluhy

4.3.1 Kendallova klasifikace

Chceme-li stručně a přehledně vyjádřit typ systému hromadné obsluhy podle jeho základních charakteristik, používáme běžně ustáleného symbolického označení, které v teorii hromadné obsluhy zavedl D.G.Kendall. V jeho klasifikaci jsou systémy tříděny podle tří hlavních hledisek. Podle:

- typu stochastického procesu popisujícího příchod požadavků k obsluze,
- zákona rozložení délky doby obsluhy,
- počtu obslužných linek, jež jsou k dispozici.

Informace o těchto třech charakteristikách je zakódována ve tvaru $X/Y/c$, kde na místě X a Y jsou velká písmena a c je přirozené číslo (popř. ∞), které značí počet obslužných linek. Za X a Y je dosazen vždy jeden ze symbolů M, D, G, E_n, K_n, GI.

Význam těchto symbolů vysvětluje tabulka 4.1.

Tabulka 4.1:

symbol	X	Y
M	Poissonův proces příchodů tj. exponenciální rozložení vzájemně nezávislých intervalů mezi příchody	exponenciální rozložení doby obsluhy
E_k	Erlangovo rozložení intervalů mezi příchody s parametry λ a k	Erlangovo rozložení doby obsluhy s parametry λ a k
K_n	rozložení χ^2 intervalů mezi příchody, n stupňů volnosti	rozložení χ^2 doby obsluhy
D	pravidelné deterministické příchody	konstantní doba obsluhy
G	žádné předpoklady o procesu příchodu	jakékoliv rozložení doby obsluhy
GI	rekurentní proces příchodů	

4.3.2 Markovovské systémy

Systémy $M/M/n$ jsou poměrně jednoduché a snadno analyticky řešitelné. K zachycení jejich hlavních vlastností vystačíme s teorií Markovovských procesů. Systémy $M/M/n$ se v teorii hromadné obsluhy studovaly od samých jejích počátků, a to nejen pro jednoduchou strukturu, ale také proto, že v četných konkrétních případech dávají vhodný teoretický model systémů, jehož prostřednictvím můžeme ověřit výsledky získané simulačním řešením.

Obecné řešení

Předpokládejme homogenní Markovovský systém popsaný intenzitami pravděpodobností přechodů

$$a_{i,i+1} = \lambda_i,\ a_{i+1,i} = \mu_i,\ a_{ii} = \lambda_i + \mu_i \text{ pro } i \in \mathbf{N}$$

Ostatní intenzity jsou nulové.

Stavy $s \in \Im = \mathbf{N}$ se zpravidla interpretují jako počty požadavků v systému. Přechod ze stavu s do stavu $s+1$ pak odpovídá příchodu nového požadavku, přechod do stavu $s-1$ ukončení obsluhy jednoho požadavku. Intervaly mezi příchody mají pro markovovský systém v každém stavu exponenciální rozložení pravděpodobností. Odtud též například vyplývá, že

$$a_{i,i+1} = \lim_{\tau \to 0} \frac{p_{1,i+1}(\tau)}{\tau} = lim_{\tau \to 0} \frac{1}{\tau} \mathcal{P}(X(t+\tau) = i+1 | X(t) = i) =$$

$$\lim_{\tau \to 0} \frac{1}{\tau} p(\ \text{"v intervalu } (t, t+\tau) \text{ přijde 1 požadavek"}\ ) =$$

$$\lim_{\tau \to 0} \frac{1}{\tau} p(\ \text{"interval mezi příchody požadavků"}\ < \tau) =$$

$$\lim_{\tau \to 0} \frac{1}{\tau} F(\tau) = \lim_{\tau \to 0} \frac{1 - e^{-\lambda_i \tau}}{\tau} = \lambda_i.$$

Označíme-li vektor rozložení stavů $p(t) = (p_1(t), p_2(t), ...)$, můžeme podle (17) přímo psát rovnice popisující tento systém:

$$\frac{d}{dt} p_0(t) = -\lambda_0 p_0(t) + \mu_1 p_1(t)$$

$$\frac{d}{dt} p_k(t) = \lambda_{k-1} p_{k-1}(t) - (\lambda_k + \mu_k).p_k(t) + \mu_{k+1} p_{k+1}(t) \quad \forall k > 0. \tag{4.23}$$

Pravděpodobnosti $p_k(t)$ dostaneme řešením této soustavy rovnic s přihlédnutím k podmínce $\sum_{k=0}^{\infty} p_k(t) = 1$ a k počáteční podmínce dané rozložením pravděpodobností $p_k(0)$, $k = 1, 0, 2, ...$ Mnohdy se zajímáme jen o limitní chování systému při $t \to \infty$, tehdy zejména o limity $\lim_{t \to \infty} p_k(t) = p_k$, $k = 0, 1, 2, ...$ Ty vyjadřují rozložení pravděpodobností určitého počtu požadavků v systému, který je v daných podmínkách bez přerušení po dlouhou dobu a stačil se stabilizovat; změny v systému probíhají podle ustáleného řádu, jenž je nezávislý na stavu systému. Limity $p_k i$ dostaneme tak, že položíme ve vztahu (23) všechny derivace na levých stranách rovny nule a na pravých stranách odstraníme závislosti na t. Dospějeme tak k soustavě lineárních algebraických rovnic

$$\lambda_{k-1} p_{k-1} - (\lambda_k + \mu_k) p_k + \mu_{k+1}.p_{k+1} = 0 \quad \forall k > 0$$

$$-\lambda_0.p_0 + \mu_1.p_1 = 0, \tag{4.24}$$

k nimž je nutno připojit podmínku $\sum_{k=0}^{\infty} p_k = 1$, aby řešení mělo pravděpodobnostní význam. Soustavu budeme řešit zavedením substituce:

$$z_k := -\lambda_k p_k + \mu_{k+1} p_{k+1} \quad \forall k \in \mathbf{N}. \tag{4.25}$$

Soustava (24) tak nebude tvaru:

$$z_0 = 0$$

$$z_k - z_{k-1} = 0 \quad \forall k > 0. \tag{4.26}$$

Řešení této soustavy je triviální, $z_k = 0$ pro všechna $k \in \mathbf{N}$. To ovšem znamená, že podle (25)

$$p_{k+1} = \frac{\lambda_k}{\mu_{k+1}}.p_k \quad \forall k \in \mathbf{N} \tag{4.27}$$

a tedy

$$p_k = \frac{\lambda_{k-1}...\lambda_1\lambda_0}{\mu_k...\mu_2\mu_1}p_0 = p_0\frac{\prod_{i=0}^{k-1}\lambda_i}{\prod_{i=1}^{k}\mu_i} \quad \forall k > 0. \tag{4.28}$$

Přitom ovšem musí být splněna podmínka $\sum_{k=0}^{\infty} p_k = 1$, tzn.

$$p_0(1 + \sum_{k=1}^{\infty}\left(\frac{\prod_{i=0}^{k-1}\lambda_i}{\prod_{i=1}^{k}\mu_i}\right)) = 1.$$

Označme $S := 1 + \sum_{k=1}^{\infty}\left(\frac{\prod_{i=0}^{k-1}\lambda_i}{\prod_{i=1}^{k}\mu_i}\right)$. Pokud tento součet existuje, je:

$$p_0 = S^{-1}$$
$$p_k = \frac{\prod_{i=0}^{k-1}\lambda_i}{\prod_{i=1}^{k}\mu_i}S^{-1} \quad \forall k > 0. \tag{4.29}$$

Vidíme, že nutnou podmínkou pro existenci ustáleného rozložení stavů $p(\infty)$, pro které byly sestaveny rovnice (24), je aby $S < \infty$. Pro posouzení této podmínky lze využít řady

$$S(z) = 1 + \sum_{k=1}^{\infty}\frac{\prod_{i=0}^{k-1}\lambda_i}{\prod_{i=1}^{k}\mu_i}z^k. \tag{4.30}$$

Je-li R poloměr konvergence řady $S(z)$ a platí-li $R > 1$, pak $S = S(1)$ existuje a systém má ustálený stav. Pro $R < 1$ je systém nestabilní, pro $R = 1$ nelze obecně závěr o stabilitě učinit. Tyto důvody vedou k zavedení veličiny $\rho := R$, zvané využití obsluhy. Podmínka stabilizovatelnosti systému je $\rho < 1$. Hodnotu ρ lze stanovit např. pomocí Cauchyova-Hadamardova kritéria:

$$\rho = \limsup_{n\to\infty}\left|\frac{\prod_{0}^{n}\lambda_i}{\prod_{1}^{n+1}\mu_i} : \frac{\prod_{0}^{n-1}\lambda_i}{\prod_{1}^{n}\mu_i}\right| = \limsup_{n\to\infty}\frac{\lambda_n}{\mu_{n+1}}. \tag{4.31}$$

Systém M/M/1

Systém $M/M/1$ je charakterizován takto: požadavky přicházejí do systému jednotlivě, navzájem nezávisle a nezávisle na chodu vlastní obsluhy, ato tak, že intervaly mezi jejich příchody jsou nezávislé náhodné veličiny s exponenciálním zákonem rozložení pravděpodobností s konstantním parametrem λ, $0 < \lambda < \infty$; parametr λ nezávisí ani na čase, ani na

okamžitém stavu systému. Požadavky, které nemohou být okamžitě obslouženy proto, že jediná obslužná linka je právě obsazena, čekají v jednoduché neomezené frontě. Do fronty se řadí tak, jak přišly, bez předbíhání a předností, z fronty předčasně neodcházejí. Délka obsluhy je náhodná veličina s exponenciálním rozložením pravděpodobností, s parametrem $\mu = 1/T_0$, $0 < \mu < \infty$. Po skončení obsluhy požadavky systém ihned opouštějí.

Označme $X(t)$ počet požadavků, které jsou v okamžiku t právě ve sledovaném systému. Je-li $X(t) = 0$, je systém prázdný a obsluha stojí; při $X(T) > 0$ je jeden požadavek obsluhován a ostatní čekají ve frontě. Vzhledem k náhodnému charakteru systému tvoří $(X(t))$ stochastický proces. Proces $(X(t))$ je homogenní a markovovský. Je-li v okamžiku $t > 0$ skutečná hodnota $X(t) = j$, může se tato hodnota změnit příchodem dalšího požadavku na $j + 1$ anebo (je-li $j > 0$) dokončením právě probíhající obsluhy na $j - 1$.

V souladu s označením v odstavci 4.3.2.1. lze tedy tento systém popsat intenzitami pravděpodobností přechodů

$$\lambda_i = \lambda, \quad \mu_{i+1} = \mu \quad \text{pro všechna } i \in \mathbf{N}.$$

Pro určení ustálených pravděpodobností nejprve stanovíme

$$S = 1 + \sum_{k=1}^{\infty} \left(\frac{\lambda}{\mu}\right)^k = \sum_{k=0}^{\infty} \rho^k = (1-\rho)^{-1},$$

kde $\rho = \lambda/\mu$ ve shodě s (31). Hledané pravděpodobnostní rozložení stavů tedy je

$$p_k = \rho^k(1-\rho).$$

Počet požadavků ve stabilizovaném systému má tedy geometrické rozložení pravděpodobností s parametrem ρ. Na základě pravděpodobností p_k určíme další charakteristiky systému.

Průměrný počet požadavků v systému

$$L_q = \sum_{k=1}^{\infty} k p_k = \rho(1-\rho) \sum_{k=1}^{\infty} k\rho^{k-1} = \rho(1-\rho)^{-1}$$

a rozptyl tohoto počtu

$$\sum_{k=1}^{\infty} k^2 p_k - \rho^2(1-\rho)^{-2} = \rho(1-\rho)^{-2}.$$

Průměrný počet požadavků ve frontě

$$L_w = \sum_{k=1}^{\infty} k p_{k+1} = (1-\rho)\rho^2 \sum_{k=1}^{\infty} k\rho^{k-1} = \rho^2(1-\rho)^{-1}$$

a rozptyl tohoto počtu

$$\rho^2(1+\rho-\rho^2)(1-\rho)^{-2}.$$

Označme $t_w k$ dobu, po kterou bude muset čekat ve frontě požadavek, jenž byl v okamžiku t' svého příchodu do systému právě k-tým ve frontě; tento jev má pravděpodobnost p_k. Doba $t_w k$ je ovšem náhodná, skládá se jednak ze zbytku doby obsluhy požadavku, který byl v okamžiku t' právě v obsluze a dále z úplných dob obsluhy všech $k - 1$ požadavků stojících ve frontě před naším požadavkem. To vše jsou navzájem nezávislé náhodné veličiny

a každá z nich má exponenciální rozložení pravděpodobností s parametrem μ. Veličina $t_w k$ má rozložení Erlangovo s parametry μ a k. Tedy

$$p(t_w k > w) = e^{-\mu w} \sum_{j=0}^{k-1} \frac{(\mu w)^j}{j!}, \qquad k = 1, 2, ...; \quad w \geq 0,$$

neboť $t_w k > w$ znamená, že v intervalu délky w byla ukončena obsluha nejvýše $k-1$ požadavků a rozložení počtu obchodů je Poissonovská. Požadavek, který nalezl systém prázdný, byl vzat do obsluhy bez čekání:

$$p(t_w 0 = 0) = 1.$$

Podle vzorce (3") o úplné pravděpodobnosti určíme rozložení doby čekání t_w libovolného náhodně vybraného požadavku; pro $w \geq 0$

$$\begin{aligned} p(t_w > w) &= \sum_{k=0}^{\infty} p_k p(t_w k > w) = \\ &= \sum_{k=1}^{\infty} (1-\rho)\rho^k e^{-\mu w} \sum_{j=1}^{k-1} \frac{(\mu w)^j}{j!} = \\ &= (1-\rho)e^{-\mu w}\rho \sum_{j=0}^{\infty} \left[\frac{(\lambda w)^j}{j!}\right] \sum_{r=j}^{\infty} \rho^r = \\ &= \rho e^{-\mu w} e^{\lambda w} = \rho e^{-(\mu-\lambda)w}. \end{aligned}$$

Přitom $p(t_w = 0) = p_0 = 1 - \rho$.

Takže zákon rozložení pročekané doby se skládá ze dvou složek: z diskrétního skoku o velikosti $1-\rho$ v bodě 0 a ze spojité složky exponenciálního typu s parametrem $\mu - \lambda$ pro $w \geq 0$. Pravděpodobnost $p(t_w < 0)$ je rovna nule. Tomuto rozdělení odpovídá střední hodnota

$$T_w = E(t_w) = \rho(\mu - \lambda)^{-1}$$

a její rozptyl

$$D(t_w) = \rho(2-\rho)(\mu-\lambda)^{-2}.$$

Celková doba t_q strávená požadavkem v systému se od doby t_w liší jen o dobu jeho vlastní obsluhy, což je opět náhodná veličina s exponenciálním rozložením pravděpodobností s parametrem μ, nezávislá na t_w. Pro $w \geq 0$ dostáváme analogicky jako jako pro t_w vztah

$$p(t_q > w) = \sum_{k=0}^{\infty} p_k e^{-\mu w} \sum_{j=0}^{k} \frac{(\mu w)^j}{j!} = \sum_{j=0}^{\infty}\sum_{r=0}^{\infty} (1-\rho)e^{-\mu w}\rho^r \rho^j \frac{(\mu w)^j}{j!} = e^{-(\mu-\lambda)w}.$$

Střední hodnota veličiny t_q:

$$T_q = E(t_q) = (\mu - \lambda)^{-1} \text{ a její rozptyl je } (\mu-\lambda)^{-2}.$$

Toto odvození zákonů rozložení pravděpodobností dob t_w a t_q platí pouze pro stabilizovaný systém, a tedy nutně předpokládá nerovnost $\lambda < \mu$. Tato nerovnost znamená, že při plném vytížení je obslužná linka schopná za jednotku času obsloužit v průměru více požadavků, než jich do systému přijde. Naopak nerovnost $\lambda > \mu$ by znamenala, že celková kapacita obslužné linky je nedostatečná, linka nestačí zvládnout tolik požadavků, kolik jich požaduje obsluhu; v tomto případě fronta neomezeně narůstá.

Mohlo by se zdát, že ideální by měla být rovnost $\lambda = \mu$, ale ve skutečnosti napjatá časová bilance takového systému bez rezerv vede k tomu, že se systém nestabilizuje; každá časová ztráta se tu jeví nakonec jako nenahraditelná.

Pravděpodobnosti p_k a rozložení t_w a t_q jsou hlavními charakteristikami, které zajímají zákazníky. Z hlediska provozovatele systému je však zajímavé i využívání systému. Obslužná linka pracuje průměrně 60ρ minut v hodině. Využívá se jí na 100ρ %.

Jednou z často zkoumaných charakteristik systémů hromadné obsluhy jsou též rozložení délky period nepřetržitého chodu obsluhy. Pro systém $M/M/1$ lze toto rozložení odvodit úvahou: Vezmeme homogenní markovovský proces Y stejného typu, jako je proces X popisující chod systému $M/M/1$ jenom s tím rozdílem, že v procesu Y je intenzita přechodu $a_{01} = a$.

Příslušné pravděpodobnosti $p_k(t)$ procesu Y vyhovují rovnicím:

$$\begin{aligned}
\frac{d}{dt}p_0(t) &= \mu p_1(t), \\
\frac{d}{dt}p_1(t) &= -(\lambda+\mu)p_1(t) + \mu p_2(t), \\
\frac{d}{dt}p_k(t) &= \lambda p_{k-1}(t) - (\lambda+\mu)p_k(t) + \mu p_{k+1}(t) \quad k = 2, 3, ...
\end{aligned}$$

V kladných hodnotách probíhá proces Y podle stejných zákonitostí jako proces X, jakmile však v některém okamžiku dospěje k hodnotě nula, zůstaneme v tomto stavu navždy.

Řešme tuto soustavu rovnic s podmínkou $\sum_k p_k(t) = 1$ pro počáteční podmínku $p_1(0) = 1$.

Potom $p_0(t)$ bude pravděpodobnost, že se během intervalu $(0, t\rangle$ proces Y dostal z počáteční hodnoty $Y(0) = 1$ do hodnoty $Y(t) = 0$. Tedy $p_0(t)$ je přímo distribuční funkcí periody nepřetržité obsluhy systému $M/M/1$.

Střední hodnota doby nepřetržitého chodu obsluhy je rovna $(\mu - \lambda)^{-1}$ a její rozptyl je $(\mu + \lambda)/(\mu - \lambda)^{-3}$.

Mezi intervaly nepřetržitého chodu obsluhy leží intervaly, v nichž obslužná linka nepracuje. Ty mají ovšem všechny stejné exponenciální rozložení s parametrem λ, takže jejich průměrná délka je $\frac{1}{\lambda}$.

Frontové řády

Frontovým řádem nazýváme souhrn pravidel, jež určují, jak se v daném systému hromadné obsluhy požadavky řadí do fronty a jak jsou z ní vybírány do vlastní obsluhy.

Od *řádného frontového řádu*, v němž se zachovává pořadí požadavků v průběhu celé doby od okamžiku vstupu do systému až do okamžiku odchodu požadavku po ukončení obsluhy, se mohou jiné řády lišit jednak pravidly řazení přicházejících požadavků do fronty, jednak pravidly výběru požadavků z fronty do obsluhy. Jestliže se při uvolnění obslužné linky bere jako další ten požadavek, který stojí právě ve frontě na posledním místě, jde o *inverzní frontový řád*.

Při studiu systémů s inverzním frontovým řádem je třeba si především uvědomit, že řada vlastností systému $M/M/1$ se nijak nezmění, změníme-li v něm řádný frontový řád za inverzní. Samotný proces $(X(t))_{t\in T}$ zachycuje pouze celkový počet požadavků v systému, ale nijak neregistruje jejich pořadí. Nezmění se ani pravděpodobnosti $p_{jk}(\tau)$, ani limity p_k. Stejný bude i průměrný počet požadavků ve stabilizovaném systému, i průměrný počet požadavků ve frontě. Změní se však rozložení doby čekání požadavků ve frontě, resp. doby strávené požadavkem v systému. Požadavek Z, který při svém příchodu zastane obslužnou linku obsazenou jiným požadavkem, musí nejprve počkat, až skončí obsluha; tato doba je

náhodná a má exponenciální rozložení pravděpodobnosti s parametrem μ. Může se stát, že mezitím přijdou do systému další požadavky ještě dříve, než požadavek přijde na řadu. Požadavek Z musí čekat tak dlouho, až se zcela vyčerpá řada novějších požadavků a obslužná linka se uvolní. Kdyby v systému nebyl požadavek Z ani žádný jiný požadavek "starší", zastavila by se v takovém případě obsluha; jsme tu ve stejné situaci s jakou jsme se setkali při odvozování rozložení délek period nepřetržitého chodu obsluhy. V případě inverzního frontového řádu mají náhodné veličiny $t_{wk}, k = 1, 2, 3, ...$ všechny stejné rozložení pravděpodobnosti, nezávislé na k:

$$p(t_w = 0) = p_0 = 1 - \rho$$

$$p(t_w > w) = \rho[1 - F(w)] \qquad w \geq 0,$$

F je distribuční funkce délek period nepřetržitého chodu obsluhy. Vztah $T_w = \rho(\mu - \lambda)^{-1}$, tj. střední doba, kterou stráví požadavek čekáním ve frontě, je při inverzním frontovém řádu stejná, jako při řádném řádu. Rozptyl $D(t_w) = \lambda(\rho^2 - \rho + 2)(\mu - \lambda)^{-3}$ je větší, než rozptyl při řádném frontovém řádu.

Jiným způsobem výběru požadavků z fronty je obsluha v náhodném pořadí.

Ve všech těchto případech dostaneme rozložení náhodné veličiny t_q, celkové doby strávené požadavkem v systému, jako součet t_w a doby obsluhy, tedy

$$T_q = T_w + \mu^{-1} = T_w + T_0 = (\mu - \lambda)^{-1},$$

$$D(t_q) = D(t_w) + \mu^{-2}.$$

Dalšími typy frontových řádů jsou ty, u nichž se mění pravidla řazení přicházejících požadavků do fronty. Nejznámějším a nejvýznamnějším příkladem jsou systémy s prioritami. Požadavky jsou rozděleny do několika kategorií; uvnitř každé kategorie jsou požadavky rovnocenné, ale mezi kategoriemi platí určitá hierarchie předností, která se projevuje tím, že se požadavky s vyšší prioritou řadí před všechny požadavky s prioritou nižší. Bude-li v době příchodu požadavku s vyšší prioritou obsluhován požadavek s prioritou nižší, mohou nastat tyto případy:

(1) jeho obsluha je normálně dokončena (slabé přednosti)

(2) jeho obsluha je přerušena a do obsluhy je přijat požadavek s vyšší prioritou (silné nebo absolutní přednosti) a při tom:

   (a) požadavek, jehož obsluha byla přerušena, odchází vůbec ze systému, anebo

   (b) se vrací do fronty a když se dostane na řadu, pak se

      * pokračuje v obsluze tam, kde se přestalo,
      * začíná v obsluze znovu od začátku.

   Existuje ještě další varianta osudu čekajících požadavků: dobrovolné odchody z fronty. Ty mohou být dvou typů:

   (a) ihned při vstupu - požadavek se do fronty vůbec nezařadí (rezignace),

   (b) po určité době čekání požadavek ztratí trpělivost a frontu opustí (odpadnutí).

Systémy $M/M/n$

Pro stabilizaci systému $M/M/1$ musela být splněna podmínka $\rho < 1$, čili $\lambda < \mu$, jinak fronta neomezeně roste. Převýší-li však požadavky možnosti obsluhy, lze řešit tuto situaci v podstatě třemi způsoby:

(1) zvýšit intenzitu obsluhy, tj. docílit vyššího parametru,

(2) extenzívně rozšířit obsluhu, tj. zvýšit počet obslužných linek,

(3) omezit příliv požadavků.

Druhá cesta vede k vytvoření systému s vyšším počtem obslužných linek. V našem případě je to systém $M/M/n$, $1 \leq n \leq \infty$ s řádným frontovým řádem. Požadavkům je k dispozici n nezávislých a rovnocenných obslužných linek; požadavky čekají ve frontě jen tehdy, jsou-li všechny linky obsazeny. Fronta je jedna, společná pro všechny linky.

Uvažujme opět o intenzitách přechodů. Při jakémkoli počtu požadavků v sytému je intenzita $\lambda_i = \lambda$ neměnná. Jinak je tomu ovšem s intenzitami přechodů μ_i. Pro $i \geq n$ je obsluhováno n požadavků, takže intenzita odchodů požadavků je $n\mu$, je-li μ intenzita odchodů z jednoho místa obsluhy. Pro $i < n$ je pouze i obsluh činných, tzn. $\mu_i = i\mu$. To znamená, že

$$\forall i \; \lambda_i = \lambda, \quad \mu_i = i\mu \text{ pro } i < n, \quad \mu_i = n\mu \text{ pro } i \geq n.$$

Graf přechodů je na *obr.* **??**

Pro získání rozložení počtu požadavků v systému opět nejprve stanovíme

$$\rho = \frac{\lambda}{n\mu},$$

$$S = 1 + \sum_{k=0}^{n-1} \frac{\lambda^k}{k!\mu^k} + \sum_{k=n}^{\infty} \frac{\lambda^k}{n!n^{k-n}\mu^k}, \qquad \text{takže položíme} - \text{li } \omega := \frac{\lambda}{\mu},$$

$$p_0^{-1} = \sum_{k=0}^{n-1} \frac{\omega^k}{k!} + \frac{n^n}{n!} \sum_{k=n}^{\infty} (\frac{\omega}{n})^k.$$

Dále

$$\begin{aligned} p_k &= \frac{\omega^k}{k!} p_0 \quad (k \leq n), \\ p_k &= \frac{\omega^k}{n!n^{k-n}} p_0 = \rho^{k-n} p_n \quad (k > n). \end{aligned} \tag{4.32}$$

Z pravděpodobností p_k vypočteme další charakteristiky stabilizovaného systému.

Pravděpodobnost okamžité obsluhy požadavku bez čekání:

$$\sum_{k=0}^{n-1} p_k = p_0 \sum_{k=0}^{n-1} \frac{\omega^k}{k!}.$$

Požadavek bude čekat s pravděpodobností:

$$\pi = \sum_{k=n}^{\infty} p_k = p_0 \frac{\omega^n}{n!} \sum_{k=0}^{\infty} (\frac{\omega}{n})^k = p_0 \frac{\omega^n}{n!} (1-\rho)^{-1} = p_n (1-\rho)^{-1}.$$

Průměrný počet požadavků ve frontě:

$$L_w = \sum_{r=0}^{\infty} r p_{n+r} = p_n \sum_{r=0}^{\infty} r\rho^r = p_n \rho (1-\rho)^{-2} =$$

$$= p_0 \frac{\omega^{n+1}}{(n-1)!} \frac{1}{(n-\omega)^2}.$$

Průměrný počet obsazených linek:

$$\nu = \sum_{k=1}^{n} k p_k + \sum_{k=n+1}^{\infty} n p_k = p_0 \sum_{k=1}^{n} k \frac{\omega^k}{k!} + p_0 \sum_{k=n+1}^{\infty} n\omega^k \frac{n^{n-k}}{n!} =$$

$$= p_0 \omega \left[\sum_{k=0}^{n-1} \frac{\omega^k}{k!} + \frac{n^n}{n!} \sum_{k=n}^{\infty} \left(\frac{\omega}{n}\right)^k \right] = \omega = \frac{\lambda}{\mu} = n\rho.$$

Volných obslužných linek je průměrně $n - \nu = n(1-\rho)$.

Průměrný počet požadavků v systému:

$$L_q = \sum_{k=0}^{\infty} k p_k = \sum_{k=1}^{n} k p_k + \sum_{k=n+1}^{\infty} k p_k =$$

$$= \sum_{k=1}^{n} k p_k + \sum_{k=n+1}^{\infty} n p_k + \sum_{k=n+1}^{\infty} (k-n).p_k = \omega + L_w$$

Rozložení doby čekání t_w: Odvodili jsme, že požadavek bude čekat s pravděpodobností π, čili $p(t_w > 0) = \pi = 1 - p(t_w = 0)$. Označme $t_w k, k = 1, 2, 3, ...$ dobu, kterou pročeká požadavek, jenž se v okamžiku svého příchodu do systému postaví na k-té místo ve frontě, tzn. že zůstane v systému právě $k+n-1$ požadavků, n v obsluze, $k-1$ ve frontě. Pro $w \geq 0$ je

$$p(t_w > w) = \sum_{k=1}^{\infty} p_{n+k-1} p(t_{wk} > w), \tag{4.33}$$

$p(t_{wk} > w)$ je pravděpodobnost, že během časového intervalu délky w skončí obsluhu nejvýš $k-1$ požadavků, přičemž n linek je stále v provozu. Počet ukončených obsluh za těchto podmínek má Poissonovo rozložení pravděpodobností s parametrem $n\mu w$, takže

$$p(t_{wk} > w) = \sum_{j=0}^{k-1} e^{-n\mu w} \frac{(n\mu w)^j}{j!}. \tag{4.34}$$

Dosazením do (33) za p_{n+k-1} z (32) a za $p(t_{wk} > w)$ z (34) dostaneme pro $w > 0$:

$$\begin{aligned} p(t_w > w) &= \sum_{k=0}^{\infty} (\frac{\omega}{n})^k p_n \sum_{j=0}^{k} e^{-n\mu w} \frac{(n\mu w)^j}{j!} = \\ &= p_n e^{-n\mu w} \sum_{k=0}^{\infty} \sum_{j=0}^{k} (\frac{\omega}{n})^k \frac{(n\mu w)^j}{j!} = \\ &= p_n e^{-n\mu w} \sum_{j=0}^{\infty} \left[\frac{(n\mu w)^j}{j!} \right] \sum_{k=j}^{\infty} (\frac{\omega}{n})^k = \\ &= p_n e^{-n\mu w} \sum_{j=0}^{\infty} \left[\frac{(\frac{n\mu w \omega}{n})^j}{j!} \right] (1 - \frac{\omega}{n})^{-1} = \\ &= p_n (1 - \frac{\omega}{n})^{-1} e^{-(n\mu - \lambda)w} = \pi e^{-(n\mu - \lambda)w}. \end{aligned} \tag{4.35}$$

Doba čekání na obsluhu se skládá z diskrétní složky (skok o velikosti $1 - \pi$ v bodě nula) a ze spojité složky exponenciální (35) s parametrem $n\mu - \lambda$. Střední hodnota je

$$T_w = E(t_w) = \frac{\pi}{n\mu - \lambda},$$

rozptyl

$$D(t_w) = \frac{\pi(2 - \pi)}{n\mu - \lambda}.$$

Obdobné charakteristiky bychom mohli odvodit i pro náhodnou veličinu t_q — celkovou dobu, kterou stráví požadavek v systému.

Systém $M/M/n$ se v některých směrech liší jen nepodstatně od systému $M/M/1$ s n-násobným parametrem rozložení doby obsluhy. Ty vlastnosti systému, které jsou nezávislé na osudu požadavků po jejich vstupu do obsluhy, zůstávají stejné, ať je zvýšení kapacity dosaženo extenzívně nebo intenzivně. Platí to např. pro délku fronty, dobu čekání t_w i pro rozložení period nepřetržitého chodu obsluhy. V systémech $M/M/\infty$ je každý požadavek obsloužen ihned, bez čekání. Takovému systému se můžeme libovolně přiblížit, volíme n dostatečně velké, aby některá charakteristika (např. T_w) byla dostatečně malá, resp. blízká obdobné charakteristice systému $M/M/\infty$.

Pro systémy $M/M/\infty$ jsou pravděpodobnosti

$$p_k = p_0 \frac{\omega^k}{k!} = e^{-\omega} \frac{\omega^k}{k!} \qquad k = 0, 1, 2, ...$$

Při rozhodování o počtu n se střetávají zájmy zákazníků a provozovatele obsluhy. Jednou cestou k řešení je konstrukce systémů hromadné obsluhy s proměnným počtem obslužných linek. V systému je vedle určitého počtu n linek, které jsou neustále k dispozici, ještě několik dalších, které se uvádějí do činnosti podle potřeby tak, aby fronta čekajících požadavků nepřekročila určitou mez.

Systémy s omezenou délkou fronty

Zavedení možnosti ztrát požadavků v daném systému hromadné obsluhy může být jednou z cest, jak řešit rozpor mezi nedostatečnou kapacitou systému a požadavky na obsluhu.

Uvažujeme systém typu $M/M/n$ s parametry λ a μ, ve kterém je maximální počet požadavků čekajících ve frontě dán číslem $r \geq 0$: každý požadavek, který přijde v okamžiku,

kdy délka fronty je právě r, je odmítnut. Počet požadavků v systému se může měnit od nuly do $n+r$. Všechny pravděpodobnosti $p_k(t)$ pro $k > n+r$ jsou nulové, intenzity přechodů jsou (viz *obr.* **??**)

$\lambda_k = \lambda$ pro $k < n+r$, $\qquad$ $\lambda_k = 0$ pro $k \geq n+r$,
$\mu_k = k\mu$ pro $k < n$, $\qquad$ $\mu_k = n\mu$ pro $n \leq k \leq n+r$,
$\mu_k = 0$ pro $k > n+r$,

Řešení tohoto systému získáme, položíme-li $\omega := \frac{\lambda}{\mu}$,

$$p_0^{-1} = S = \sum_{k=0}^{\infty} \frac{\omega^k}{k!} + \frac{\omega^n}{n!} \sum_{k=1}^{r} (\frac{\omega}{n})^k,$$

$$p_k = p_0 \frac{\omega^k}{k!} \quad \text{pro } 0 \leq k \leq n,$$

$$p_k = p_0 \frac{\omega^k}{n! n^{k-n}} = p_n (\frac{\omega}{n})^{k-n} \quad \text{pro } n \leq k \leq n+r.$$

p_{n+r} znamená pravděpodobnost ztráty požadavku,
$\sum_{k=0}^{n-1} p_k$ pravděpodobnost, že požadavek bude obsloužen bez čekání,
$\sum_{k=0}^{r-1} p_{n+k}$ pravděpodobnost, že bude požadavek nucen čekat,
$L_q = \sum_{k=1}^{n+r} k p_k$ průměrný počet požadavků v systému,
$L_w = \sum_{k=1}^{r} k p_{n+k}$ průměrný počet požadavků ve frontě,
$\nu = \sum_{k=1}^{n} k p_k + n \sum_{k=1}^{r} p_{n+k} = \omega(1 - p_{n+r})$ průměrný počet obsazených obslužných linek.
Charakteristiky vztahující se k době čekání t_w je třeba brát jako podmíněné — za podmínky, že nedošlo ke ztrátě požadavku.

Doby čekání t_{wk} mají stejná rozložení pravděpodobnosti jako v systému s neomezenou frontou. Pro $w \geq 0, k = 1, 2, ..., r$ dostáváme rovnici

$$\begin{aligned} p(t_w > w) &= \sum_{k=1}^{r} c p_{n+k-1} p(t_{wk} > w) = \sum_{k=0}^{r-1} c p_n (\frac{\omega}{n})^k \sum_{j=0}^{k} e^{-n\mu w} \frac{(n\mu w)^j}{j} \hat{=} \\ &\hat{=} c p_n \sum_{k=0}^{r-1} (\frac{\omega}{n})^k \int_w^{\infty} e^{-n\mu y} \frac{(n\mu y)^k}{k!} n\mu dy = \\ &= c p_n \int_{n\mu w}^{\infty} e^{-z} \sum_{k=0}^{r-1} \frac{(\frac{\omega}{n} z)^k}{k!} dz, \end{aligned} \tag{4.36}$$

kde za c dosadíme buď $(1-p_{n+r})^{-1}$, uvažujeme-li rozložení doby t_w za podmínky, že nedošlo ke ztrátě, nebo , je-li podmínkou, že $\left[\sum_{j=0}^{r-1} p_{n+j}\right]^{-1}$, požadavek skutečně čekal. V prvním

případě je třeba k (36) připojit $p(t_w = 0) = \sum_{k=0}^{n-1} p_k / \sum_{k=0}^{n+r-1} p_k$, ve druhém případě $p(t_w = 0) = 0$.

Rovnost ($\hat{=}$) plyne ze vztahu $\int_x^\infty e^{-z} \frac{z^k}{k!} dz = \sum_{j=0}^{k} e^{-x} \frac{x^j}{j!}$, který lze dokázat postupnou integrací per partes.

Pro první případ bude platit, je-li $\rho = \frac{\lambda}{(n\mu)}$:

$$T_w = E(t_w) = \sum_{k=0}^{r-1} c p_{n+k} \frac{(k+1)}{(n\mu)} = \frac{\frac{\omega^n}{n!} \sum_{k=0}^{r-1} (k+1)\rho^k}{n\mu \left[\sum_{k=0}^{n-1} \frac{\omega^k}{k!} + \frac{\omega^n}{n!}(1-\rho^r)(1-\rho)^{-1} \right]}.$$

Ve druhém případě

$$T_w = E(t_w) = \frac{1}{n\mu} \frac{1-\rho}{1-\rho^r} \sum_{k=1}^{r} k\rho^{k-1} = \frac{1}{n\mu} \left(\frac{1}{1-\rho} - \frac{r\rho^r}{1-\rho^r} \right).$$

Při $r \to \infty$ dostaneme systém $M/M/n$ s neomezenou frontou. Druhým mezním případem je $r = 0$, v tomto případě dostaneme řešení

$$p_k = \frac{\omega^k}{k!} \left[\sum_{j=0}^{n} \frac{\omega^j}{j!} \right]^{-1}, \qquad k = 0, 1, 2, ..., n.$$

Pravděpodobnost ztráty je p_n, průměrný počet požadavků v systému je roven počtu obsazených linek:

$$L_q = \nu = \sum_{k=1}^{n} k.p_k = \omega(1 - p_n).$$

Kapitola 5

Modelování náhodných jevů a metoda Monte Carlo

5.1 Náhodné veličiny

K popisu náhodných hromadných dějů potřebujeme obecně číselné údaje; přitom tyto údaje nejsou konstantní, nýbrž vykazují náhodné výchylky. Takovou náhodnou číselnou hodnotou je například výsledek jednoho hodu při hře s kostkou; jiným příkladem může být třeba počet volání, která přijdou na telefonní ústřednu v daném časovém intervalu, nebo počet atomů radioaktivní látky, které se v daném časovém intervalu rozpadnou.

Chceme-li charakterizovat nějaký číselný údaj, musíme vědět, které hodnoty přicházejí pro něj v úvahu a s jakými pravděpodobnostmi těchto hodnot nabývá. Veličiny tohoto druhu, závisející na náhodě, se nazývají *náhodnými veličinami*. Náhodné veličiny používáme převážně tam, kde nevystačíme jen s otázkou, zda *náhodný jev* nastal nebo nenastal. Definici náhodné veličiny $X : \Omega \to \mathbf{R}$ definované na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$ jsme podali v odstavci 4.2.1.

K vyšetřování náhodné veličiny X na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$ jsme definovali její pravděpodobnostní rozložení prostřednictvím distribuční funkce

$$F : \mathbf{R} \to \langle 0, 1\rangle,\ F(x) = p(X < x)$$

$(p(X < x) := p(\{\omega \in \Omega | X(\omega) < x\}))$. Připomeňme, že distribuční funkce F náhodné veličiny X na pravděpodobnostním prostoru $(\Omega, \mathcal{R}, p)$ má tyto vlastnosti:

1. F je neklesající funkce
2. $\lim_{x \to -\infty} F(x) = 0$
3. $\lim_{x \to \infty} F(x) = 1$
4. $\forall x \in (-\infty, \infty) \quad F(x) \in \langle 0, 1\rangle$
5. $p(x_1 \leq X < x_2) = F(x_2) - F(x_1)$

Řekněme, že náhodná veličina X je *spojitá*, existuje-li integrovatelná funkce $f : \mathbf{R} \to \mathbf{R}^+$ tak, že

$$\forall x \in \mathbf{R} \quad F(x) = \int_{-\infty}^{x} f(x)dx.$$

Funkci f nazýváme *hustotou* spojité náhodné veličiny x. Z vět o integrálech vyplývá, že distribuční funkce spojité náhodné veličiny je spojitou funkcí a že pro hustotu této veličiny platí

$$\forall x \in \mathbf{R} - \mathbf{M} \quad F'(x) = f(x)$$

kde $\mathbf{M}$ je množina míry nula.

Náhodnou veličinu X, která má spočetnou množinu hodnot, nazýváme *diskrétní náhodnou veličinou*. Distribuční funkce diskrétní náhodné veličiny je schodovitou funkcí.

Vlastnosti diskrétní a spojité náhodné veličiny

Nechť X je diskrétní náhodná veličina, nabývající hodnot z konečné množiny M. Pak na množině M definujeme funkci $f(x_1)$ vztahem:

$$f(x_i) = p(x = x_i) \quad \text{pro} \quad x_i \in M$$

Tuto funkci nazýváme *frekvenční funkcí náhodné veličiny* X. Pro distribuční funkci $F(x)$ náhodné veličiny X, jež nabývá hodnot z množiny M a má frekvenční funkci $f(x_i)$, platí:

$$F(x) = \sum_{x_i \in M, x_i < X} f(x_i) \quad \text{a též} \quad \sum_{x_i \in M} f(x_i) = 1$$

U spojitých náhodných veličin platí při spojitosti distribuční funkce, že $p(X = x) = 0$.

Nechť Xje spojitá náhodná veličina s distribuční funkcí $F(x)$, jež je spojitá a po částech hladká na oboru I náhodné veličiny X, kde I je spojitá podmnožina reálných čísel. Uvažujeme-li nyní interval $\langle x, x+h)$, pak pravděpodobnost, že hodnota náhodné veličiny X padne do tohoto intervalu je

$$p(x \leq X < x + h) = F(x + h) - F(x).$$

Budeme-li nyní uvažovat délku intervalu $h \to 0$ a zároveň hodnotu pravděpodobnosti znormujeme podělením této hodnoty délkou intervalu h, pak dostáváme vztah:

$$\lim_{h \to 0} \frac{p(x \leq X < x + h)}{h} = \lim_{h \to 0} \frac{F(x + h) - F(x)}{h}$$

V případě, že distribuční funkce má v bodě x derivaci, platí

$$\lim_{h \to 0} \frac{p(x \leq X < x + h)}{h} = F'(x)$$

Má-li funkce $F(x)$ v bodě x derivaci, pak zde existují derivace zprava a zleva. Je-li $F(x)$ po částech hladká, pak její derivace neexistuje pouze ve spočetně mnoha izolovaných bodech.

Má-li distribuční funkce náhodné veličiny X derivaci na celém oboru hodnot $I \subset \mathbf{R}$, pak dostáváme funkci $f(x)$

$$f(x) = F'(x)$$

a nazýváme ji *funkcí hustoty pravděpodobnosti spojité náhodné veličiny* X.

Nechť X je spojitá náhodná veličina s distribuční funkcí $F(x)$; pak funkcí hustoty pravděpodobnosti náhodné veličiny X budeme nazývat každou funkci $f(x)$, pro níž platí tyto vztahy:

1. $f(x) \geq 0$ pro $x \in \mathbf{R}$

2. $\int_{-\infty}^{\infty} f(x)dx = 1$

3. $F(x) = \int_{-\infty}^{x} f(\xi)d\xi$ pro $x \in \mathbf{R}$

Rozložení spojité náhodné veličiny bývá nejčastěji popsáno funkcí hustoty pravděpodobnosti. Dále platí vztah:

$$p(x_1 \leq X < x_2) = \int_{x_1}^{x_2} f(\xi)d\xi$$

Charakteristiky náhodných veličin

Jak již bylo řečeno, náhodná veličina je charakterizována vymezením oboru hodnot a znalostmi pravděpodobností ve formě distribuční funkce, případně funkce hustoty pravděpodobnosti u spojitých náhodných veličin, nebo frekvenční funkce u diskrétních náhodných veličin. Tento popis je však mnohdy nepřehledný a nevhodný k praktickým aplikacím. Proto zavádíme několik číselných hodnot, které náhodnou veličinu charakterizují. Tyto hodnoty nazýváme charakteristikami náhodné veličiny. *Charakteristiky* můžeme rozdělit:

- Charakteristiky *polohy*, které mají obecně vystihnout polohu rozložení na přímce (obecně v r-rozměrném euklidovském prostoru), tj. posunutí vzhledem k počátku. Nejčastější charakteristikou je střed, dalšími charakteristikami polohy jsou modus, medián a kvantily.
- Charakteristiky *variability*, tj. rozsahu kolísání hodnot náhodné veličiny kolem středu; mezi charakteristiky variability patří rozptyl náhodné veličiny a směrodatná odchylka.
- Charakteristiky *šikmosti* obecně udávají nesymetričnost rozložení pravděpodobností nebo četností.
- Charakteristiky *špičatosti* porovnávají variabilitu rozložení ve středu a na krajích definičního oboru.

Tyto charakteristiky lze stanovit různým způsobem. Nejužívanějším a nejpropracovanějším způsobem je určení charakteristik na základě momentů. Rozlišujeme obecné momenty (ve vztahu k počátku) a centrální momenty (ve vztahu ke středu rozložení). Obecné momenty definujeme:

a) je-li X diskrétní náhodná veličina s frekvenční funkcí $f(x_i)$ a množinou hodnot M, pak obecný moment k-tého řádu je dán vztahem:

$$m_k(X) = \sum_{x_i \in M} x_i^k f(x_i) \quad \text{kde} \quad k = 1, 2, ...$$

b) je-li X spojitá náhodná veličina s funkcí hustoty $f(x_i)$, pak obecný moment k-tého řádu je definován vztahem:

$$m_k(X) = \int_{-\infty}^{\infty} x^k f(x)dx \quad \text{kde} \quad k = 1, 2, ...$$

Takto definovaných momentů nyní použijeme k formulaci základních charakteristik náhodných veličin.

Střední hodnotou náhodné veličiny X rozumíme číslo $E(X)$ definované vztahem:

a) je-li X diskrétní náhodná veličina s frekvenční funkcí $f(x)$, pak

$$E(X) = \sum_{x_i \in M} x_i f(x_i)$$

b) je-li X spojitá náhodná veličina s funkcí hustoty $f(x)$, pak

$$E(X) = \int_{-\infty}^{\infty} x f(x) dx$$

Můžeme tedy psát $E(X) = m_1(X)$. Střední hodnotu označujeme též symbolem μ.
Na základě střední hodnoty dále definujeme centrální momenty:

a) je-li X diskrétní náhodná veličina s frekvenční funkcí $f(x_i)$ a oborem hodnot M, pak centrální moment k-tého řádu je definován vztahem:

$$M_k(X) = \sum_{x_i \in M} [x_i - E(X)]^k f(x_i)$$

b) je-li X spojitá náhodná veličina s funkcí hustoty pravděpodobnosti $f(x)$, pak

$$M_k(X) = \int_{-\infty}^{\infty} [x - E(X)]^k f(x) dx.$$

Takto zadaným centrálním momentem, vztaženým ke střední hodnotě $E(X)$ náhodné veličiny X, definujeme charakteristiku variability — *rozptyl* $D(X)$. Rozptyl $D(X)$ náhodné veličiny X definujeme:

a) je-li X diskrétní náhodná veličina s frekvenční funkcí $f(x_i)$ a s oborem hodnot M, pak je rozptyl definován vztahem:

$$D(X) = \sum_{x_i \in M} [x_i - E(X)]^2 f(x_i)$$

b) je-li X spojitá náhodná veličina s funkcí hustoty $f(x)$, pak rozptyl definujeme vztahem:

$$D(X) = \int_{-\infty}^{\infty} [x - E(X)]^2 f(x) dx.$$

Využitím centrálního momentu pak dostáváme vztah $D(X) = M_2(X)$.

Další charakteristiky náhodné veličiny definujeme takto:

- *Koeficient šikmosti* (nesymetrie) náhodné veličiny X je dán vztahem:

$$\gamma_1 = \frac{M_3(X)}{[\sigma(X)]^3} \qquad \sigma(X) = \sqrt{D(X)}$$

  Jiným vyjádřením šikmosti je koeficient:

$$\beta_1 = \left[\frac{M_3(X)}{[\sigma(X)]^3}\right]^2$$

  Častěji se používá koeficientu γ_1 přičemž se značí jako β_1.

- *Koeficient špičatosti* je dán:

$$\beta_2 = \frac{M_4(X)}{[\sigma(X)]^4}$$

  Někdy se používá koeficientu:

$$\gamma = \beta_2 - 3$$

Charakteristiky rozložení pravděpodobnosti hodnot náhodných veličin tedy získáme na základě jejich obecných a centrálních momentů. Výpočet těchto momentů podle definičních vztahů však může být někdy problematický, a proto zde používáme *momentové vytvořující funkce* $m_x(z)$, která je definována vztahem $m_x(z) = E(e^{zx})$, kde z je pomocný parametr.

Pro diskrétní náhodnou veličinu X s frekvenční funkcí $f(x_i)$ a oborem hodnot M je

$$m_x(z) = \sum_{x_i \in M} e^{zx_i} f(x_i)$$

Pro spojitou náhodnou veličinu X s funkcí hustoty $f(x)$ je

$$m_X(z) = \int_{-\infty}^{\infty} e^{zx} f(x) dx$$

Postupným derivováním funkce $m_x(z) = E(e^{zx})$ a dosazením za $z = 0$ dostáváme:

$$m'_x(z) = E(Xe^{zx}), \quad m'_x(0) = E(X) = m_1(X)$$

$$m''_x(z) = E(X^2 e^{zx}), \quad m''_x(0) = E(X^2) = m_2(X)$$

Pro výpočet centrálních momentů pak s výhodou můžeme použít substituce:

$$Y = X - E(X)$$

Pak $D(X) = m''_y(0)$
Podobně též platí $D(X) = m''_x(0) - [m'_x(0)]^2$.

5.2 Generování náhodných veličin

Poněvadž jsou náhodné veličiny číselnými veličinami, mluvíme často o náhodných číslech. Náhodná čísla potřebujeme jak pro modelování spojitých, tak i diskrétních stochastických systémů; např. pro modelování fyzikálních a chemických jevů nebo systémů hromadné obsluhy, tak např. pro náhodné vybírání z mnoha variant, jako zdroj konstant pro různé algoritmy (např. hashovací funkce), nebo pro řešení některých úloh numerické matematiky metodou Monte Carlo.

Použití fyzikálních generátorů, které vytvářejí náhodná čísla na základě fyzikálních náhodných procesů v přídavném zařízení počítače, je nákladné a spojené s mnoha technickými obtížemi. Používáme proto generátorů pseudonáhodných čísel, které můžeme realizovat programově.

Posloupností pseudonáhodných čísel nazveme posloupnost $y_0, y_1, y_2, ...$ definovanou rekurentním vztahem

$$y_n = F(y_{n-1}, y_{n-2}, ..., y_{n-r})$$

která má pro $n \geq r$ statistické vlastnosti posloupnosti, získané náhodným výběrem ze souboru s rovnoměrným rozložením pravděpodobnosti. Pro generování pseudonáhodných čísel má významné postavení varianta rovnoměrného rozložení v intervalu $\langle 0, 1\rangle$ (viz. dále). Poněvadž čísla zobrazená v číslicovém počítači mají omezený počet číslic, jde o kvazirovnoměrné rozložení diskrétní náhodné veličiny. Na základě n dvojkových číslic můžeme zobrazit celkem 2^n navzájem různých čísel. Označme rovnoměrně rozloženou náhodnou veličinu ξ a kvazirovnoměrnou diskrétní veličinu ξ'. Mají-li pseudonáhodná čísla reprezentovat (v pevné řadové čárce) čísla z intervalu $\langle 0, 1\rangle$, pak dostáváme soubor $2^n - 1$ různých hodnot $\frac{i}{2^n} (i = 1, ..., 2^n - 1)$, z nichž každá se objevuje s pravděpodobností $\frac{1}{2^n - 1}$.

Vyjádříme vztahy mezi středními hodnotami s rozptyly rovnoměrného a kvazirovnoměrného rozložení pravděpodobnosti:

$$E(\xi') = \frac{1}{2^n - 1} \sum_{i=1}^{2^n - 1} \frac{i}{2^n} = \frac{1}{2} = E(\xi)$$

$$D(\xi') = \frac{1}{2^n - 1} \sum_{i=1}^{2^n - 1} (\frac{i}{2^n} - \frac{1}{2})^2 = \frac{1}{2^n - 1} \sum_{i=1}^{2^n - 1} \frac{i^2}{2^{2n}} - \frac{1}{2^n - 1} \sum_{i=1}^{2^n - 1} \frac{i}{2^n} + \frac{1}{4}$$

Po vyčíslení sumy kvadrátů a dosazení střední hodnoty získáme vztah:

$$D(\xi') = \frac{1}{12}(1 - \frac{1}{2^n - 1}) = D(\xi)(1 - \frac{1}{2^n - 1})$$

Při větším počtu dvojkových řádů můžeme teoretickou nepřesnost rozptylu zanedbat, a proto lze uvažovat o rovnoměrném rozložení, i když jde ve skutečnosti o rozložení kvazirovnoměrné.

Na posloupnost pseudonáhodných čísel klademe tyto základní požadavky: rovnoměrné rozložení, statistickou nezávislost, reprodukovatelnost, rychlé generování, minimální obsazení paměti počítače příslušným programem.

Zvolíme-li vhodnou metodu generování pseudonáhodných čísel, můžeme splnit všechny uvedené požadavky kromě neopakovatelnosti pseudonáhodných čísel. Každá posloupnost pseudonáhodných čísel má vždy určitou konečnou periodu a naší snahou je, aby tato perioda byla co nejdelší. Velkou výhodou tohoto způsobu je možnost reprodukování posloupnosti náhodných čísel, poněvadž je vytváříme podle jednoznačných aritmetických operací.

Metody vytváření pseudonáhodných čísel

Z mnoha metod, které byly pro generování pseudonáhodných čísel vytvořeny, vybereme pouze některé jednoduché metody pro ilustraci.

a) Zvolíme s-místné číslo; povýšíme je na druhou a vypíšeme s prostředních číslic. S takto nalezeným s-místným číslem proces opakujeme. Je-li s dostatečně velké, např. $s \geq 10$, můžeme takto nalezené číslice považovat za číslice s-místných pseudonáhodných čísel z intervalu $\langle 0, 1 \rangle$.

b) Číslo s dostatečně velkým počtem číslic násobíme konstantním součinitelem a vypíšeme s prostředních číslic výsledku atd.

c) Číslo s velkým počtem číslic povýšíme na druhou a za náhodné číslo volíme zbytek po dělení této druhé mocniny dostatečně velkým prvočíslem.

Dále můžeme vytvářet pseudonáhodná čísla různými přemisťovacími metodami, které využívají speciálních instrukcí počítačů jako jsou posuny, binární součty bez přenosu a podobně. Posloupnosti pseudonáhodných čísel lze také různě zlepšovat. Někdy se používá tohoto postupu: v paměti počítače vyhradíme určitý počet buněk (asi 100), do kterých uložíme pseudonáhodná čísla vytvořená libovolnou vhodnou metodou. Hodnotu náhodného čísla náhodně vybereme z tohoto seznamu a prázdné paměťové místo zaplníme nově vygenerovanou hodnotou.

Statistické vlastnosti posloupnosti náhodných čísel se zpravidla zlepší, použijeme-li více nezávislých zdrojů náhodných čísel, které deterministicky nebo náhodně střídáme.

Kongruentní metoda generování pseudonáhodných čísel. Popis kongruentního generátoru a návrh jeho konstant.

V lineárním kongruentním generátoru se posloupnost pseudonáhodných čísel získává na základě vztahu:

$$X_{n+1} = (aX_n + c) mod\ m \qquad [RAND]$$

kde a, X_0, c, m jsou přirozená čísla. Nazýváme je takto:

X_0 ... počáteční hodnota
a ... multiplikativní konstanta
c ... aditivní konstanta
m ... modul

Tato čísla nejsou samozřejmě libovolná. Na jejich výběru závisí perioda i rozložení posloupnosti generovaných pseudonáhodných čísel.

Připomeňme, že výraz $p \equiv q(mod\ m)$ čteme: p je kongruentní s q podle modulu m. Platí, že $p \equiv q(mod\ m)$, když $p - q$ je dělitelné m. To znamená, že zbytek po dělení čísla p číslem m je roven zbytku po dělení čísla q číslem m. Můžeme tedy psát:

$$p = p_1 m + z$$

$$q = q_1 m + z$$

Jedno z čísel p, q ve výrazu $p \equiv q(mod\ m)$ je zpravidla přímo zbytkem po dělení modulem m. Pro výběr konstant a, X_o, c, m platí pravidla:

a) **Výběr modulu** m

Při výběru modulu $m$se snažíme splnit tyto požadavky:

1. co nejdelší periodu posloupnosti
2. co největší rychlost generování pseudonáhodných čísel.

Požadavku ad 1) můžeme vyhovět výběrem co největšího modulu m. Položíme tedy m rovno maximálnímu číslu typu *integer* zobrazitelnému v jednom slově počítače (používáme pro ně označení *maxint*). Na 32 bitovém počítači se maxint rovná číslu 2147483647. Nyní se ještě pokusíme vyhovět požadavku ad 2):

Operaci "modulo m" bychom mohli realizovat celočíselným dělením, ale bylo by to příliš náročné na čas, protože dělení trvá na počítači poměrně dlouho.

Zvolíme-li však $m = maxint$, pak se dá výraz $(aX_n + c)mod\ m$ nahradit výrazem $(aX_n+c)+m+1$. Vyčíslování tohoto výrazu bude podstatně rychlejší, jeho nevýhodou však je, že jej nelze naprogramovat v některém z vyšších jazyků, jejichž překladače by hlásily přetečení, které zde nutně musí vzniknout.

b) **Výběr konstant** a **a** c

Předpokládáme, že modul m je již zvolen podle bodu a). Konstanty a a c pak vybíráme tak, aby perioda byla co největší, tj. právě rovna modulu m. Tento požadavek lze splnit za podmínek:

1. c a m jsou navzájem nesoudělná čísla
2. $b = a - 1$ je dělitelné číslem p pro libovolné p, jež je dělitelem čísla m
3. b je dělitelné čtyřmi, jestliže m je dělitelné čtyřmi.

V zájmu urychlení generování lze volit $c = 0$, takovou volbou se však zpravidla zkrátí perioda. Nicméně předpokládáme, že i pro $c = 0$ je možno dosáhnout vyhovujících výsledků a pro generátor, který jsme zkoumali, vždy platí, že $c = 0$.

Je-li modul m prvočíslo, pak podmínkám ad 1) až ad 3) vyhovuje např. multiplikativní konstanta ve tvaru

$$a = zk + 1,$$

kde $2 \leq k < e$ a z je základ číselného zobrazení v počítači (v našem případě je to 2) a z^e je pak největší číslo typu *integer*, zobrazitelné v počítači, zvětšené o 1. V našem případě $z^e = 2^31$. Multiplikativní konstanta tohoto typu má však tu nepříjemnou vlastnost, že vede na nedostatečně náhodná čísla. Proto pro správný výběr multiplikativní konstanty zavádíme *mohutnost lineární kongruentní posloupnosti* M. Mohutnost se určuje u posloupnosti s maximální periodou jako nejmenší číslo Mtakové, že platí:

$b^M \equiv O(mod\ m)$, kde M musí být celé číslo.

(Např. pro $m = 2^35$ a $a = 2^k + 1$ je $b = 2^k$. Potom pro $k = 18$ je výraz $b^2 = 2^{2k}$ dělitelný číslem m, mohutnost je tedy rovna 2. pro $k = 17, 16, 15, ..., 12$ je mohutnost rovna 3.)

Z hlediska mohutnosti je nejvýhodnější k malé, ale to vede také na malou délku periody, což je nevýhodné. Proto volíme multiplikativní konstantu ve tvaru $a = 2^k + 2^i + 2^j + 1$ (např. ve tvaru $2^23+2^14+2^2+1$), což nám umožní získat generátor s velkou mohutností i velkou hodnotou multiplikativní konstanty.

Návrh konstanty a se obvykle provádí spektrálním testem, jenž je popsán v literatuře. Osvědčená je multiplikativní konstanta získaná z firemní literatury IBM: $a = 1220703125$.

Podle [36] lze vhodně zvolit multiplikativní konstantu:

$$a = 8i \pm 3 \quad \text{při} \quad c = 0$$

kde i je libovolné celé číslo. Je vhodné vybrat a blízké k $2^{d/2}$, kde d je počet bitů.

c) **Výběr počáteční hodnoty X_o**

Výběr počáteční hodnoty X_o je naším hlavním úkolem. Protože však právě pro tento výběr známe nejméně pravidel, musíme jej provádět téměř empiricky. X_o by mělo být dostatečně velké liché číslo, nemělo by být soudělné s modulem m.

Příklad generátoru rovnoměrného rozložení $R(0,1)$ v jazyce PASCAL

Generátor s názvem `Random`, který se vyvolává jako funkce a generuje čísla typu `real` z intervalu $\langle 0, 1)$.

```
function Random: real;
begin
   ix := ix * 1220703125;
   if ix < 0 then
      ix := ix + maxint + 1;
   random := ix / maxint
end;
```

Jde o kongruentní generátor (s modulem `maxint` a multiplikativní konstantou 1 220 703 125) s následnou transformací na interval $\langle 0, 1)$. Využívá jazyka PASCAL-EC, který v tomto případě po násobení při přetečení nehlásí chybu. O proměnné `ix` typu `integer` se předpokládá, že je deklarovaná v nadřazeném bloku a že je jí přiřazena nějaká počáteční hodnota před prvním použitím funkce `Random`, např. `ix := 1537`.

5.3 Transformace rovnoměrného rozložení na požadovaný typ rozložení

Při modelování stochastických procesů často potřebujeme posloupnosti náhodných čísel s jiným, než rovnoměrným rozložením pravděpodobnosti. Tyto posloupnosti lze získat transformací rovnoměrného rozložení na požadovaný průběh. Základem je tedy kvalitní generátor, produkující rovnoměrně rozložená náhodná čísla z intervalu $\langle 0, 1)$. Pro vlastní provádění transformace existuje několik metod, kterých využíváme podle toho, o jaké rozložení jde a současně musíme vzít v úvahu přesnost a efektivnost celého procesu. Dříve, než určité metody použijeme pro konkrétní výpočet, musíme provést testování získaného průběhu (viz dále) a zhodnocení, zda je vytvořené rozložení pro řešenou úlohu dostatečně přesné; jednotlivé typy úloh mohou klást na posloupnost náhodných čísel různé požadavky. Při výběru vhodné metody transformace musíme zpravidla volit kompromis mezi její efektivností, požadavky na paměť a dosahovanými výsledky. Nejčastěji používanými metodami pro transformování náhodných čísel na zadané rozložení pravděpodobnosti jsou:

- metoda inverzní transformace

- metoda vylučovací
- metoda kompoziční
- jiná vhodná metoda, využívající např. definice rozložení, aproximace některé z jejích charakteristických funkcí apod.

Metoda inverzní transformace

Nechť náhodná veličina R má rovnoměrné rozložení $R(0,1)$ a F je funkce, splňující podmínky pro distribuční funkci (viz začátek kapitoly 5). Potom platí:

Náhodná veličina $F^{-1} \bullet R$ má rozložení s distribuční funkcí F, neboť

$$p((F^{-1} \bullet R) < x) = p(F \bullet (F^{-1} \bullet R) < F(x)) = p(R < F(x)) = F(x)$$

Takto můžeme přesně generovat náhodná čísla s daným rozložením. V případě, že distribuční funkce daného rozložení nemá inverzní funkci (např. když distribuční funkci nelze vyjádřit elementárními funkcemi), pak při nevhodnosti jiných metod můžeme použít aproximace této funkce jinou vhodnou funkcí, jejíž inverzní funkce je známá. Taková transformace již patří k přibližným metodám.

Metoda vylučovací

Nechť $f : (x_1, x_2) \to (0, M)$ je požadovaná funkce hustoty generované náhodné veličiny,
$X : \Omega \to (x_1, x_2)$ je náhodná veličina s rozložením $R(x_1, x_2)$
$Y : \Omega \to (0, M)$ je náhodná veličina s rozložením $R(0, M)$
Potom pro každé $t \in (x_1, x_2)$ platí:

$$p(X < t | Y < f(X)) = \int_{x_1}^{t} f(x)dx$$

za předpokladu, že náhodné veličiny X, Y jsou nezávislé.

Důkaz: Označme $\vec{Z} := (X, Y)$. Lehce lze dokázat, že pro rozložení náhodného vektoru $\vec{Z}$ platí

$$\forall E \subset (x_1, x_2) \times (0, M) \quad p_{\vec{Z}}(E) := p(\vec{Z} \in E) = \frac{1}{M(x_2 - x_1)} m_2(E)$$

(m_2 je borelovská míra na $\mathcal{R}_{\mathbf{R}^2}$).

Označme dále

$$E := \{[x, y] | x_1 < x < t\} \quad S := \{[x, y] | 0 < y < f(x)\}, \quad t \in \mathbf{R}$$

Potom můžeme psát

$$p(X < t | Y < f(X)) = p_{\vec{Z}}(E|S) = \frac{p_{\vec{Z}}(E \cap S)}{p_{\vec{Z}}(S)} = \frac{m_2(E \cap S)}{m_2(S)} = \int_{x_1}^{t} f(x)dx,$$

přičemž poslední rovnost plyne z toho, že

$$m_2(S) = \int_{x_1}^{x_2} f(x)dx = 1$$

$$m_2(E \subset S) = \int_{x_1}^{t} f(x)dx.$$

Důsledek: Náhodnou veličinu ξ s funkcí hustoty $f : (x_1, x_2) \to (0, M)$ budeme generovat takto:

1. Generujeme číslo x z rozložení $R(x_1, x_2)$.
2. Generujeme číslo y z rozložení $R(O, M)$.
3. Jestliže $y < f(x)$, pak x prohlásíme za hodnotu náhodné veličiny ξ, jinak opakujeme celý postup znovu.

Efektivnost této metody úzce souvisí s poměrem plochy S a plochy obdélníka $(x_2 - x_1)$. M, kde plocha S je plocha ohraničená osou x a funkcí hustoty $f(x)$. Jelikož většina funkcí hustoty, popisujících dané rozložení, nemá zprava, zleva nebo zprava i zleva ohraničený definiční obor, musíme provést jisté zanedbání a udělat určitou restrikci funkce hustoty na ohraničený interval. Efektivnost metody totiž se zmenšováním plochy S k ploše obdélníka klesá, proto zde volíme kompromis mezi časovou náročností transformace a šířkou intervalu (x_1, x_2).

5.3.1 Metoda kompoziční

Nechť $V : \omega \to \{1, 2, ..., n\}$ je náhodná veličina s frekvenční funkcí

$$h : \{1, 2, ..., n\} \to \langle 0, 1\rangle, \quad h_i = p(V = i),$$

takže platí

$$(1) \qquad \sum_{i=1}^{n} h_i = 1;$$

dále nechť $X_1, X_2, ..., X_n : \Omega\mathbf{R}$ jsou náhodné veličiny s funkcemi hustoty $g_1, g_2, ..., g_n$, takže platí

$$(2) \qquad \int_{-\infty}^{\infty} g_i(x)dx = 1$$

pro $i = 1, 2, ..., n$. Potom náhodná veličina Y definovaná

$$\forall\omega \in \Omega \quad Y(\omega) := \begin{cases} X_1(\omega) & \text{jestliže} \quad V(\omega) = 1 \\ X_2(\omega) & \qquad\quad V(\omega) = 2 \\ \vdots & \\ X_n(\omega) & \qquad\quad V(\omega) = n \end{cases}$$

má rozložení s funkcí hustoty pravděpodobnosti

$$(3) \qquad f(x) = \sum_{i=1}^{n} h_i g_i(x)$$

Důkaz: Podle věty o úplné pravděpodobnosti platí pro každé $t \in \mathbf{R}$:

$$p(Y < t) = p(Y < t|V = 1)p(V = 1) + ... + p(Y < t|V = n)p(V = n) =$$

$$= p(X_1 < t)p(V = 1) + ... + p(X_n < t)p(V = n) =$$

$$= \sum_{i=1}^{n} h_i \int_{-\infty}^{t} g_i(x)dx = \int_{-\infty}^{t} (\sum_{i=1}^{n} h_i g_i(x))dx.$$

Důsledek: Jestliže máme za úkol generovat čísla z rozložení daného funkcí hustoty

$$f = \sum_{i=1}^{n} f_i$$

kde $\forall i \in \{1, 2, ..., n\}$ f_i je nezáporná integrovatelná funkce, budeme postupovat takto:

1. Funkci f upravíme na tvar (3), aby platily podmínky (1),(2):
   pro $i = 1, 2, ..., n$ položíme

$$h_i := \int_{-\infty}^{\infty} f_i(x)dx$$

$$g_i := \frac{1}{h_i} f_i.$$

   (ověřte si, zda nyní skutečně platí uvedené podmínky)

2. Generujeme $v \in \{1, 2, ..., n\}$ z rozložení daného frekvenční funkcí h.

3. Generujeme číslo y z rozložení daného funkcí hustoty g_v. Číslo y je pak hledaná hodnota náhodné veličiny.

Praktické použití: Kompoziční metoda umožňuje rozložit složitou funkci hustoty pravděpodobnosti na řadu jednodušších funkcí, které usnadní generování. Důvodem k použití této metody může být také snaha o dosažení vyšší efektivnosti při použití vylučovací metody. Tato situace je znázorněna na obrázku.

Při použití vylučovací metody na intervalu $\langle x_1, x_2' \rangle$ by účinnost metody byla malá vzhledem k velké ploše obdélníka $\langle 0, M \rangle \times \langle x_1, x_2' \rangle$. Rozložením funkce hustoty

$$f(x) = f(x)\chi_{<\xi_1,\xi_2>}(x) + f(x)\chi_{<\xi_1',\xi_2'>}(x)$$

při použití kompoziční metody dosáhneme mnohem lepší účinnosti.

Jiné metody

Někdy využíváme definice náhodné veličiny s daným rozložením, případně jejího zjednodušeného popisu. Proto často dochází ke vzniku dvou- a vícenásobných transformací, kdy zadanou náhodnou veličinu získáme na základě rozložení, jež je také výsledkem transformace. Takové postupy volíme pouze při nevhodnosti ostatních metod, protože tyto transformace již bývají časově i realizačně náročné. Do této skupiny také spadají metody aproximační,

kde některou z charakteristických funkcí rozložení (distribuční funkci nebo funkci hustoty) aproximujeme vhodnější funkcí za určených dovolených chyb a pak realizujeme některou z uvedených metod. Tyto transformační metody bývají poměrně jednoduché a jejich přesnost je postačující. Přesto je třeba provádět ověřování transformovaných náhodných veličin, které mohou chybu základního rozložení, z něhož byly získány, zvětšovat až několikanásobně.

U diskrétních náhodných veličin uvažujeme za základ transformačních vztahů přímo definici rozložení, nebo využíváme upravené metody inverzní transformace. Další možností může být aproximace diskrétního rozložení některým spojitým rozložením.

5.4 Nejužívanější rozložení pravděpodobností náhodných veličin

V této části popíšeme často používaná rozložení pravděpodobností spojitých a diskrétních náhodných veličin. V popisu uvedeme:

- výskyt veličin s tímto rozložením, jeho definici a použití,
- základní funkce popisující vlastnosti náhodné veličiny,
- hlavní charakteristiky rozložení (střední hodnotu, rozptyl apod.),
- způsob generování pseudonáhodných čísel s tímto rozložením.

K zápisu nejpoužívanějších transformačních vztahů využijeme programovacího jazyka PASCAL.

5.4.1 Spojité náhodné veličiny

Rovnoměrné rozložení

Toto rozložení značíme $R(a, b)$, kde a a b jsou jeho parametry. Představují dolní a horní hranici oboru hodnot náhodné veličiny s tímto rozložením. Náhodná veličina nabývá hodnot rovnoměrně z intervalu $\langle a, b\rangle$, tzn. že funkce hustoty pravděpodobnosti musí být na tomto intervalu konstantní nenulová a jinde nulová:

$$f(x) = \begin{cases} 0 & x < a \\ \frac{1}{b-a} & a \leq x \leq b \\ 0 & x > b \end{cases}$$

Integrací této funkce získáme distribuční funkci rozložení:

$$F(x) = \begin{cases} 0 & x < a \\ \frac{x-a}{b-a} & a \leq x \leq b \\ 0 & x > b \end{cases}$$

Normováním získáme z obecného rovnoměrného rozložení náhodnou veličinu X s rozložením $R(0, 1)$, kdy X nabývá hodnot od 0 do 1:

$$\forall \omega \in \Omega \quad X(\omega) \in \langle 0, 1\rangle.$$

V tomto případě má funkce hustoty tvar

$$f(x) = \begin{cases} 1 & \text{pro} \quad 0 \leq x \leq 1 \\ 0 & \text{jinde} \end{cases}$$

a distribuční funkce má tvar

$$F(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$$

Toto rozložení slouží většinou jako základ pro transformace na ostatní typy rozložení, a proto je získáváme některou z metod přímého generování pseudonáhodných čísel, např. metodou kongruentní.

Charakteristiky rozložení:

Střední hodnota: $E(x) = \frac{a+b}{2}$
Rozptyl: $D(x) = \frac{(b-a)^2}{12}$

Transformační vztahy pro obecné rovnoměrné rozložení:

S výhodou použijeme metody inverzní transformace

$$F(X) = \frac{X-a}{b-a} \Rightarrow X = F(X)(b-a) + a$$

Dosadíme-li do tohoto vztahu za symbol $F(X)$ náhodnou veličinu s rozložením $R(0,1)$, kterou považujeme za výchozí, pak náhodná veličina X bude mít rozložení $R(a,b)$.

Zápis ve formě pascalovské procedury:
(identifikátorem `Random` je označena náhodná veličina s $R(0,1)$.

```
function ROVREAL (A,B: real): real;
var X: real;
begin
   X := Random;
   ROVREAL := (B - A) * X + A
end;
```

V praxi nastane často případ, kdy potřebujeme náhodnou veličinu, jež má rovnoměrné rozložení, ale nabývá pouze hodnot z množiny celých čísel. Tato veličina sice spadá pod diskrétní náhodné veličiny, přesto ji uvedeme v tomto odstavci.

Jestliže tato náhodná veličina X nabývá hodnot z množiny $M = \{a, a+1, a+2, ..., b\}$, kde může nabýt podle pravidla rovnoměrnosti kterékoliv hodnoty z množiny M se stejnou pravděpodobností $p = \frac{1}{n}$, kde $n = card\ M$, pak můžeme při generování použít rovnoměrného rozložení $R(0,1)$. Jsou-li parametry rozložení celá čísla a, b, pak využitím transformačního vztahu pro rovnoměrné rozložení v oboru reálných čísel a operace zaokrouhlení (označme $ROUND$) můžeme vytvořit vztah

$$X = ROUND((b-a+1)x + (a-0.5)),$$

kde x je náhodná veličina s rozložením $R(0,1)$.

Zápis v Pascalu:

```
function ROVINTG (A,B: integer): integer;
var X: real;
begin
   X := Random;
   ROVINTG: = ROUND ((B-A+1) * X + (A-0.5))
end;
```

Efektivnost těchto metod je maximální - transformace je jednoduchá, rychlá a spolehlivá. Je samozřejmé, že chyby rozložení $R(0,1)$ se tu projeví v nezmenšené míře.

Exponenciální rozložení

Exponenciální rozložení je časté rozložení dob obsluhy, časových intervalů mezi poruchami a mezi příchody do front v procesech hromadné obsluhy. Je to nesymetrické rozložení.
Funkce hustoty rozložení:

$$f(x) = \frac{1}{A} e^{-\frac{1}{A}(x-x_0)} \quad \text{pro} \quad x \geq x_0$$

Distribuční funkce

$$F(x) = \begin{cases} 1 - e^{-\frac{1}{A}(x-x_0)} & \text{pro} \quad x \geq x_0 \\ 0 & \text{pro} \quad x < x_0 \end{cases}$$

kde A a x_0 jsou parametry.

Charakteristiky rozložení:

Střední hodnota:	$E(X) = x_0 + A$
Rozptyl:	$D(X) = A^2$.

U tohoto rozložení často pokládáme $x_0 = 0$.

Pro generování čísel s exponenciálním rozložením můžeme použít metody inverzní transformace, na jejímž základě získáme vztah

$X = x_0 + (-A \ln x)$, kde x má rozložení $R(0, 1)$.

Všimněme si, že ve vztahu se vyskytuje člen $\ln(x)$ místo členu $\ln(1 - x)$, který by odpovídal inverzi distribuční funkce F. Dá se lehce dokázat, že za předpokladu, že náhodná veličina R má rozložení $R(0, 1)$, má náhodná veličina $1 - R$ také rozložení $R(0, 1)$. Použitím členu $\ln(x)$ tedy dosáhneme stejného účinku jako použitím členu $\ln(1 - x)$, ale ušetříme jednu aritmetickou operaci.

Zápis s využitím procedury `Random`:

```
function Exponential (A,X0 : real) : real;
var X : real;
begin
   X := Random;  // problém: 0  (asi je lepší 1-Random)
   Exponential := X0 - A * ln(X)
end;
```

Tato transformace je velmi rychlá a efektivní, chyba rozložení je opět způsobena pouze chybou transformovaného rozložení $R(0, 1)$

Normální rozložení

Normální (někdy též Gaussovo) rozložení je nejdůležitějším rozložením spojité náhodné veličiny. Je použitelné všude, kde jsou změny náhodné veličiny způsobeny současným vlivem velkého počtu vzájemně nezávislých faktorů, které se uplatňují přibližně stejnou měrou. Za určitých podmínek se jím dají modelovat i rozložení diskrétních náhodných veličin.

Toto rozložení má funkci hustoty

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

kde σ a μ jsou reálné parametry rozložení; $\sigma > 0$; $x \in (-\infty, \infty)$.
Na základě transformace proměnné x

$$y = \frac{x - \mu}{\sigma}$$

dostáváme normované normální rozdělení s parametry $\mu = 0$ a $\sigma = 1$, jež má funkci hustoty

$$\varphi(y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}}$$

Distribuční funkce normálního rozložení má tvar:

$$F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(\xi-\mu)^2}{2\sigma^2}} d\xi$$

Tento integrál nelze vyjádřit elementárními funkcemi, a proto distribuční funkcí uvádíme ve tvaru (obdobném i pro normované rozložení)

$$\Phi(y) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y} e^{-\frac{\psi^2}{2}} d\psi$$

Platí, že

$$F(x) = \Phi(\frac{x - \mu}{\sigma}).$$

Hodnoty funkce Φ bývají ve statistických tabulkách tabelovány.

Charakteristiky rozložení:

Střední hodnota: $E(X) = \mu$
Střední hodnota pro normovanou veličinu: $E(X) = 0$
Rozptyl: $D(X) = \sigma^2$
Rozptyl pro normovanou veličinu: $D(X) = 1$
Normovaná šikmost: $\beta_1 = 0$
Normovaná špičatost: $\beta_2 = 3$

Generování náhodné veličiny s normálním rozložením:

Při generování vycházíme ze skutečnosti, že pro hodnoty náhodné veličiny X s parametry μ a σ s normálním rozložením platí podle zpětného transformačního vztahu $X = Y\sigma + \mu$, kde Y je náhodná veličina s normovaným normálním rozložením. Proto získáváme hodnoty náhodné veličiny s normovaným rozložením, které podle tohoto vztahu dále transformujeme.

Způsobů získání náhodné veličiny s parametry $\mu = 0$ a $\sigma = 1$ je několik. Nemůžeme využít metody inverzní transformace, neboť nelze získat funkci inverzní k distribuční funkci. Vhodnější je metoda vylučovací, jež využívá funkce hustoty náhodné veličiny. Zde je třeba stanovit hranice oboru hodnot tak, abychom při co nejmenších chybách dosáhli co největší efektivnosti. Vzhledem k nízkým hodnotám funkce hustoty $\phi(x)$ pro $x < -5$ a $x > 5$ můžeme pro série čísel do počtu kolem 10^6 považovat za obor hodnot náhodné veličiny interval

$$X \in (\mu - 5\sigma, \mu + 5\sigma).$$

Pro malé série čísel dokonce stačí omezení

$$X \in (\mu - 3\sigma, \mu + 3\sigma).$$

která u normované veličiny představuje volbu

$$X \in (-5, 5) \quad \text{resp.} \quad X \in (-3, 3).$$

Maximum funkce hustoty normované veličiny pak je pro $x = 0$

$$f(0) = \frac{1}{\sqrt{2\pi}} = M$$

Na základě tohoto algoritmu pak lze vytvořit transformační proceduru.

Další metodou je metoda sečítání nezávislých náhodných veličin, která využívá centrální limitní věty, podle níž rozdělení střední hodnoty n nezávislých náhodných veličin, rozložených podle libovolného zákona případně různých zákonů, se při zvětšování n blíží k normálnímu rozložení. Charakteristiky náhodných veličin musí být konečné. Jako nezávislé náhodné veličiny bereme hodnoty veličiny s rozložením $R(0, 1)$. Pak při n hodnotách této veličiny dostáváme hodnotu náhodné veličiny s normovaným normálním rozložením podle vztahu

$$X = \frac{\sum^n R - \frac{n}{2}}{\sqrt{\frac{n}{12}}}.$$

Jedná se o vícekrokovou metodu, kdy k získání jedné hodnoty s normálním rozložením potřebujeme více hodnot náhodné veličiny $R(0, 1)$. S rostoucím n se zvyšuje přesnost metody, klesá však rychlost a tím i celková efektivnost této transformace. Musíme tedy volit vhodný kompromis — např. $n = 12$

$$X = \sum^1 2R - 6$$

a pro náhodnou veličinu s obecně normálním rozložením s parametry μ a σ

$$X = (\sum^1 2R - 6)\sigma + \mu.$$

V obou vztazích má veličina R rozložení $R(0, 1)$.

Zápis v Pascalu:

```
function Normal1(EX,STX: real): real;
var X, SUM : real;
I : integer;
begin
   SUM := 0;
   for I:=1 to 12 do
   begin
      X := Random;
      SUM := SUM + X;
   end;
   Normal1 := STX * (SUM - 6) + EX
end;
```

Dalším zpřesněním metody by mohlo být použití $n = 24$ apod. Tato transformace již přináší vlastní chybu, neboť vznikla na základě nahrazení nekonečného počtu náhodných veličin počtem konečným. Přesto je v této podobě relativně přesná a rychlá. Pro správnou funkci generátoru je třeba zajistit nezávislost rovnoměrně rozložených hodnot.

Další metodou je přímá transformace rovnoměrně rozložených čísel. Určitým matematickým postupem lze ukázat, že jsou-li náhodná čísla X_1, X_2 rovnoměrně rozložená $R(0,1)$, pak čísla Y_1 a Y_2 definovaná vztahy

$$\begin{aligned} Y_1 &= \sqrt{-2\ln X_1}\cos(2\pi X_2) \\ Y_2 &= \sqrt{-2\ln X_1}\sin(2\pi X_2) \end{aligned}$$

mají normované normální rozložení.

K získání dvojice normálně rozložených čísel tedy potřebujeme dvojici čísel s rozložením $R(0,1)$.

Zápis transformace v Pascalu:

```
function Normal2(EX,STX: real): real;
const PI = 3.1415926;
var X,Y: real;
begin
   X := Random;
   Y := Random;
   NORMAL2 := STX * sqrt((-2(*LN(X)) * cos(2*PI*Y) + EX
end;
```

V tomto zápisu jsou odpovídající symboly `EX` $= \mu$; `STX` $= \sigma$.

Přesnost metody je podle testů rovnoměrnosti vyšší, než u předcházející transformace, časová náročnost není větší.

Od normálního rozložení je odvozena řada náhodných veličin. Jde např. o *polonormální rozložení*, jež udává absolutní hodnotu náhodné veličiny s normálním rozložením s parametry $\mu = 0$ a σ. Na základě symetričnosti máme funkci hustoty ve tvaru

$$f(x) = \frac{1}{\sigma}\sqrt{\frac{2}{\pi}}e^{-\frac{x^2}{2\sigma^2}} \quad \sigma > 0 \quad x \geq 0$$

a charakteristiky:

Střední hodnotu: $E(X) = \sigma\sqrt{\frac{2}{\pi}}$
Rozptyl: $D(X) = \sigma^2(1 - \frac{2}{\pi})$
Normovanou šikmost: $\beta_1 = 0.995$
Špičatost: $\beta_2 = 3.869$

Toto číslo je větší než nula, což znamená, že rozložení je pravostranně asymetrické.
Pro generování hodnot náhodné veličiny s polonormálním rozložením použijeme transformace pro normální rozložení a výsledek bereme v absolutní hodnotě.

Další je *logaritmicko-normální rozložení* s parametry μ a σ s funkcí hustoty pravděpodobnosti

$$f(x) = \frac{1}{\sigma x\sqrt{2\pi}}e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$$

Je to opět nesymetrické rozložení s charakteristikami:

Střední hodnotou: $E(X) = e^{(\mu+\frac{\sigma^2}{2})}$

Rozptylem: $D(X) = e^{(2\mu+\sigma^2)(e^{\sigma^2}-1)}$

Toto rozložení mívají např. náhodné veličiny, jejichž hodnoty jsou dány působením mnoha účinků s velmi slabým vlivem. Transformační vztah vychází z transformace pro normální rozložení

$$Y_1 = \exp[\mu + \sqrt{-2\sigma^2 \ln X_1}\cos(2\pi X_2)]$$

$$Y_2 = \exp[\mu + \sqrt{-2\sigma^2 \ln X_1}\sin(2\pi X_2)]$$

kde x_1 a x_2 jsou hodnoty náhodné veličiny s rozložením $R(0,1)$.

Pearsonovo rozložení χ^2

Rozložení χ^2 je definováno na základě normálního rozložení s parametry $\mu = 0$ a $\sigma = 1$. Máme-li n náhodných veličin s tímto rozložením $X_1, X_2, ..., X_n$, pak náhodná veličina χ^2 definovaná vztahem

$$\chi^2 = \sum_{i=1}^{n} X_i^2$$

má Pearsonovo rozložení. Parametrem rozložení je počet stupňů volnosti, jenž je tvořen počtem sčítanců v sumě v definičním vztahu.

Funkce hustoty pravděpodobnosti:

$$f(x) = \frac{1}{2^{\frac{n}{2}}\Gamma(\frac{n}{2})} e^{-\frac{x}{2}} x^{(\frac{n}{2}-1)} \quad \text{pro} \quad x \geq 0$$

Charakteristiky rozložení:

Střední hodnota: $E(\chi^2) = n \quad D(\chi^2) = 2n.$
Rozptyl: $\beta_1 = 2\sqrt{\frac{2}{n}} \quad \beta_2 = 3 + \frac{12}{n}.$

Protože $\beta_1 > 0$, má rozložení pravostrannou asymetrii. Toto rozložení je důležité zejména proto, že je základem pro testování statistických hypotéz Pearsonovým testem dobré shody. Vztahy pro transformaci můžeme nalézt např. ve tvaru

$\chi^2 = -2 \ln \prod_{i=1}^{\frac{n}{2}} X_i$, kde X_i má rozložení $R(0,1)$.

Tento postup je jednodušší, než využití definičního vztahu. Pro $n > 30$ liché můžeme lépe využít aproximace

$\chi^2 = \frac{1}{2}(X + \sqrt{2n-1})^2$, kde X má normované normální rozložení.

Poznámka: V literatuře se symbolu χ^2 užívá v těchto významech:

a) jako označení pro Pearsonovo rozložení

b) jako symbolu, označujícího náhodnou veličinu s Pearsonovým rozložením

c) jako symbolu, označujícího proměnnou při popisu funkcí, charakterizujících náhodnou veličinu s Pearsonovým rozložením (Např. funkce hustoty).

Tak např. můžeme psát, že funkce hustoty pravděpodobnosti náhodné veličiny χ^2 s rozložením $\chi^2(n)$ má tvar:

$$f(\chi^2) = \frac{1}{2^{\frac{n}{2}}\Gamma(\frac{n}{2})} e^{-\frac{\chi^2}{2}} (\chi^2)^{(\frac{n}{2}-1)}, \quad \chi^2 > 0$$

Stejný formalismus se používá u rozložení t a F.

Studentovo rozložení t

Toto rozložení má rovněž velký význam při testování statistických hypotéz. Je definováno normovaným normálním rozložením X a rozložením χ^2 s n stupni volnosti:

$$t = \frac{x}{\sqrt{\frac{\chi^2}{n}}}.$$

Funkce hustoty tohoto rozložení má tvar

$$f(t) = \frac{1}{\sqrt{n}} \frac{1}{B(\frac{1}{2}, \frac{n}{2})} (1 + \frac{t^2}{n})^{-\frac{n+1}{2}}$$

kde symbol B představuje funkci beta. Na základě funkce Γ můžeme $f(t)$ zapsat ve tvaru:

$$f(t) = \frac{1}{\sqrt{\pi n}} \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} (1 + \frac{t^2}{n})^{-\frac{n+1}{2}}$$

Charakteristiky rozložení:

Střední hodnota: $E(t) = 0$
Rozptyl: $D(t) = \frac{n}{n-2}$
Normovaná šikmost: $\beta_1 = 0$, což značí, že rozložení je souměrné
Špičatost: $\beta_2 = 3\frac{n-2}{n-4}$

Pro transformaci využijeme v tomto případě definičního vztahu. Tím musíme vycházet z transformačních vztahů pro normální a Pearsonovo rozdělení.

Fisher Snedecorovo rozdělení (F - rozdělení)

Tohoto rozložení se také často využívá k testování statistických hypotéz. Je definováno dvěma veličinami s rozložením χ^2 s m a n stupni volnosti:

$$F = \frac{\frac{\chi^2_m}{m}}{\frac{\chi^2_n}{n}} = \frac{\chi^2_m n}{\chi^2_n m}$$

Funkce hustoty $f(F)$ tohoto rozložení má tvar

$$f(F) = \frac{1}{B(\frac{m}{2}, \frac{n}{2})} (\frac{m}{n})^{\frac{m}{2}} F^{\frac{m}{2}-1} (1 + \frac{m}{n} F)^{-\frac{m+n}{2}}, \quad F > 0$$

kde m a n mají stejný význam jako v definičním vztahu.

Charakteristiky:
Střední hodnota: $E(F) = \frac{n}{n-2}, \quad n > 2$
Rozptyl: $D(F) = \frac{2n^2(m+n-2)}{m(n-2)^2(n-4)}, \quad n > 4$

Pro generování náhodných čísel s rozložením F vyjdeme z definice — vygenerujeme dvě náhodná čísla s rozložením χ^2_m a χ^2_n a dosadíme do definičního vztahu.

Rozložení gama

Toto rozložení je hlavním rozložením matematické statistiky pro náhodné veličiny s oborem hodnot z jedné strany ohraničeným. Funkce hustoty tohoto rozložení má tvar:

$$f(x) = \frac{1}{\Gamma(m)d^m} e^{-\frac{x}{d}} x^{(m-1)}, \quad \text{pro} \quad x \geq 0 \quad m, d > 0$$

Konstanty m a d jsou parametry rozložení, přičemž m je parametr tvaru a d je parametr rozměru.
Charakteristiky rozložení:

Střední hodnota: $E(X) = md$
Rozptyl: $D(X) = md^2$
Normovaná šikmost: $\gamma_1 = \sqrt{\beta_1} = \frac{2}{\sqrt{m}}$
Normovaná špičatost $\beta_2 = \frac{3(m+2)}{m}$ je kladná, protože rozložení je zleva omezeno.
Uvažujeme-li celé číslo, pak Γ-funkci ve funkci hustoty můžeme nahradit faktoriálem

$$f(x) = \frac{1}{(m-1)!d^m} e^{-\frac{x}{d}} x^{m-1}$$

a dostáváme *Erlangovo rozložení* jako zvláštní případ Γ rozložení. Je-li navíc $m = 1$, pak dostáváme exponenciální rozložení. Erlangovo rozložení popisuje čas, nutný pro výskyt m nezávislých náhodných jevů, jestliže nastávají s konstantní pravděpodobností $\frac{1}{d}$. Významnou oblastí využití tohoto rozložení jsou procesy hromadné obsluhy, kdy požadavek prochází m zařízeními a v každém stráví časový úsek, určený exponenciálním rozložením s parametrem d. Pak celkovou dobou obsluhy je náhodná veličina s Erlangovým rozložením o parametrech m a d.

Generování čísel o rozložení Γ je obtížnější a zvláštní postupy zde nebudeme uvádět.

Rozložení beta

Pro úplnost uvedeme rozložení beta, popisující náhodné veličiny ohraničené z obou stran. Funkce hustoty pravděpodobnosti má tvar

$$f(x) = \frac{1}{B(p,q)} x^{p-1}(1-x)^{q-1}, \quad x \in (0,1)$$

kde parametry p, q funkce jsou kladná reálná čísla.

Charakteristiky:
Střední hodnota: $E(X) = \frac{p}{p+q}$
Rozptyl: $D(X) = \frac{pq}{(p+q)^2(p+q+1)}$
Existuje několik způsobů generování náhodných čísel s rozložením beta. Pro ilustraci uvedeme alespoň jednu možnost. Náhodnou veličinu X s rozložením β můžeme vyjádřit jako poměr dvou náhodných veličin X_1 a $X_1 + X_2$ s rozložením gama. Pro parametry platí:

$$m_1 = p \quad m_2 = q \quad d_1 = d_2 = 1$$

5.4.2 Diskrétní náhodné veličiny

Binomické rozložení

Náhodnou veličinu s binomickým rozložením dostaneme na základě této úvahy: Provedeme-li po sobě n nezávislých pokusů a sledujeme nastoupení určitého jevu s konstantní pravděpodobností p, pak pravděpodobnost, že právě x_i pokusů bude úspěšných a $n - x_i$ neúspěšných, je rovna při daném uspořádání

$$p^{x_i}(1-p)^{h-x_i} \quad x_i \in \{0, 1, 2, ..., n\}.$$

přitom počet možných posloupností jevů je vyjádřen kombinací x_i-té třídy z n prvků. Pravděpodobnost, že dostaneme právě x_i úspěšných pokusů, je

$$f(x_i) = \binom{n}{x_i} p^{x_i}(1-p)^{n-x_i} \quad 0 \leq x_i \leq n,$$

což je základní zákon binomického rozložení náhodné veličiny X. Distribuční funkce

$$F(x) = \sum \binom{n}{x_i} p^{x_i}(1-p)^{n-x_i}$$

přičemž sumace jde přes všechna x_i vyhovující nerovnostem

$$0 \leq x_i \leq n \quad x_i \leq n$$

Charakteristiky rozložení:
Střední hodnota: $E(X) = np$
Rozptyl: $D(X) = np(1-p)$
Lze dokázat, že pro rostoucí n se rozložení této náhodné veličiny X blíží normálnímu rozložení $N(np, np(1-p))$. Alternativní rozložení, jako zvláštní případ rozložení binomického, je pro $n = 1$ a tedy $x \in \{0, 1\}$.
Metody generování:

Alternativní rozložení získáme na základě inverzní transformace. Jestliže nvx má rozložení $R(0, 1)$, pak, je-li $x \leq p$ má náhodná veličina hodnotu 1 a při $x > p$ je její hodnota nulová. V Pascalu můžeme funkci realizovat také jako booleovskou, jež nabývá hodnot logické 1 nebo 0 (`true` nebo `false`).

```
function ALTERNATIV(P:real): Boolean;
var X: real;
begin
   X := Random;
   ALTERNATIV := x <= p
end;
```

Binomické rozložení generujeme jako součet n hodnot náhodné veličiny X s alternativním rozložením s parametrem p

$$Y = \sum_n X(p)$$

a další metodou může být metoda inverzní transformace.

Geometrické rozložení

Při provádění řady nezvávislých Bernoulliových pokusů pro nastoupení sledovaného jevu má pravděpodobnost nastoupení jevu až v $(x_i + 1)$-tém pokusu (po x_i neúspěšných) tvar

$$f(x_i) = p(1-p)^{x_i},$$

kde p je konstantní pravděpodobnost nastoupení jevu v jednom pokusu. Distribuční funkci vyjádříme jako součet geometrické řady:

$$F(x_i) = 1 - (1-p)^{x_i+1}$$

Geometrické rozložení je zvláštním případem Pascalova rozložení. Náhodnou veličinu Y s tímto rozložením generujeme opět na základě alternativního rozložení s parametrem p. Opakovaně generujeme hodnoty $A(p)$ tak dlouho, až nastane případ $A_i(p) = 1$ pak $Y = i - 1$. Další možností je opět metoda inverzní transformace.

Hypergeometrické rozložení

Toto rozložení má náhodná veličina X, udávající počet vybraných prvků s určitou sledovanou vlastností při výběru bez vrácení ze souboru N prvků. Vybereme celkem n prvků, vykazujících danou vlastnost. Frekvenční funkce rozložení je

$$f(x_i) = \frac{\binom{M}{x_i}\binom{N-M}{n-x_i}}{\binom{N}{n}}, \quad \text{kde} \quad x_i \in \{\max\{0, M-N+n\}, ..., \min\{M, n\}\}.$$

Charakteristiky rozložení:
Střední hodnota: $E(X) = n\frac{M}{N}$
Rozptyl: $D(X) = n\frac{M}{N}(1 - \frac{M}{N})\frac{N-n}{N-1}$
Tohoto rozložení se používá zejména při statistické kontrole jakosti.

Poissonovo rozložení

Náhodná veličina X s Poissonovým rozložením má frekvenční funkci
$f(x_i) = \frac{\lambda^{x_i}}{x_i!}e^{-\lambda}$, kde $\lambda > 0$ je reálný parametr $x_i \in \{0, 1, ...\}$
Tímto rozložením lze pro dosti velká λ aproximovat binomické rozložení s parametry λ a $p < 0.1$.

Charakteristiky rozložení:
Střední hodnota: $E(x) = \lambda$
Rozptyl: $D(x) = \lambda$
Normovaná šikmost: $\beta_1 = \frac{1}{\sqrt{\lambda}}$
Špičatost $\beta_2 = \frac{1}{\lambda} + 3$
Poissonova rozložení se často používá v systémech hromadné obsluhy, kde je jeho výskyt typický zejména u řídce se vyskytujících jevů.

Při generování můžeme použít např. metody inverzní transformace. Pro $\lambda \to \infty$ můžeme Poissonovo rozložení aproximovat normálním rozložením. Má-li náhodná veličina Z normované normální rozložení, pak hledanou hodnotu získáme podle vztahu:

$$X = Z\sqrt{\lambda} + \lambda$$

5.5 Testování náhodných čísel

Způsoby vytváření posloupností pseudonáhodných čísel jsou založeny na předpokladech, které odpovídají realizaci daného zákona rozložení pravděpodobností jen přibližně. O vlastnostech pseudonáhodných generátorů náhodných čísel bývá předem něco známo – střední hodnota, rozptyl, korelace apod. Všechny tyto hodnoty se však týkají celé nekonečné realizace posloupnosti náhodných čísel (celé periody) a jaká je jejich stabilita v závislosti na počáteční podmínce, nebylo dosud vyjasněno. Dnešní stav je takový, že si můžeme být jisti, že generátor má určitou vlastnost, jen když tuto vlastnost potvrdí statistické testy. Těchto testů bylo doposud navrženo velmi mnoho. Jejich cílem je prověřit, zda můžeme zkoumanou posloupnost náhodných čísel považovat za náhodný výběr ze souboru náhodných čísel s určitým pravděpodobnostním rozložením.

Pro testování náhodných čísel nejprve zavedeme univerzální statistické kritérium χ^2, kterého využíváme téměř ve všech testech tímto způsobem:

Všechna náhodná čísla rozdělíme do k kategorií a provádíme n navzájem nezávislých pokusů, to znamená, že n-krát generujeme veličinu s rovnoměrným nebo jiným pravděpodobnostním rozložením. Symbolem p_s označíme pravděpodobnost, že výsledek pokusu padne do kategorie s a y_s nechť je počet pokusů, které skutečně padly do kategorie s. Zformulujeme pak vztah

$$V = \sum_{1 \le s \le k} \frac{(y_s - np_s)^2}{np_s} \qquad (CHI)$$

kde V je výsledná hodnota testu χ^2.

Když jsme získali hodnotu V, musíme určit, zda je tato hodnota vyhovující. Zavádíme proto stupeň volnosti, jenž je v našem případě roven

$$\nu = k - 1,$$

to znamená, že stupeň volnosti je o 1 menší, než je počet kategorií.

Známe-li výsledek testu χ^2 a stupeň volnosti, můžeme podle statistických tabulek určit hladinu významnosti, které tato hodnota odpovídá.

Hladina významnosti je předem zvolená pravděpodobnost, která je dostatečně malá pro zamítnutí hypotézy o rozložení daného náhodného čísla. Hladinu významnosti značíme symbolem α.

Při využití testu χ^2 postupujeme zpravidla takto:

1. Zjistíme hodnotu V.
2. Pro tuto hodnotu a pro příslušný stupeň volnosti najdeme odpovídající hladinu významnosti, případně její nejbližší nižší hodnotu.

Generátor náhodných čísel pokládáme za vyhovující, je-li hladina významnosti větší, než 0.95. Samozřejmě musíme dbát na to, aby byl počet pokusů n co největší. V případě, že je teoretický počet čísel, které padnou do s-té kategorie, menší než 5, nezahrnujeme tuto kategorii do výpočtů a stupeň volnosti ν je tedy nižší.

5.5.1 Testy rovnoměrnosti rozložení

V těchto testech využíváme vzorce (CHI), podle něhož přímo srovnáváme skutečnost s teoretickými hodnotami.

Test na rovnoměrnost

Interval, do něhož padnou generované pseudonáhodné veličiny, se rozdělí do n podintervalů; pravděpodobnost, že generovaná veličina padne obecně do k-tého intervalu je $1/n$. Pak statisticky zhodnotíme kritériem χ^2 teoretický počet generovaných veličin, které padnou do jednotlivých intervalů a jejich skutečný počet. Můžeme také brát v úvahu dvojice až n-tice sousedních čísel a zjišťovat, kolik jich padne do dvou až n-rozměrných intervalů.

Rovnoměrnost dvojic

Jednotkový čtverec rozdělíme na větší počet (asi 100) dílů. Potom bereme po sobě následující dvojice náhodných čísel a zjišťujeme, kolik jich padne do jednotlivých intervalů (dvojici považujeme za souřadnice bodu, který padne do jednotkového čtverce). V jedné sérii zkoumáme asi 50 000 hodnot. Zhodnocení výsledků metody provedeme opět testem χ^2.

Rovnoměrnost trojic

Při tomto testu postupujeme stejně jako v předcházejícím, bereme však trojice náhodných čísel a uvažujeme jednotkovou krychli. Volíme asi 1 000 intervalů a 30 000 náhodných čísel.

Rozložení maxima z n členů

Bereme n-tice po sobě následujících náhodných čísel $x_1, x_2, x_3, ..., x_n$. Vypočítáme hodnoty $U = [\max(x_1, x_2, ..., x_n)]^n$. Takto vzniklou posloupnost hodnot U testujeme na rovnoměrnost. Volíme $n = 2, 3, 4, 5, 10$ a v jednom pokusu testujeme 100 000 hodnot.

5.5.2 Testy náhodnosti rozložení

Rozhodnutí o náhodnosti posloupnosti pseudonáhodných čísel není jednoduché, poněvadž v konečné posloupnosti pseudonáhodných čísel můžeme vždy nějakou zákonitost najít. Proto zjišťujeme, do jaké míry je tato posloupnost náhodná a míru kvantitativně ohodnotíme.

Test na intervaly

V tomto testu prověřujeme délku posloupnosti pseudonáhodných čísel mezi dvěma hodnotami, které padnou do téhož, předem zvoleného, intervalu. Vyčíslíme tedy počty posloupností stejných délek a statisticky je zhodnotíme kritériem χ^2.

Zavedeme-li hodnotu $p =$*horní mez intervalu – dolní mez intervalu*, pak pravděpodobnost, že posloupnost bude mít délku t, je: $p_t = p(1-p)^t$, tedy pro posloupnost délky nula platí $p_0 = p$. Tohoto testu se dá využít pouze pro generátor čísel typu `real` s rozložením $R(0, 1)$.

Test sběratele kuponů

Zjišťuje se délka posloupnosti pseudonáhodných čísel taková, že se v ní každý možný člen posloupnosti objeví alespoň jednou. Pro kritérium χ^2 se používá vztahu

$$q_r = 1 - \frac{d!}{d^r} \left\{ \begin{array}{c} r \\ d \end{array} \right\},$$

kde q_r je pravděpodobnost, že posloupnost délky r neobsahuje všechna čísla generovaná generátorem, d je největší možné číslo, které generátor může vytvořit, zvětšené o 1, $\left\{ \begin{array}{c} r \\ d \end{array} \right\}$ je Stirlingovo číslo druhého druhu (viz *tab. 5.1*). Testu sběratele kuponů se dá použít pro generátor čísel typu integer v intervalu $\langle 0, d-1 \rangle$. Stirlingova čísla se zapisují ve tvaru: $x = \left\{ \begin{array}{c} n \\ k \end{array} \right\}$

Tabulka 5.1: Stirlingova čísla 2. druhu

	k								
n	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0
3	0	1	3	1	0	0	0	0	0
4	0	1	7	6	1	0	0	0	0
5	0	1	15	25	10	1	0	0	0
6	0	1	31	90	65	15	1	0	0
7	0	1	63	301	350	240	21	1	0
8	0	1	127	966	1701	1050	266	28	1

Test mezer

Uvažujeme-li tři sousední čísla posloupnosti, pak existuje právě 6 možností vzájemných relací:

$$a < b < c, a > b > c, a > c > b, a < c < b, c > a > b, c < a < b$$

Možnost, že by se dvě nebo tři čísla v trojici rovnala, vylučujeme; při vhodně zvolených konstantách generátoru se to nestává. Zjistíme, kolikrát se ve zkoumané posloupnosti objeví každá z těchto možností a statisticky je zhodnotíme testem χ^2 s tím, že předpokládáme, že pravděpodobnost je vždy rovna 1/6.

Poker test (test pětic)

Test na rovnoměrnost by mohl dát dobré výsledky i v případě posloupnosti čísel, která je nedostatečně náhodná. Příkladem takové posloupnosti je např.

11112222333344445555666677778888 atd.

Je zřejmé, že počet členů posloupnosti, které padnou do jednotlivých intervalů, sice odpovídá rovnoměrnému rozložení, ale tato posloupnost se ani zdaleka nepodobá posloupnosti náhodné. Proto zavádíme poker test: Posloupnost generovaných čísel rozdělíme do pětic a zjistíme počet různých čísel v pětici, který označíme např. písmenem r. Pak použijeme kritéria χ^2 s pravděpodobnostmi

$$p_r = \frac{d(d-1)...(d-r+1)}{d^k} \left\{ \begin{array}{c} k \\ r \end{array} \right\}$$

kde p_r je pravděpodobnost, že v pětici bude právě r různých čísel, k je počet čísel ve skupině (v našem případě je to 5), d je maximální možná hodnota generované veličiny zvětšené o 1, a $\left\{ \begin{array}{c} k \\ r \end{array} \right\}$ je Stirlingovo číslo druhého druhu.

Tohoto testu se dá využít pouze pro testování posloupností, jejichž členy jsou čísla typu integer a padnou do intervalu $\langle 0, d-1 \rangle$.

Dále existuje třída empirických testů, které nevyužívají kritéria χ^2. Z těchto testů uvedeme pouze Hammingův test.

Hammingův test

Tento test má odhalit, zda se některé hodnoty náhodných čísel nevyskytují s větší četností. Zkoumáme velikost součtu

$$\sum_{i=1}^{n-1}(y_i - 0.5)(y_{i+1} - 0.5) \doteq 0$$

Je-li generátor dobře navržen, je hodnota tohoto součtu přibližně nulová.

5.5.3 Testování transformovaných rozložení

Při realizaci generátorů náhodných čísel na počítači s použitím transformačních procedur si musíme ověřit jejich správnou činnost. V našem případě vycházíme ze základního rozložení $R(0,1)$, které transformujeme na libovolné rozložení. Největší nepřesnost transformovaných náhodných veličin tedy může způsobit chyba v základním rozložení. Přesto, i při použití správně navrženého základního rozložení, musíme provést testování transformovaných veličin, neboť některé chyby, jež se zde projevují, v základním generátoru nedokážeme odhalit.

Test dobré shody χ^2

Základním testem, jehož jsme pro testování transformovaných náhodných veličin použili, je test dobré shody χ^2, který odhalí nejen správnost parametrů rozložení, ale testuje i tvar rozložení. Celý algoritmus testu můžeme shrnout do čtyř základních kroků:

1. Vygenerování kontrolovaného souboru hodnot náhodné veličiny s daným rozložením.
2. Získání srovnávacího souboru se správnými parametry a tvarem.
3. Srovnání obou souborů výpočtem hodnoty testu χ^2.
4. Zhodnocení výsledků testu na základě určité předem zvolené hladiny významnosti.

Ad 1) Vygenerujeme velkou sérii n hodnot testované náhodné veličiny (použijeme $n = 10\,000$). Na základě znalosti předpokládaného oboru hodnot M náhodné veličiny provedeme jeho dělení na m (volíme $m = 100$) stejných, navzájem disjunktních tříd. Pak provedeme třídění souboru náhodných čísel tak, že hledáme četnosti, se kterými náhodná čísla padnou do jednotlivých tříd dělení oboru hodnot M. Je-li oborem hodnot M interval $\langle x_1, x_2 \rangle$, pak při dělení tohoto intervalu na m tříd je pro hodnotu x náhodné veličiny indexem třídy, do které hodnota padne,

$$j = entier((x - x_1)\frac{m}{x_2 - x_1} + 1).$$

Provedením tohoto dělení získáme zpracovaný soubor četností, jehož použijeme ve výpočtu hodnoty testu.

Ad 2) Kontrolní (srovnávací) soubor můžeme získat několika způsoby:

a) Použitím distribuční funkce $F(x)$ daného rozložení, kdy kontrolní četnosti n_j odpovídají třídám dělení oboru M, získáme jako součin počtu generovaných hodnot n a rozdílu hodnot distribuční funkce $F(x)$ v krajních bodech j-té třídy oboru M

$$n_j = n[F(x_1 + j\Delta x) - F(x_1 + (j-1)\Delta x)],$$

kde $\Delta x = \frac{x_2 - x_1}{m}$ je velikost tříd.

b) Použitím funkce hustoty daného rozložení, kde četnost v j-té třídě je

$$n_j = n \int_{x_1+(j-1)\Delta x}^{x_1+j\Delta x} f(x)dx$$

pro případ, že neznáme distribuční funkci rozložení.

c) U diskrétních náhodných veličin volíme třídy tak, aby každá obsahovala právě jednu hodnotu oboru hodnot náhodné veličiny, příp. vždy stejný počet možných hodnot veličiny. Pak můžeme využít frekvenční funkce $f(x_j)$

$$n_j = nf(x_j)$$

Dělení oboru hodnot M v 1. a 2. kroku algoritmu musí pro správný výsledek testu souhlasit.

Ad 3) Provedeme srovnání obou souborů četností — kontrolovaného a kontrolního — výpočtem hodnoty testu dobré shody podle vztahu

$$\chi^2 = \sum_{j=1}^{m} \frac{(N_j - n_j)^2}{n_j}, \quad n_j > 5$$

kde N_j jsou četnosti kontrolovaného souboru a n_j četnosti kontrolního souboru. Výpočet omezíme pouze pro četnosti $n_j > 5$, což je podmínkou správnosti testu.

Ad 4) Test zhodnotíme na základě tabulky rozložení χ^2. Předpokládáme-li určitou hladinu významnosti p naší hypotézy, pak jestliže $\chi^2 \leq x_p$, kde χ^2 je hodnota testu a x_p je kvantil rozložení pro danou hladinu významnosti p a pro počet stupňů volnosti, jenž odpovídá počtu četností $n_j > 5$ v sumě výrazu pro výpočet hodnoty testu. Jestliže $\chi^2 > x_p$, pak transformace nevyhovuje.

5.6 Metoda Monte Carlo

Metoda Monte Carlo je experimentální numerická metoda, která řeší danou úlohu experimentováním se stochastickým modelem, v němž se využívá vzájemného vztahu mezi hledanými veličinami a pravděpodobnostmi, se kterými nastanou určité jevy. Za zakladatele této metody se považují američtí matematikové J. von Neumann a S. M. Ulam, kteří v r. 1949 publikovali článek "The Monte Carlo method". Historicky prvním příkladem použití metody Monte Carlo je Buffonova úloha pro určení hodnoty čísla π.

Buffonova úloha

V rovině je narýsována soustava rovnoběžek ve stejných vzdálenostech $2a$, kde $a > 0$. Na rovinu házíme shora jehlu délky $2b$; platí, že $0 < b < a$. Máme určit pravděpodobnost, že jehla protne některou rovnoběžku (=jev A).

Řešení: Nechť y značí vzdálenost středu S jehly od nejbližší rovnoběžky a Θ je úhel sevřený jehlou a uvedenou rovnoběžkou. Veličinami y a Θ je poloha jehly vzhledem k uvažované rovnoběžce jednoznačně určena — viz *obr.* **??**.

Všechny možné polohy jehly vzhledem k nejbližší rovnoběžce jsou reprezentovány všemi body (Θ, y) obdélníka Ω určeného intervaly $\Theta \in \langle 0, \pi \rangle$, $y \in \langle 0, a \rangle$. Uvažovaná jehla protne některou rovnoběžku, bude-li platit, že

$$y \leq b \sin \Theta$$

Proto je jevu A příznivá oblast Δ, která je ohraničena křivkou $y = b \sin \Theta$ a osou Θ — *obr.* **??**. Poněvadž $|\Omega| = a\pi$

$$|\Delta| = \int_0^{\pi} b \sin \Theta d\Theta = [-b \cos \Theta]_0^{\pi} = 2b,$$

hledaná pravděpodobnost

$$p(A) = \frac{|\Delta|}{|\Omega|} = \frac{2b}{\pi a}$$

Pravděpodobnost $p(A)$ je přímo úměrná velikosti jehly a nepřímo úměrná vzdálenosti rovnoběžek. Položíme-li $a = b = 1$, dostaneme jednoduchý vztah

$$p(A) = \frac{2}{\pi}$$

Budeme-li házet jehlu na papír, na který jsem nakreslili soustavu rovnoběžek od sebe stejně vzdálených, mohou nastat tyto případy:

- jehla spadne na papír — uskutečnění pokusu;
- jehla protne některou rovnoběžku — příznivý výsledek pokusu;
- jehla spadne mimo papír — neuskutečnění pokusu.

Označíme-li n počet uskutečněných pokusů a m počet příznivých případů, pak relativní četnost $\frac{m}{n}$ je odhadem pravděpodobnosti $p(A)$ jevu A, že jehla protne některou rovnoběžku. Jde tedy o vyjádření téže pravděpodobnosti, a to jednou vzorcem klasické pravděpodobnosti

$$p(A) = \frac{m}{n}$$

a podruhé vzorcem pro geometrickou pravděpodobnost

$$\frac{|\Delta|}{|\Omega|} = \frac{2b}{\pi a}$$

Platí tedy vztah

$$\frac{2}{\pi} \doteq \frac{m}{n},$$

ze kterého můžeme přibližně určit hodnotu čísla

$$\pi \doteq \frac{2n}{m}$$

Je zřejmé, že čím více pokusů uskutečníme, tím přesnější hodnotu čísla π získáme.

Na tomto příkladě můžeme ukázat tři charakteristické vlastnosti metody Monte Carlo.

1. Při použití metody Monte Carlo nemusíme znát přesné vztahy mezi danými a hledanými veličinami úlohy; stačí jen vyjasnit souhrn podmínek, za kterých nastane příslušný jev.

2. Jednoduchá struktura algoritmu; jde pouze o vyhodnocení výsledku jednoho pokusu — v našem případě o vyhodnocení, zda jehla protne některou rovnoběžku. Tento postup opakujeme n-krát a jednotlivé pokusy jsou na sobě nezávislé. Proto se metoda Monte Carlo nazývá také metodou statistických zkoušek.

3. Chyba metody je úměrná výrazu $\sqrt{D/n}$, kde D je jistá konstanta a n je počet pokusů. Z tohoto vztahu vidíme, že metodou Monte Carlo nemůžeme dosáhnout velké přesnosti výpočtu. Chceme-li chybu metody zmenšit desetkrát, musíme zvýšit počet pokusů stokrát. Metoda je dostatečně efektivní, stačí-li nám výsledek s přesností 5 – 20%. Odhad přesnosti výpočtu metodou Monte Carlo má pravděpodobnostní charakter. Můžeme pouze udat meze, za které chyba nevzroste s pravděpodobností blízkou jedničce.

Metodou Monte Carlo se řeší např. některé soustavy lineárních rovnic, invertování matic, výpočet vícerozměrných integrálů, řešení Dirichletova problému, řešení funkcionálních rovnic různých typů aj. Metody Monte Carlo lze použít i pro řešení různých úloh jaderné fyziky. Rozvoj této metody je úzce svázán s rozvojem číslicové výpočetní techniky a programových prostředků pro generování náhodných čísel.

5.6.1 Příklad použití metody Monte Carlo pro řešení Dirichletovy úlohy z oblasti parciálních diferenciálních rovnic

Je známo, že metoda sítí není při řešení parciálních diferenciálních rovnic aplikovatelná na vícerozměrné úlohy. Dochází zde k prudkému vzrůstu mřížových bodů sítě a vzhledem k velkým nárokům na paměť a dobu výpočtu, se tato metoda stává neúnosnou i pro výkonný počítač. Proto je v těchto případech výhodné použít metody Monte Carlo, jejíž výrazná výhoda se projeví zvláště tehdy, potřebujeme-li vypočítat hodnotu jen v jednom nebo několika málo bodech sítě.

Zadání Dirichletovy úlohy: V rovině R_2 je dána oblast D a na její hranici je definována funkce $f(x, y) = f(Q)$. Hledáme řešení $u(x, y)$ Laplaceovy rovnice

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{5.1}$$

která na hranici Γ nabývá hodnot

$$u(Q) = f(Q), \quad \text{kde} \quad Q \in \Gamma \tag{5.2}$$

Zvolíme-li v rovině R_2 čtvercovou síť a převedeme-li rovnici (5.1) na diferenční, dostaneme okrajový problém

$$\begin{aligned} u(P) &= \frac{1}{4}(u(P_1) + u(P_2) + u(P_3) + u(P_4)) \\ u(Q_j) &= f(Q_j), \end{aligned} \tag{5.3}$$

kde Q_j je libovolný hraniční mřížový bod.

Vytvořme stochastický model, jenž souvisí s rovnicí (5.3). Předpokládejme, že vyjdeme z určitého, pevně nebo náhodně zvoleného vnitřního bodu sítě. S pravděpodobností 1/4 náhodně zvolíme jeden ze 4 směrů, vlevo, nahoru, vpravo, dolů. Přejdeme tak na sousední mřížový bod, z něhož pokračujeme stejným způsobem dále až obdržíme posloupnost mřížových bodů sítě, začínající bodem P a končící hraničním bodem Q. Symbolem $p(P, Q)$ označme pravděpodobnost, že se dostaneme z bodu P do bodu Q. Pravděpodobnosti, že se dostaneme z bodu P do bodů P_i (kde $i = 1, 2, 3, 4$) jsou rovny 1/4; podle věty o úplné pravděpodobnosti platí vztah

$$p(P, Q) = \frac{1}{4}(p(P_1, Q) + p(P_2, Q) + p(P_3, Q) + p(P_4, Q)) \tag{5.4}$$

Tato rovnice je pravděpodobnostním vyjádřením diferenční rovnice (5.3); splňuje okrajové podmínky:

$$\begin{aligned} p(Q_i, Q_i) &= 1 \\ p(Q_j, Q_i) &= 0 \quad \text{pro} \quad Q_j \neq Q_i \end{aligned} \tag{5.5}$$

Při daných okrajových podmínkách existuje pouze jediná funkce, splňující rovnici (5.3). Modelujeme-li náhodnou procházku, která vždy začíná v bodě P, n-krát a stanovíme-li počet m případů, v nichž náhodná procházka skončila v bodě Q, získáme vztah

$$\frac{n}{m} \approx p(P, Q) \tag{5.6}$$

Předpokládejme, že když se dostaneme do hraničního mřížového bodu Q, zaplatíme poplatek $f(Q)$. Výše tohoto poplatku představuje náhodnou veličinu, jejíž náhodnost spočívá v tom, že se z bodu P dostaneme do hraničního bodu Q_i náhodně; označíme ji $\xi(P)$. Veličina $\xi(P)$ tedy může náhodně nabývat hodnot $f(Q_1), f(Q_2), ..., f(Q_s)$, kde $Q_1, Q_2, ..., Q_s$ je množina hraničních mřížových bodů. Pravděpodobnost, že zaplatíme poplatek $f(Q_i)$ je rovna $p(P, Q_i)$; střední hodnota poplatku je určena vztahem

$$W(P) = E\xi(P) = \sum_{i=1}^{s} f(Q_i)p(P, Q_i) \tag{5.7}$$

S přihlédnutím k rovnicím (5.4) a (5.5) odtud vyplývá, že střední hodnota poplatku zaplaceného při náhodné procházce začínající v bodě P, je řešením okrajového problému (5.3). Realizace této úlohy na počítači je snadná.

Nechť má bod P souřadnice (x_0, y_0) a libovolný prvek posloupnosti souřadnice (x, y). Body $(x + h, y), (x, y + h), (x - h, y), (x, y - h)$, kde h je délka strany čtverce sítě, nazveme sousedními k bodu (x, y). Přechod z libovolného mřížového bodu do sousedního realizujeme generátorem rovnoměrně rozdělených pseudonáhodných čísel, jenž vygeneruje číslo z intervalu $(0, 1)$ které označíme μ. Bude-li

$\mu \in (0, 0.25\rangle$, pak má nový bod souřadnice $(x + h, y)$

$\mu \in \langle 0.25, 0.5\rangle$, pak má nový bod souřadnice $(x, y + h)$
$\mu \in \langle 0.5, 0.75\rangle$, pak má nový bod souřadnice $(x - h, y)$
$\mu \in \langle 0.75, 1\rangle$, pak má nový bod souřadnice $(x, y - h)$
Při řešení registrujeme tyto veličiny: souřadnice (x_0, y_0) počátečního bodu P, souřadnice obecného bodu (x, y) a počet realizovaných náhodných procházek.

Pro ilustraci dosažené přesnosti řešení uvádíme v tabulce 5.2 deterministicky vyřešenou úlohu pro čtvercovou oblast rozdělenou sítí na 5×5 uzlových bodů.

Tabulka 5.2: Zadané okrajové podmínky a správné řešení úlohy

	4	−7	10	
2	3	2	7	10
−2	4	5	6	11
8	10	8	1	−10
	20	16	0	

V tabulce 5.3 jsou uvedena výsledná řešení pro vnitřní body sítě, kterých bylo dosaženo při určitém počtu pokusů. Se zvětšujícím se počtem pokusů se řešení blíží správným hodnotám.

Tabulka 5.3: Výsledné řešení pro vnitřní síťové body

100 uskutečněných pokusů					
	1	2	3	4	5
1	0	4	−7	10	0
2	2	3.4285	3.6875	4.6666	10
3	−2	3.0909	−1.1000	2.5384	11
4	8	7.5454	8.4285	1.7500	−10
5	0	20	16	0	0

500 uskutečněných pokusů					
	1	2	3	4	5
1	0	4	−7	10	0
2	2	2.6575	4.1562	6.1162	10
3	−2	3.7894	3.2500	5.2830	11
4	8	8.8723	5.5090	0.5536	−10
5	0	20	16	0	0

10 000 uskutečněných pokusů					
	1	2	3	4	5
1	0	4	−7	10	0
2	2	3.0258	2.3953	6.7132	10
3	−2	3.8071	4.9079	6.1156	11
4	8	9.5711	7.7772	1.3668	−10
5	0	20	16	0	0

100 000 uskutečněných pokusů					
	1	2	3	4	5
1	0	4	−7	10	0
2	2	3.0196	2.1068	6.9355	10
3	−2	3.9235	4.9627	6.0009	11
4	8	9.9050	7.9593	1.0018	−10
5	0	20	16	0	0

Kapitola 6

Modelování a simulace diskrétních systémů

Základní charakteristikou diskrétního systému je změna stavu systému skokem, na základě *události*. Mezi jednotlivými událostmi se stav systému nemění a událost trvá nulovou dobu. Vztah mezi událostmi a stavem diskrétního systému je na *obr.* **??**. Symboly $E_1, E_2, ...$ reprezentují výskyty událostí, $s_0, s_1, ...$ definují stavy mezi událostmi a $T_1, T_2, ...$ časy výskytu událostí.

Do třídy diskrétních systémů patří diskrétní systémy *deterministické* a *stochastické*. Pod pojmem *diskrétní simulace* (z angl. *discrete event simulation*) se většinou rozumí simulace systémů stochastických. Tato nepřesnost v označení je dána skutečností, že se termínem diskrétní simulace zdůrazňují specifické metody modelování a nikoliv aplikační oblasti nebo specifika modelovaných systémů, jak je tomu například u systémů logických (deterministických), se kterými se seznámíme později.

Pro modelování diskrétních *stochastických* systémů, ke kterým patří především systémy hromadné obsluhy, je charakteristická aplikace stochastických metod řešení a speciálních programovacích jazyků. Stochastický charakter změn v modelovaném systému, který vyplývá ze stochastických aproximací náhodných jevů, vede k výpočtům s náhodnými čísly. Tuto metodu, při níž model přijímá podněty charakterizované určitým pravděpodobnostním rozložením a kdy usuzujeme na chování systému na základě statistických charakteristik o reakcích modelu, považujeme někdy za modifikaci metody Monte Carlo a bývá také označována jako *simulace Monte Carlo*.

Statistický charakter získaných výsledků do jisté míry implikuje účel simulačních modelů této třídy. Zpravidla jde o posouzení výkonnosti modelovaného systému, využití různých zdrojů, se kterými se v rámci systému disponuje, nebo nalezení kritických, či citlivých míst v systému. Je zřejmé, že počet aplikačních oblastí je pro tento typ simulace prakticky neohraničený. Důležité místo zde má návrh počítačových systémů na systémové úrovni.

6.1 Diskrétní simulační jazyky

Uveďme nejdříve charakteristické rysy diskrétních simulačních jazyků.

(1) *Paralelnost:* Protože po sobě jdoucí změny stavu modelu mohou být dány zcela nezávislými událostmi různých procesů, simulační jazyky využívají koncepce modelového času k uspořádání událostí podle okamžiků, ve kterých mají události nastat. Některé

jazyky mají možnost "paralelního" programování na jednoprocesorovém počítači — kvaziparalelního programování.

(2) *Velikost:* Většina systémů, které chceme modelovat na počítači, je rozsáhlá jak v objemu, tak i v rozmanitosti. Proto je společným rysem všech diskrétních simulačních jazyků *dynamické přidělování paměti*, umožňující uvolňovat paměťový prostor těch prvků modelu, které se stávají neaktivními (odcházejí ze systému), pro nové požadavky na paměť. Rozmanitost modelovaných systémů se projevuje značnou délkou simulačních programů.

(3) *Strukturální změny:* Mnoho dynamických systémů je postaveno na pojmech pohyb resp. tok, což vede ke změnám prostorové konfigurace v čase. Simulační jazyky pak obsahují prostředky pro *zpracování seznamů*, které umožňují realizovat strukturální změny v systému.

(4) *Vzájemné závislosti*: Složité vztahy mezi složkami systému často vedou k extrémně složitým závislostem, podmiňujícím výskyt některých událostí. Kromě booleovských výrazů, tak jak jsou známy z obecných programovacích jazyků, mají simulační jazyky ještě další možnosti pro popis vzájemných závislostí. Jsou odvozeny z množinové algebry a z predikátového počtu.

(5) *Stochastické jevy:* Při popisu událostí nebo procesů, jejichž kauzální vztahy nejsou známy nebo jsou irelevantní, se často používá generátorů náhodných čísel s různým pravděpodobnostním rozložením. Typickými reprezentanty těchto stochastických systémů jsou systémy hromadné obsluhy.

(6) *Statistická analýza:* Vyhodnocení chování stochastických simulačních modelů vyžaduje aplikaci metod statistické analýzy. Součástí simulačních jazyků jsou tudíž ty prostředky, které umožňují automaticky získávat a tisknout histogramy, průměry, standardní odchylky apod.

Mnohé z uvedených prostředků nelze považovat za specifické pouze pro simulační jazyky. Vývoj programovacích jazyků ukázal, že takové prvky, jako je dynamické přidělování paměti nebo zpracování seznamů, mají mnohem širší oblast aplikací a jsou proto samozřejmou součástí univerzálních programovacích jazyků.

Vývoj prostředků pro popis diskrétních simulačních modelů se datuje od počátku šedesátých let vznikem celé řady rozličných jazyků nebo specializovaných souborů procedur. Z jazyků, které jsou vybudovány jako soubory procedur v obecných programovacích jazycích, jsou nejznámější jazyky GASP, využívající prostředků jazyka FORTRAN, a SIMON, využívající prostředků ALGOLu 60. Daleko bohatší prostředky pro popis modelu i provádění simulačních experimentů však poskytují jazyky, které vyžadují speciální kompilátor. Je to např. jazyk SIMSCRIPT, jenž patří k nejstarším a nejužívanějším simulačním jazykům.

Modelovaný systém je v jazyku SIMSCRIPT popsán entitami, které jsou definovány svými vlastnostmi - atributy. Podle toho, zda entity zůstávají v modelu po celou dobu simulace či nikoliv, dělíme je na trvalé a přechodné. Toto dělení je významné především pro způsob přidělování paměti a díky tomu patří SIMSCRIPT mezi simulační jazyky málo náročné na paměť. Entity mohou být sdružovány do souborů, které lze organizovat podle pravidel FIFO (first in, first out), LIFO (last in, first out) nebo podle priorit.

Simulační běh je zajišťován *kalendářním programem*, který pracuje se záznamy událostí a zajišťuje výběr a zpracování událostí s nejnižší hodnotou času jejího výskytu. Zpracování události spočívá ve vyvolání odpovídajícího podprogramu události, který připravuje programátor. Podprogramy událostí zajišťují jednak změny stavu entit, shromažďování statistik a

jejich tisk, jednak plánování nových, resp. rušení již vytvořených událostí. Tyto, programem vytvářené události, jsou označovány jako vnitřní, na rozdíl od vnějších událostí, které jsou připraveny programátorem před zahájením simulace a do systému se zavádějí souborem vnějších událostí. Stane-li se, že ve stejném okamžiku má nastat vnější i vnitřní událost, nastane vnější událost jako první. Poznamenejme, že filozofie jazyka neumožňuje vyvolávat události podmíněně, tj. vyvolat událost teprve tehdy, dostane-li se systém do určitého stavu (jde o tzv. *interrogativní řazení událostí*).

Součástí jazyka SIMSCRIPT je zabudovaný generátor náhodných čísel a prostředky pro shromažďování a zpracování statistik.

Simulační jazyk CSL je založen na poněkud odlišném principu, než většina ostatních simulačních jazyků. Dynamika modelovaného systému se vyjadřuje nikoliv událostmi, ale činnostmi (*activities*). Výstavba modelu spočívá v určení činností, které způsobí změny stavu modelu, podmínek, které musí být splněny, aby činnosti mohly nastat, a druhů změn, k nimž dochází působením činností.

Časové hodnoty všech činností se uvádějí relativně vzhledem k aktuální hodnotě modelového času. Kromě příkazů jazyka FORTRAN jsou v CSL k dispozici příkazy pro práci se seznamy. Zabudované prostředky pro generování náhodných veličin a pro shromažďování statistik lze označit za poměrně chudé, což je asi dáno poměrně starším datem vzniku tohoto jazyka.

Velmi mocným simulačním jazykem je SIMULA. Její poslední verze, SIMULA 67, je univerzálním programovacím jazykem, který je rozsáhlým rozšířením ALGOLu 60. SIMULA obsahuje tři systémové třídy. Třída SIMSET poskytuje prostředky pro práci s obousměrně vázanými seznamy. Třída BASICIOdefinuje prostředky pro vstup a výstup. Pro simulaci je určena třída SIMULATION, která je podtřídou třídy SIMSET.

Nejdůležitějším pojmem tohoto jazyka je pojem *třídy*, resp.*podtřídy*. Deklarace třídy představuje vzor údajů a procedur, na jejichž základě lze vytvářet neomezený počet instancí. Každá podtřída libovolné třídy od ní dědí data i činnosti.

Pro systémovou třídu SIMULATION je významný pojem *proces*. Procesem zde rozumíme objekt, který je definován svými vnitřními i vnějšími vlastnostmi a předpisem o svém chování, jež může být rozloženo v čase. Každá událost v procesu se vyznačuje aktivní fází procesu - posloupností příkazů jiných než plánovacích. Končí-li aktivní fáze, lokální řízení procesu zůstává na konci plánovacího příkazu, který tuto aktivní fázi ukončil a naplánoval další aktivní fázi tohoto procesu. Jakmile přejde řízení opět na tento proces, pohybuje se lokální řízení procesu po příkazech jeho další aktivní fáze.

Práce s modelovým časem znamená práci se seznamem záznamů událostí. Práci s tímto seznamem usnadňují systémové procedury, které umožňují naplánování nové aktivní fáze procesu v zadaném čase, potlačení právě běžícího procesu na určitou i neurčitou dobu, potlačení provádění libovolného procesu apod.

Součástí systémové třídy SIMULATION jsou rovněž bohaté prostředky pro práci s náhodnými veličinami, pro shromažďování a vyhodnocování statistik.

Lze říci, že SIMULA, která jako první použila objektově orientovaný přístup, je zatím nejpropracovanějším simulačním jazykem a její význam daleko přesahuje oblast modelování a simulace.

Principy objektově orientovaného programování jsou dále rozvíjeny v modernějších jazycích, jako jsou Smalltalk a C++. Tyto univerzální jazyky jsou často využívány i pro potřeby modelování a simulace a za tím účelem jsou doplňovány speciálními knihovnami tříd. Příkladem může být simulační knihovna SIMLIB pro C++.

Alternativou imperativních jazyků mohou být jazyky funkcionální, logické a jazyky založené na reprezentaci modelu grafem (sítí).

Do třídy funkcionálních a logických jazyků patří různé modifikace jazyků LISP a Prolog pro diskrétní simulaci, které umožňují poměrně snadno vytvořit nejen simulační model a simulátor, ale i inteligentní uživatelské rozhraní, správu modelů a prostředky pro zpracování výsledků simulace včetně přípravy a realizace simulačních běhů. Z grafově orientovaných jazyků uvedeme GPSS, Q-GERT a *Petriho sítě vyšší úrovně*.

K přednostem jazyka GPSS náleží snadnost výuky, poněvadž GPSS nenavazuje na žádný jiný programovací jazyk. Jazyk GPSS je považován za snadno osvojitelný i proto, že není nadstavbou jiného programovacího jazyka, jak je tomu často v případě simulačních i jiných problémově orientovaných jazyků.

Uživatel má k dispozici soubor standardních bloků a programování v tomto jazyce spočívá ve výběru vhodných typů bloků pro danou úlohu a ve specifikaci jejich vzájemného propojení. Vytvořenou sítí bloků pak procházejí přechodné entity, zvané *transakce*, které mohou mít určitý počet vlastností. Program v GPSS pracuje tak, že generuje transakce, uvádí je do pohybu sítí bloků v příslušném modelovém čase a provádí všechny akce, které jsou důsledkem pohybu transakcí. Soubor bloků v GPSS má tyto funkce: generování a eliminace transakcí, zpožďování transakcí na určitou dobu modelového času, ovlivňování cesty transakcí modelovaným systémem, vytváření a tisk žádaných charakteristik o modelovaném systému.

Grafická forma reprezentace modelu, popsaného v GPSS, usnadňuje návrh modelu a umožňuje model dobře dokumentovat. Formalizace zápisu modelu prostřednictvím sítí je základem řady dalších jazyků, například Q-GERT.

Q-GERT má poměrně jednoduchou a srozumitelnou grafickou formu zápisu modelu, která představuje zobecnění uzlově a zároveň hranově ohodnocených grafů. Hrany v Q-GERT sítích představují většinou určité činnosti probíhající v čase, jejichž doby trvání jsou konstanty nebo hodnoty náhodných veličin. Uzly jsou použity pro modelování rozhodovacích míst, front apod.

Prvky, které se pohybují soustavou uzlů a hran, se nazývají transakce. Tento termín je použit v analogickém významu jako v jazyku GPSS. Transakce mohou představovat prvky různé povahy - reálné objekty, informace apod. Mají určitý počet parametrů, jejichž hodnoty mohou být použity pro rozlišení různých objektů, pro ovlivnění dráhy transakcí sítí, pro záznam informací apod. Hodnoty parametrů se mohou měnit při vstupu do jednotlivých uzlů sítě.

Dynamika modelovaného systému je reprezentována pohybem transakcí hranami a uzly Q-GERT sítě, která obsahuje speciální uzly, v nichž transakce vznikají a zanikají, jsou rozštěpovány na několik kopií, slučovány apod.

Jazyky GPSS a Q-GERT mají, jak uvidíme později, velmi úzký vztah k *barveným Petriho sítím* (Coloured Petri nets), které, stejně jako ostatní třídy *Petriho sítí*, mají jednu velice významnou vlastnost - možnost ověření důležitých charakteristik modelovaného systému analýzou struktury sítě. Tato možnost je dána skutečností, že Petriho síť má jednoduchou a přesně definovanou sémantiku. Význam analýzy sítě spočívá v tom, že může odhalit chyby v návrhu modelu tím, že její výsledky jsou, na rozdíl od výsledků stochastické simulace, přesné. Dále se však budeme zabývat především simulací.

Filosofii diskrétního modelování a simulace vysvětlíme v dalších částech tohoto textu jednak pomocí *Petriho sítí*, jednak s využitím objektového přístupu v rámci popisu simulační knihovny SIMLIB pro C++ v další kapitole.

6.2 Aplikace Petriho sítí v modelování a simulaci

Petriho sítě vznikly zásluhou C. A. Petriho jako prostředek pro popis paralelních činností a asynchronních událostí. Tento matematický model je založen na modelování kauzálních vztahů mezi událostmi a jejich vlivu na stav systému.

6.2.1 Paralelní systém a jeho řídicí struktura

Chování diskrétního systému můžeme modelovat obecně nedeterministickým stavovým automatem, přičemž přechod systému ze stavu s_i do stavu s_j interpretujeme jako událost u (*obr.* **??**). Událost u je zde *podmíněna* stavem s_i systému. Posloupnost událostí generovaných systémem nazveme *procesem*.

Postupnou dekompozicí zjišťujeme, že se systém skládá z podsystémů, které jsou opět systémy, a tedy jsou modelovatelné automaty s jejich vlastními stavy. Přitom každý z těchto systémů vyvíjí vlastní aktivitu, popsatelnou procesem, který probíhá *paralelně* s ostatními procesy. Systém, ve kterém po dekompozici můžeme identifikovat jednotlivé paralelně probíhající procesy, nazveme *paralelním systémem*.

V paralelním systému dochází k vzájemným interakcím systémů (synchronizaci procesů), a tyto interakce potřebujeme také adekvátně modelovat (*obr.* **??**). Je tedy třeba stavový automat vhodným způsobem modifikovat tak, aby událost u mohla být podmíněna stavy jednotlivých podsystémů (procesů). Výsledkem takové modifikace může být *Petriho síť*.

Abstrakci modelovaného systému z hlediska *podmínek* a *událostí* a z hlediska koordinace a synchronizace *procesů* nazveme *řídicí strukturou* paralelního systému. Petriho síť je modelem této řídicí struktury.

6.2.2 Petriho síť jako model paralelního systému

Okamžitý stav systému po dekompozici je určen okamžitými stavy všech jeho podsystémů, resp. stavy jednotlivých procesů. Tyto stavy podsystémů nazveme *parciálními stavy systému* a chápeme je jako dílčí aspekty celkového stavu systému. Podle *obr.* **??** je událost u podmíněna tím, že systém se nachází ve stavu s_i. Protože se však po dekompozici podílejí na stavu s_i systému jisté parciální stavy, z nichž $s_i^1, s_i^2, ..., s_i^m$ jsou určující pro výskyt události u, je na příslušné úrovni dekompozice událost u podmíněna těmito parciálními stavy. Důsledkem vzniku události u jsou pak parciální stavy $s_j^1, s_j^2, ..., s_j^n$, které se podílejí na stavu s_j systému. Modely parciálních stavů se v terminologii Petriho sítí nazývají *místa Petriho sítě* (places) a v grafu sítě se značí kroužky nebo elipsami. Skutečnost, že se systém nachází v parciálním stavu s_x^y, se graficky značí tečkou, které říkáme *značka* (token), v příslušném místě (parciálním stavu).

Model události u, která převede systém ze stavu s_i do stavu s_j, se v terminologii Petriho sítí nazývá *přechod* (transition). Přechod t se v grafu sítě značí obdélníčkem (nebo silnou čárkou), do kterého vstupují orientované hrany ze *vstupních míst*, které reprezentují *vstupní podmínky* (*vstupní místa*, podmínky proveditelnosti) přechodu t, a z kterého vystupují orientované hrany do *výstupních míst*, které reprezentují *výstupní podmínky* (*výstupní místa*, důsledky provedení) přechodu t (*obr.* **??**). Grafem Petriho sítě je tedy orientovaný graf se dvěma typy uzlů, místy a přechody, a s orientovanými hranami mezi místy a přechody (tzv. bipartitní graf).

Přechod t je *proveditelný* (událost může nastat), jestliže v každém jeho vstupním místě je značka (všechny podmínky pro uskutečnění události jsou splněny). *Provedení* přechodu t (vznik události) spočívá v odstranění značek ze všech vstupních míst a v umístění značek

do všech výstupních míst přechodu t. Okamžitý stav systému je modelován *značením* sítě, což je informace o umístění značek v místech.

Uvedené úvahy vystačily s nejvýše jednou značkou v místě, což odpovídá tzv. *C/E Petriho sítím* (*Condition/Event Petri nets*). Obecnější, *P/T Petriho sítě* (*Place/Transition Petri nets*), však pracují s libovolným počtem značek v místě. Každou hranu grafu takové sítě opatřujeme kladným celočíselným atributem, který specifikuje, kolik značek z příslušného místa je zapotřebí k provedení přechodu, resp. kolik značek se odebere/umístí z/do vstupního/výstupního místa přechodu jeho provedením (pro případ jedné značky se explicitně neuvádí). Provedením přechodu se pak z/do místa odstraní/přidá příslušný počet značek. Využití těchto i jiných modelovacích schopností Petriho sítě ukážeme později.

6.2.3 Definice Petriho sítě

Nyní budeme definovat Petriho síť, její značení a dynamické chování.

(1) *Petriho síť* N je čtveřice $N = (P, T, B, F)$, kde
$P = \{p_1, p_2, ..., p_n\}$ je konečná množina míst $(n > 0)$,
$T = \{t_1, t_2, ..., t_m\}$ je konečná množina přechodů $(m > 0)$, $P \cap T = \emptyset$,
$B : P \times T \to \mathbf{N}$ je zpětná incidenční funkce, $\mathbf{N} = \{0, 1, 2, 3, ...\}$,
$F : P \times T \to \mathbf{N}$ je přímá incidenční funkce.

(2) Množinu $\bullet t = \{p | B(p, t) > 0\}$ nazveme *vstupní množinou* přechodu t.
Množinu $t\bullet = \{p | F(p, t) > 0\}$ nazveme *výstupní množinou* přechodu t.
Místo $p \in \bullet t$ nazveme *vstupním místem* přechodu t.
Místo $p \in t\bullet$ nazveme *výstupním místem* přechodu t.

(3) Funkci $M : P \to \mathbf{N}$ nazveme *značení sítě*. Dvojici (N, M) nazveme značenou Petriho sítí. Počáteční značení sítě zapisujeme M_0.

(4) Přechod $t \in T$ je *proveditelný* v M právě tehdy, když $\forall p \in P : M(p) \geq B(p, t)$.

(5) Jestliže $t \in T$ je proveditelný ve značení M_i, pak t může být *proveden*, (fired), čímž vznikne značení M_j, ve kterém $\forall p \in P : M_j(p) = M_i(p) - B(p, t) + F(p, t)$. Zápis $M_i[t\rangle M_j$ znamená, že značení M_j je přímo dosaženo z M_i provedením přechodu t.

(6) Posloupnost přechodů $\sigma = t_1 t_2 ... t_n$ $(t_i \in T, i = 1, 2, ..., n)$ nazveme *výpočetní posloupnost*, existuje-li množina značení $\{M_1, M_2, ..., M_n\}$ taková, že $M_0[t_1\rangle M_1$, $M_1[t_2\rangle M_2$, ..., $M_{n-1}[t_n\rangle M_n$. Zápis $M_0[\sigma\rangle M_u$ znamená, že značení M_u je dosažitelné z M_0 provedením posloupnosti přechodů σ. Množina $R(N, M) = \{M' | M[\sigma\rangle M'\}$ je množina všech značení dosažitelných z M. Množina $L(N, M) = \{\sigma | M[\sigma\rangle M'\}$ je množina všech výpočetních posloupností aplikovatelných z M.

V některých případech, jako je modelování vyrovnávacích pamětí, skladů apod., je výhodné uvažovat kapacitu míst. Pak doplníme definici Petriho sítě takto:

(7) *Kapacita míst* je funkce $K : P \to \mathbf{N} \cup \{\infty\}$, která každému místu přiřazuje přirozené číslo, udávající maximální možný počet značek v místě nebo symbol ∞ pro neomezenou kapacitu. V síti se zavedenou kapacitou míst pro každé značení $M \in R(N, M_0)$ musí platit $M(p) \leq K(p)$ pro všechna $p \in P$. Kapacita ovlivní proveditelnost přechodu tak, že kromě obvyklé podmínky proveditelnosti přechodu t musí navíc pro všechna $p \in t\bullet$ platit: $M(p) - B(p, t) + F(p, t) \leq K(p)$, tj. ve výstupních místech musí být dostatek volné kapacity pro uložení značek.

Analýzou struktury sítě můžeme zjistit řadu důležitých vlastností sítě, jako je *bezpečnost*, *omezenost*, *konzervativnost* a *živost* sítě. Prostředky pro analýzu Petriho sítí, které jsou schopny o uvedených vlastnostech rozhodnout, bývají standardní výbavou systémů pro práci s Petriho sítěmi. Zájemce o metody analýzy Petriho sítí odkazujeme na [24].

6.2.4 Evoluce Petriho sítě

Dynamické chování (postupné změny značení) sítě označujeme jako *evoluci* Petriho sítě a probíhá takto:

(1) Síť je inicializována počátečním značením.

(2) Určí se množina proveditelných přechodů W, $W \subseteq T$.

(3) Je-li $W = \emptyset$, evoluce končí (došlo například k uváznutí, deadlock), jinak je z množiny W vybrán jeden proveditelný přechod $t \in W$, je proveden a pokračuje se bodem (2).

Evoluce Petriho sítě simuluje chování modelovaného systému. Příklad evoluce Petriho sítě je na *obr.* **??**.

6.2.5 Modelování pomocí podmínek a událostí

Jak již bylo naznačeno, diskrétní systém můžeme modelovat primitivy dvou typů, *událostmi* a *podmínkami*. Události jsou akce systému. Vznik událostí je řízen stavem systému, který může být popsán jako množina podmínek. Podmínka může být vyjádřena predikátem, závislým na stavu systému, a může být v daném stavu buď pravdivá (splněna), nebo nepravdivá (nesplněna).

K tomu, aby určitá událost nastala, musí být splněny některé podmínky, tzv. *vstupní podmínky* události. Vznik události může způsobit, že vstupní podmínky přestanou platit a jiné, *výstupní podmínky*, začnou platit.

Představme si, jako příklad, procesor, zpracovávající požadavky. Procesor čeká, dokud se na vstupu neobjeví požadavek, pak požadavek zpracuje a pošle jej na výstup. Podmínky (predikáty) tohoto systému jsou:

a. Procesor čeká.

b. Požadavek se objevil a čeká.

c. Procesor zpracovává požadavek.

d. Požadavek je zpracován.

Události budou:

1. Vznik požadavku.

2. Začátek zpracování požadavku procesorem.

3. Ukončení zpracování požadavku.

4. Odeslání požadavku na výstup.

Vstupní podmínky události 2 (začátek zpracování požadavku procesorem) jsou (a) procesor čeká a (b) požadavek se objevil a čeká. Výstupní podmínkou události 2 je (c) procesor zpracovává požadavek. Podobně můžeme definovat vstupní a výstupní podmínky ostatních událostí a zkonstruovat následující přehled událostí a jejich vstupních a výstupních podmínek:

Událost	Vstupní podmínky	Výstupní pomínky
1	-	b
2	a,b	c
3	c	d,a
4	d	-

Tento pohled na systém může být snadno modelován Petriho sítí. Podmínky jsou modelovány místy a události jsou modelovány přechody Petriho sítě. Vstupní místa přechodu modelují vstupní podmínky události, výstupní místa modelují výstupní podmínky události (důsledky události). Výskyt události je modelován provedením odpovídajícího přechodu. Splnění podmínky (pravdivost predikátu) je reprezentováno značkou v místě, které odpovídá podmínce. Provedení přechodu odstraní značky ze vstupních míst a uloží nové značky do výstupních míst, což znamená, že přestanou platit vstupní a začnou platit výstupní podmínky.

Petriho síť, která modeluje procesor zpracovávající požadavky, je na *obr.* **??**. Každý uzel sítě (přechod nebo místo) je označen odpovídající událostí nebo podmínkou.

6.2.6 Nezávislé a konfliktní události

Důležitým rysem Petriho sítí je jejich *asynchronní* charakter. Petriho sítě abstrahují od explicitního vyjadřování času nebo toku času a neobsahují tudíž časovou množinu, která je základní složkou definic systémů. To umožňuje modelovat situace, kde se události v systému vyskytují zcela asynchronně, v nepředvídatelných časových okamžicích. Předpokládá se přitom, že modelované události jsou *primitivní* v tom smyslu, že jejich provedení je okamžité a trvá tudíž nulový čas. Protože reálný čas je spojitou veličinou a pravděpodobnost současného výskytu více primitivních událostí je nulová, lze přijmout omezení (dané evolučními pravidly), že přechody jsou realizovány *sekvenčně*. V případě *neprimitivních* událostí, které trvají nenulový čas, je nutné jako události modelovat začátek a konec neprimitivní události (*obr.* **??**).

Asynchronnost Petriho sítí umožňující modelovat události bez explicitní specifikace okamžiků jejich začátku a ukončení vede k *abstrakci pojmu čas*. Tato abstrakce odráží z logického hlediska důležitou vlastnost času — stanovení *částečného uspořádání* výskytů událostí. Z tohoto hlediska je Petriho síť prostředkem, který určuje možné posloupnosti událostí jako výsledek interakce procesů modelovaného systému.

Při evoluci Petriho sítě je v každém kroku nejprve stanovena množina proveditelných přechodů. Obsahuje-li tato množina více než jeden prvek, pak výběr toho přechodu, který bude proveden, není nijak definován. Tento *nedeterminismus* může být řešen *interpretací* sítě (dodatečnými podmínkami pro provádění přechodů), případně omezen *zavedením času* jako doby trvání neprimitivních událostí. Čas může být specifikován buď *deterministicky* (konstantou), nebo *stochasticky* (pravděpodobnostním rozložením), což pak vede ke stochastickým modelům systémů.

Nedeterminismus a nemožnost současné realizace přechodů může vést ke dvěma možným situacím pro nějaké dva proveditelné přechody:

(1) Provedení libovolného jednoho přechodu nijak neovlivní proveditelnost druhého. Pak jde o vzájemně *nezávislé přechody* a mohou být provedeny v libovolném pořadí (*obr.* **??**).

(2) Provedení jednoho přechodu znemožní následné provedení druhého. Pak jde o *konfliktní přechody* (vzájemně výlučné přechody) a výběr jednoho z nich je řešen v rámci interpretace Petriho sítě (*obr.* **??**).

Z hlediska implementace čas není spojitou veličinou a tudíž se může stát, že několik událostí má nastat ve stejném modelovém čase. V tom případě se nedeterminismus výběru řeší podle *priorit* událostí, přičemž priorita může být určena buď explicitně (uživatelem), nebo implicitně (implementací simulátoru, například strategií FIFO).

6.2.7 Modelování procesů

Proces můžeme popsat jako sekvenci primitivních událostí, mezi nimiž je proces *potlačen*. Potlačení procesu podle doby trvání rozdělujeme do dvou skupin:

(1) potlačení procesu na předem známou dobu ΔT, které je vyvoláno zevnitř procesu;

(2) potlačení procesu na neurčitou dobu, přičemž neurčitost chápeme tak, že v okamžiku potlačení není jeho délka známa, neboť opětovná aktivace procesu může být vázána na splnění nějaké podmínky, vyskytující se v dalším průběhu jiného procesu.

Zkoumejme nyní příklad s procesorem z odstavce *6.2.5* z hlediska procesů. Jeden proces popisuje chování procesoru, zpracovávajícího postupně požadavky, další procesy popisují chování jednotlivých požadavků. Procesy, odpovídající požadavkům, zde vznikají, procházejí určitými stavy a zanikají. Navíc vzájemně koordinují svoji činnost s procesem procesoru tím, že některé události provádějí společně (začátek a konec zpracování požadavku procesorem). Tomuto způsobu synchronizace procesů říkáme *souběh*, *rendez-vous*.

Proces je modelován značkou, která se může vyskytovat v právě jednom z míst, reprezentujících možné stavy procesu. Část Petriho sítě, která obsahuje místa, reprezentující stavy určitého procesu, modeluje řídicí strukturu tohoto procesu (je to abstrakce programu, který řídí proces). Jak je vidět na příkladu s procesorem, různé řídicí struktury různých procesů mohou obsahovat společné přechody i společná místa.

Vznik procesu je modelován vytvořením značky. Ve fragmentu Petriho sítě na *obr.* **??** a na *obr.* **??** k tomu dojde provedením přechodu t. Zrušení procesu spočívá v odstranění značky, která ho reprezentuje, ze sítě (provedením přechodu t na *obr.* **??**). Možné větvení řídicí struktury procesu je na*obr.* **??**. Kterou větví proces projde, závisí na interpretaci Petriho sítě, nebo, v případě barvené Petriho sítě, na barvě značky (viz dále).

Petriho síť je schopna modelovat sdílení jedné řídicí struktury několika procesy tím, že v místech, patřících této řídicí struktuře, se v daném stavu systému nachází několik značek, z nichž každá reprezentuje jeden proces (*obr.* **??**). Při sdílení jedné řídicí struktury více procesy se však objevuje problém ztráty identity jednotlivých procesů (značky jsou všechny stejné a přitom modelují různé procesy), což může v některých případech vadit. Tento problém (*obr.* **??**) řeší *sítě vyšší úrovně* (například *barvené sítě*), které přiřazují značkám *atributy* (*barvy*) a přechodům *predikáty* (*stráže*), které na základě hodnot atributů značek ve vstupních místech rozhodují o proveditelnosti přechodů. Těmito sítěmi se budeme zabývat později.

6.2.8 Sdílené zdroje

Kromě společného provádění událostí může být kooperace procesů zajištěna *sdílením zdrojů*. Zdroje, resp. sdílené prostředky, jsou většinou chápány jako pasivní podsystémy, které nevy-

víjejí vlastní aktivitu a pouze mění svůj stav jako reakci na události. Zdroje jsou modelovány místy Petriho sítě, které nevystupují v roli stavu žádného z procesů. Značka v takovém místě může představovat buď informaci o stavu sdíleného zdroje (volný/obsazený apod.), nebo informaci předávanou jedním procesem prostřednictvím tohoto zdroje (coby komunikačního kanálu) jinému procesu, který na ni čeká (*obr.* **??**).

Poznamenejme, že existence aktivních a pasivních podsystémů (procesů a zdrojů) závisí výhradně na tom, jak chápeme odpovídající místa a jejich značení. Příklad z odstavce *6.2.5* můžeme tedy chápat i tak, že požadavky-procesy soupeří o procesor, který je sdíleným zdrojem s výlučným přístupem.

Abstrakcí zdroje s výlučným přístupem může být *semafor* (*obr.* **??**). Pracuje-li právě proces se sdíleným prostředkem s výlučným přístupem (provádí nad ním určité akce), říkáme, že se nachází v *kritické sekci*. V kritické sekci, příslušející určitému sdílenému prostředku, se smí v každém okamžiku nacházet nejvýše jeden proces.

Modelování obecnějšího typu zdroje a jeho přidělování můžeme zajistit zvýšením počtu značek v místě, které modeluje zdroj, a specifikací požadavků na počty značek ze strany procesů. Pak mluvíme o *kapacitě* zdroje nebo o *množině zdrojů* stejného druhu. Obsazení a uvolnění části kapacity zdroje je na *obr.* **??**, kde k je kapacita zdroje (počet značek v místě). V simulačních jazycích a v C++/SIMLIB se pro zdroj přidělovaný po částech používá termín *sklad*, *store*. Část kapacity skladu se obsazuje příkazem *enter*, část kapacity skladu se uvolňuje příkazem *leave*.

Simulační jazyky zavádějí *prioritu* procesů (podrobněji viz popis C++/SIMLIB). Výlučný přístup ke zdrojům může být ovlivněn prioritami procesů například tak, že proces s vyšší prioritou může donutit jiný proces dočasně uvolnit obsazený prostředek s výlučným přístupem. V simulačních jazycích a v C++/SIMLIB se takový prostředek nazývá zařízení, *facility*, obsazuje se příkazem *size* a uvolňuje se příkazem *release*.

6.2.9 Kvaziparalelní zpracování procesů

Zamysleme se z hlediska procesů nad evolucí Petriho sítě a implementací *simulátoru* Petriho sítě. Evoluční algoritmus byl popsán tak, že umožňuje zpracování na jednom procesoru. Procesy jsou zde zpracovávány *kvaziparalelně*, což znamená, že v daném časovém okamžiku se zpracovává nejvýše jedna primitivní událost procesu. Teprve když je proces potlačen (což nastává po každé primitivní události), může dojít k *přepnutí kontextu*, tj. může se zpracovat událost jiného pasivovaného procesu. Vzhledem k tomu, že v daném okamžiku může být proveditelných více přechodů (událostí), je opět výběr jednoho z nich obecně nedeterministický, prakticky je závislý na implementaci simulačního jazyka (může být ovlivněn prioritami procesů). V řadě simulačních jazyků, včetně C++/SIMLIB, k přepnutí kontextu dochází až v případě, že je proces potlačen na nenulový nebo neurčitý čas, tj. v průběhu zpracování neprimitivní události nebo během čekání na událost. Znamená to tedy, že jeden proces se zpracovává až do okamžiku, kdy se začne zpracovávat neprimitivní událost procesu (čekat určitou dobu) nebo čekat na nějakou událost (čekat neurčitou dobu).

6.2.10 Modelový čas, časovaná Petriho síť

Jak již bylo uvedeno, Petriho sítě pracují s abstrakcí času v podobě částečného uspořádání událostí. V případě, že potřebujeme modelovat systémy, ve kterých je známa například doba trvání některých neprimitivních událostí, nerespektováním konkrétních časových relací mezi událostmi bychom se ochudili o možnost získání důležitých informací o systému. V případě stochastické simulace pro zjištění výkonnosti systému hraje čas velmi významnou úlohu.

Čas se zavádí do Petriho sítí různými způsoby a byly vyvinuty metody pro analýzu Petriho sítí s časem. Pro naše potřeby zavedeme čas do Petriho sítě prostřednictvím *časovaného přechodu*. Každému časovanému přechodu je přiřazeno *zpoždění*, s jakým při provedení uloží značky do výstupních míst. Na *obr.* **??** je časovaný přechod a jeho náhradní schéma, ze kterého je vidět, jak časovaný přechod modeluje neprimitivní událost. Petriho síť s časovanými přechody nazveme *časovaná Petriho síť*.

Evoluce časované sítě se opírá o *globální hodiny* a *kalendář událostí*. Hodnota globálních hodin se nazývá *modelový čas*. Hodnoty modelového času mohou být diskrétní (integer) nebo spojité (real). V obou případech stav systému existuje v daném modelovém čase a modelový čas monotónně roste v průběhu simulace.

Kalendář událostí (kalendářní seznam) je seznam, jehož položkami jsou záznamy o plánovaných událostech. Záznam o události obsahuje *čas provedení* události a přesnou *specifikaci* události (podprogram a data). Seznam je vzestupně uspořádaný podle času naplánovaných událostí. Nad kalendářním seznamem jsou definovány dvě operace:

(1) *naplánování události* na daný čas — spočívá v zařazení záznamu o události do seznamu tak, aby zůstal vzestupně uspořádaný podle časů událostí (v našem případě se plánuje uložení značek do určitých míst v určitém čase),

(2) *provedení nejbližší naplánované události* — záznam o události se vyjme ze začátku seznamu (je to událost s nejnižším časem), modelový čas se nastaví na čas události a událost se provede (v našem případě jde o uložení značek do určitých míst).

Evoluce časované Petriho sítě probíhá takto:

(1) Síť je inicializována počátečním značením, modelový čas nastaven na nulu, kalendář událostí je prázdný.

(2) Určí se množina proveditelných přechodů W, $W \subseteq T$.

(3) Je-li $W = \emptyset$, pak:

   – je-li kalendář prázdný, evoluce končí,

   – jinak se z kalendáře vyjme událost u, modelový čas se nastaví na čas události u, provede se událost u (do specifikovaných míst se uloží specifikovaný počet značek) a pokračuje se bodem (2),

   jinak se vybere z W přechod t, není-li časovaný, provede se obvyklým způsobem, je-li časovaný, odstraní se vstupní značky a naplánuje se uložení výstupních značek na modelový čas zvětšený o hodnotu zpoždění přechodu t a pokračuje se bodem (2).

Doba zpoždění časovaných přechodů může být určena *konstantou* nebo *pravděpodobnostním rozložením*. V druhém případě jde o *stochastickou Petriho síť* a skutečná hodnota zpoždění se určí vygenerováním náhodné hodnoty podle daného pravděpodobnostního rozložení při každém provedení časovaného přechodu. Simulátor stochastické Petriho sítě tedy musí obsahovat generátory náhodných čísel s různými pravděpodobnostními rozloženími. Stochastická simulace pak spočívá v dlouhodobé (z hlediska modelového času) evoluci Petriho sítě s následným statistickým vyhodnocením, kterým můžeme zjistit průměrné počty značek v místech, histogramy četnosti událostí apod.

Stochastická síť na *obr.* **??** modeluje procesor z odstavce *6.2.5*. Přechod $t1$ s místem $p1$ představují generátor požadavků takový, že časový interval mezi vznikem dvou po sobě následujících požadavků je náhodná veličina s exponenciálním pravděpodobnostním rozložením se střední hodnotou $T1$. Přechod $t2$ modeluje zpracování požadavku procesorem,

přičemž doba zpracování je opět náhodná veličina s exponenciálním pravděpodobnostním rozložením, tentokrát se střední hodnotou $T2$. Stochastickou simulací můžeme zjistit například průměrný počet značek v místě $p3$, což odpovídá průměrné délce fronty požadavků na zpracování. Podobný, ale poněkud složitější příklad tohoto druhu, uvedeme v příloze.

6.2.11 Hierarchie v Petriho síti

Během dekompozice systému často postupujeme tak, že jeho řídicí strukturu specifikujeme nejprve hrubě a až následně ji zjemňujeme. Je výhodné, když model řídicí struktury je schopen tento postup akceptovat. Aplikace postupů shora-dolů a zdola-nahoru a respektování strukturovanosti systému patří k základním atributům *hierarchických sítí*. Zavedení hierarchie do Petriho sítí spočívá ve *formální substituci* vybraného uzlu sítě (přechodu nebo místa) podsítí. Podrobněji ukážeme substituci přechodu podsítí.

V nadřazené síti nejprve určíme přechod, který má být nahrazen podsítí, a specifikujeme tuto podsíť. Podsíť obsahuje *porty*, což jsou místa, kterými se připojuje k nadřazené síti. Nahrazovaný přechod v nadřazené síti opatříme informací o identifikaci podsítě a o přiřazení vstupních a výstupních míst tohoto přechodu portům podsítě. Příklad hierarchické sítě je na *obr.* **??**.

Sémantiku substituce přechodu podsítí definujeme konstrukcí ekvivalentní nehierarchické sítě. Tuto konstrukci provedeme ve třech krocích:

(1) Vypustíme nahrazovaný přechod včetně okolních hran.

(2) Vložíme kopii podsítě.

(3) Sloučíme každé vstupní/výstupní místo nahrazovaného přechodu s odpovídajícím portem podsítě.

Na *obr.* **??** je nehierarchická síť, která je ekvivalentní hierarchické síti z *obr.* **??**. Uvedená konstrukce nehierarchické sítě slouží pouze pro definici sémantiky hierarchické sítě — ve skutečnosti se neprovádí a pracuje se přímo s hierarchickou sítí. Hierarchie má vliv pouze na grafickou reprezentaci sítě, analýza a simulace se provádí tak, jako na odpovídající nehierarchické síti. Navíc je zde ovšem možnost *analyzovat a simulovat síť po částech*. Je však nutno poznamenat, že přechod, který je nahrazen podsítí, se už nechová jako přechod, ale jeho chování je určeno chováním odpovídající nehierarchické sítě.

Koncept hierarchie nám umožní vytvářet rozsáhlé sítě modulárním způsobem a dovoluje násobné použití téže podsítě v nadřazených sítích. Programové systémy pro modelování, analýzu a simulaci na bázi hierarchických sítí umožňují nejen specifikovat, ale i analyzovat a simulovat modelovaný systém vcelku i po částech na různých úrovních podrobností.

6.2.12 Interpretace Petriho sítě

Jak již víme, Petriho síť modeluje řídicí strukturu, která je abstrakcí modelovaného systému. Úroveň abstrakce můžeme snížit interpretací Petriho sítě.

Interpretace Petriho sítě zahrnuje několik složek:

(1) Množinu stavových proměnných systému.

(2) Každému přechodu je přiřazena funkce, která má definovány vstupní stavové proměnné a výstupní stavové proměnné. Tato funkce se volá při každém provedení přechodu. Provede výpočet nad vstupními stavovými proměnnými a výsledek uloží do výstupních stavových proměnných.

(3) Každému přechodu t_i je přiřazen predikát (*stráž*) G_i, který na základě hodnot stavových proměnných spolurozhoduje o proveditelnosti přechodu t_i. Pravidlo pro rozhodnutí o proveditelnosti přechodu t_i je doplněno tak, že kromě původní podmínky proveditelnosti musí navíc být predikát G_i pravdivý.

Tyto informace jsou postačující pro specifikaci systému. Ke specifikaci jeho provedení je nutno dodat ještě potřebné počáteční hodnoty stavových proměnných. Stráže většinou uvádíme v hranatých závorkách a triviální stráže, které se vždy vyhodnotí jako pravdivé, neuvádíme. Interpretovaná Petriho síť na *obr.* **??** odpovídá programu:

```
var x: integer;
read(x);
if x=0 then x:=x+1 else x:=0;
write(x);
```

Dodáním interpretace Petriho síti jsme specifikovali dvouúrovňový model systému, kde rozlišujeme *úroveň řídicí struktury* a *úroveň její interpretace*.

Interpretovaná Petriho síť je mocný modelovací prostředek, ale analýza této sítě nebere interpretaci v úvahu. To znamená, že výsledky analýzy nejsou pro chování interpretované sítě zcela závazné. Proto existuje snaha co největší část modelu popsat Petriho sítí, a teprve nezbytně nutné části přesunout do oblasti interpretace. Naráží se zde však na hranici vyjadřovacích schopností Petriho sítě. Proto byla vyvinuta některá rozšíření, umožňující zvýšit modelovací schopnosti Petriho sítě, nejvýznamnější z nich jsou *inhibitory* (viz dále).

Dalším problémem Petriho sítí byla nízká úroveň popisu modelu, daná složitým způsobem modelování zpracování dat a práce s datovými strukturami. Proto byly vyvinuty *Petriho sítě vyšší úrovně*, které se řadí k vysokoúrovňovým programovacím jazykům s datovými typy a abstrakcemi. K nejvýznamnějším sítím tohoto druhu patří *barvené Petriho sítě* (viz dále).

6.2.13 Inhibitory

Užitečným rozšířením základní definice Petriho sítě jsou inhibiční hrany (inhibitory), jejichž použití umožní některá rozhodování přesunout z interpretační úrovně do řídicí struktury. *Inhibiční hrana* je hrana z místa p do přechodu t, ovlivňující pravidla proveditelnosti přechodu t tak, že přechod t je proveditelný pouze při značení $M(p) = 0$, tj. když v místěp není žádná značka. Graficky je inhibiční hrana (inhibitor) znázorněna změnou šipky na konci hrany na kolečko. Z hlediska teorie vyčíslitelnosti je významné, že rozšířením o inhibiční hrany (a tím o možnost testu na nulu) získá Petriho síť vyjadřovací sílu Turingova stroje (je tedy schopna vyjádřit jakýkoli algoritmus). Sítě s inhibitory se však obtížněji analyzují.

6.2.14 Barvená Petriho síť

Doplněním značek o atributy, které mají vliv na proveditelnost přechodů a nesou konkrétní data pro interpretaci, vznikají *Petriho sítě vyšší úrovně*. Tyto sítě nevyžadují v interpretaci zavádět globální proměnné a tudíž interpretace neznehodnocuje příliš výsledky analýzy. Dále uvedeme síť, kterou lze chápat jako vysokoúrovňový jazyk pro specifikaci systémů s možností analýzy.

Zatímco standardní Petriho sítě (P/T sítě) modelují tok řízení a synchronizaci procesů, *barvené Petriho sítě* (*Coloured Petri nets*, dále CPN) jsou schopny individualizovat značky (značky jsou obohaceny o atributy, jejichž hodnotám z tradice říkáme barvy) a tím modelovat identitu procesů i tok konkrétních dat a jejich zpracování. Nejvýznamnější vlastností CPN je možnost jejich analýzy pomocí invariantů. Tento druh analýzy se sice konkrétními daty

(barvami značek) nezabývá, ale bere v úvahu datové typy (množiny barev) značek, míst a přechodů. Zde se však analýzou CPN nebudeme zabývat, zaměříme se jen na možnost specifikace modelu tímto formalismem pro potřeby specifikace modelu a jeho dokumentace. CPN budeme definovat neformálně pomocí příkladu.

6.2.15 Jednoduchý příklad CPN - alokace zdrojů

CPN na *obr.* **??** modeluje systém, ve kterém několik procesů soupeří o sdílené zdroje. Stejně jako v jiných třídách Petriho sítí je zde množina míst a množina přechodů. Místa reprezentují stavy, přechody reprezentují změny stavů. Každé místo může obsahovat několik značek a každá z nich nese hodnotu určitého datového typu (např. záznam, struktura). Hodnotu, která je přiřazena značce, nazýváme barva značky. Na *obr.* **??** jsou dva druhy procesů: tři q-procesy startují ve stavu A a cyklují přes pět různých stavů (A, B, C, D a E), zatímco p-procesy startují ve stavu B a cyklují přes čtyři různé stavy (B, C, D a E). Každý z těchto pěti procesů je reprezentován značkou, přičemž barvou značky je dvojice, jejíž první složka říká, zda značka reprezentuje p-proces nebo q-proces, a druhá složka je celé číslo (integer), které říká, kolik cyklů proces vykonal. V počátečním značení jsou tři $(q, 0)$-značky v místě A a dvě $(p, 0)$-značky v místě B.

Jsou dány tři druhy zdrojů: jeden r-zdroj, tři s-zdroje a dva t-zdroje (každý zdroj je reprezentován e-značkou v místech R, S, T). Výrazy přiřazené hranám říkají, kolik zdrojů jednotlivé procesy obsazují/uvolňují. Například, "$case\ x\ of\ p- > 2`e|q- > 1`e$" (na hraně z S do $T2$) říká, že p-proces potřebuje dva s-zdroje k tomu, aby se dostal ze stavu B do C, a q-proces potřebuje k témuž jeden s-zdroj. Operátor ‘ pro argumenty (n, c), kde n je přirozené číslo, c je barva, vrátí multimnožinu, která obsahuje n výskytů barvy c (a nic jiného). Analogicky, "$if\ x = q\ then\ 1`e\ else\ empty$" (na hraně z $T3$ do R) říká, že každý q-proces uvolňuje jeden r-zdroj při změně stavu z C do D, zatímco p-proces neuvolňuje nic (*empty* značí prázdnou multimnožinu). Poznamenejme, že procesy ani nevytvářejí, ani nekonzumují žádné značky (během jednoho cyklu je počet obsazených zdrojů roven počtu uvolněných zdrojů).

Všimneme-li si blíže CPN na *obr.* **??**, vidíme, že obsahuje tři části: *strukturu sítě*, *deklarace* a *popisy sítě*.

Struktura sítě je orientovaný graf se dvěma druhy uzlů, místy a přechody, propojené hranami tak, že hrana spojuje dva uzly různých druhů. Takový graf se nazývá bipartitní graf.

Deklarace v levém horním rohu říkají, že v tomto jednoduchém příkladu máme čtyři *množiny barev*, resp. datové typy (U, I, P a E) a dvě *proměnné* (x a i). Použití množin barev v CPN je analogické použití typů v programovacích jazycích: Každé místo má přiřazenu množinu barev, což znamená, že každá značka v tomto místě musí mít barvu z příslušné množiny barev. Analogicky k typům množiny barev nedefinují pouze aktuální barvy (které jsou prvky množin barev), ale také definují operace a funkce, které mohou být aplikovány na tyto barvy. V našem příkladě množina barev$U$ obsahuje dva prvky (p a q) a množina barev I obsahuje všechna celá čísla (přesněji, celá čísla z intervalu *MinInt..MaxInt*, který je dán implementací). Množina barev P je množina dvojic, jejichž první složka je typu U a druhá složka je typu I. Konečně, množina barev E obsahuje pouze jeden prvek, což znamená, že odpovídající značky nenesou žádnou informaci (často mluvíme v této souvislosti o bezbarvých značkách).

Každý popis sítě je přiřazen místu, přechodu nebo hraně. V *obr.* **??** místa mají tři druhy popisů: *jména*, *množiny barev*, a *inicializační výrazy*, přechody mají dva druhy popisů: jména a *stráže*, a hrany mají jen jeden druh popisů: *výrazy*. Všechny síťové popisy jsou umístěny

vedle odpovídajících částí sítě. Pro odlišení jsou v našem příkladu typy uvozeny znakem ':', inicializační výrazy znakem '=' a stráže jsou v hranatých závorkách.

Hodnota inicializačního výrazu místa je typu multimnožina nad množinou barev místa. Multimnožina může obsahovat prvky s násobností větší než 1. V případě CPN to znamená, že dvě značky v jednom místě mohou mít stejnou barvu. Jinými slovy, jedna barva se v jednom místě může vyskytovat dvakrát (i víckrát). V popisu sítě vynecháváme inicializační výrazy, které se vyhodnotí jako prázdná multimnožina.

Stráž přechodu je booleovský výraz (predikát), který musí být splněn, aby mohl být přechod proveden. V popisu sítě vynecháváme stráže, které se vždy vyhodnotí jako splněné (true). Výraz u hrany může, stejně jako stráž, obsahovat proměnné, konstanty, funkce a operace, které jsou definovány v deklaracích (explicitně nebo implicitně). Když jsou proměnné výrazu *navázány* (nahrazeny barvou z odpovídající množiny barev), výraz se vyhodnotí jako barva (nebo jako multimnožina barev), která musí být prvkem množiny barev, přiřazené místu do/z kterého hrana vede. Když se tatáž proměnná objeví víckrát ve stráži a ve výrazech hran, vedoucích z/do jednoho přechodu, všechny tyto výskyty proměnné musí být vázány na tutéž hodnotu (jsou to tzv. *sdílené proměnné*). Naproti tomu výskyty téže proměnné ve strážích a výrazech hran různých přechodů jsou zcela nezávislé (různé přechody jsou prováděny v různých okamžicích) a tedy mohou být vázány na různé barvy.

Jak už bylo uvedeno, CPN se skládá ze struktury sítě, deklarací a popisů sítě. Složitost popisu je distribuována mezi tyto tři části, což může být provedeno mnoha různými způsoby. Každá hrana z/do zdroje v příkladu na *obr.* **??** může mít přiřazen velmi jednoduchý výraz tvaru $f(x)$, kde funkce f je definována v deklarační části. Jinak můžeme též reprezentovat všechny zdroje jedním místem RES (*obr.* **??**). Operátor + ve výrazu znamená sjednocení multimnožin (například $2`s + 1`t$ je multimnožina obsahující dva výskyty s a jeden výskyt t).

6.2.16 Dynamické chování CPN

Sémantika CPN je velmi jednoduchá. Budeme ji demonstrovat na *obr.* **??**, který obsahuje jeden přechod z *obr.* **??**.

Přechod $T2$ obsahuje dvě proměnné (x a i) a před zkoumáním proveditelnosti přechodu musí být tyto proměnné *navázány* na barvy odpovídajících typů (tj. na prvky množin U a I). To může být provedeno mnoha různými způsoby. Jedna možnost je navázat x na p a i na 0. Pak dostaneme *navázání* (*binding*) $b1 = \langle x = p, i = 0 \rangle$. Jiná možnost (z mnoha dalších) je navázat například x na q a i na 37. Pak dostaneme $b2 = \langle x = q, i = 37 \rangle$. Počet těchto možností je dán mohutnostmi množin barev proměnných.

Pro každé navázání můžeme ověřit, zda přechod s tímto navázáním proměnných je *proveditelný* (v daném značení). To zjistíme vyhodnocením stráže a všech vstupních výrazů (výrazů přiřazených vstupním hranám přechodu). V našem případě je stráž triviální (chybějící stráž je vždy vyhodnocena jako pravdivá). Pro navázání $b1$ se výrazy u hran vyhodnotí jako $(p, 0)$ a $2`e$. Zjišťujeme, že přechod je pro $b1$ proveditelný, protože každé ze vstupních míst obsahuje minimálně tolik barevných značek, kolik specifikují vyhodnocené výrazy u odpovídajících hran (jedna $(p, 0)$-značka v B a dvě e-značky v S). Pro navázání $b2$ se výrazy u hran vyhodnotí jako $(q, 37)$ a $1`e$. Zjišťujeme, že pro $b2$ přechod není proveditelný (v B není $(q, 37)$-značka). Přechod může být proveden tolika způsoby, kolika způsoby mohou být proměnné ve vstupních výrazech a stráži navázány při zachování proveditelnosti přechodu.

Je-li přechod proveditelný (pro určité navázání proměnných), může být *proveden* a pak odstraní značky ze vstupních míst a uloží značky do výstupních míst. Počet odstraněných/uložených značek a jejich barvy jsou určeny hodnotami výrazů u odpovídajících hran

pro dané navázání proměnných. Provedení $T2$ pro navázání $b1$ odstraní $(p, 0)$-značku z B, odstraní dvě e-značky z S a uloží $(p, 0)$-značku do C.

6.2.17 Modelování podmínek a událostí pomocí CPN

V předešlém příkladu jsme ukázali modelování procesů prostředky CPN. Nyní ukážeme přístup, založený na podmínkách a událostech, na příkladu pro simulaci sice netypickém, ale dokazujícím, že modelování pomocí Petriho sítí může najít uplatnění i v umělé inteligenci. Jde o model, na kterém lze řešit problém ze světa kostek, klasický problém pro plánovací produkční systémy, jako je například STRIPS.

Znalosti o problému jsou u těchto systémů vyjádřeny formulemi predikátové logiky:

Predikát	Význam
on(x,y)	kostka x leží na kostce y
clear(x)	na kostce x nic neleží
armempty	robot nic nedrží
holding(x)	robot drží kostku x

Je třeba z počátečního stavu světa (*obr.* **??***a)*) přejít do cílového stavu světa (*obr.* **??***b)*) aplikací operátorů:

Operátor	Význam
pick(x)	robot uchopí kostku x
put(x,y)	robot položí kostku x na kostku y

Cílem je nalezení plánu, tedy posloupnosti aplikace operátorů, resp. posloupnosti událostí s odpovídajícími navázáními proměnných. Události, které jsme ztotožnili s operátory, a jejich vstupní a výstupní podmínky jsou tyto:

Událost	Vstupní podmínky	Výstupní podmínky
pick(x)	clear(x), on(x, y), armempty	holding(x), clear(y)
put(x, y)	holding(x), clear(y)	on(x, y), armempty

Model světa kostek můžeme implementovat pomocí CPN takto (*obr.* **??**): Každému predikátu pro popis stavu odpovídá místo CPN, každému operátoru odpovídá přechod CPN. Podmínky proveditelnosti a důsledky provedení operátorů jsou vyjádřeny propojením míst a přechodů CPN. Stav světa kostek je vyjádřen značením sítě. Značka v místě reprezentuje objekt, pro který je příslušný predikát pravdivý. Daty řízená inference pak spočívá v generování a *prohledávání grafu dosažitelných značení* sítě s cílem nalezení takové výpočetní posloupnosti, která vede od počátečního k cílovému značení. Příkladem úspěšného plánu může být výpočetní posloupnost ($pick(B)$, $put(B, D)$, $pick(C)$, $put(C, A)$).

6.2.18 Modifikace CPN

Pro praktické použití byla vytvořena celá řada sítí, odvozených od CPN. Některé druhy sítí pracují pouze s množinami barev místo multimnožin, některé omezují kapacitu míst, přístup k značkám v místech omezují na strategii FIFO, Petriho sítě s objekty (PNO), kde značkou je objekt, nedovolují současný výskyt jednoho objektu ve více než jednom místě sítě apod. Vznik těchto speciálních druhů sítí je motivován jednak problémem, který mají řešit, jednak možnostmi jejich analýzy a simulace.

Pro potřeby modelování stochastických systémů budeme kromě interpretované stochastické Petriho sítě používat také *stochastickou* CPN, která je zavedením pravděpodobnostních rozložení zpoždění přechodů odvozena od *časované* CPN. Časovaný přechod zde má tutéž sémantiku, jak byla vysvětlena v odstavci *6.2.10*. Modelování prostřednictvím stochastické CPN ukážeme na příkladu víceprocesorového systému v příloze *A* společně s modelem téhož systému v C++/SIMLIB.

Kapitola 7

Popis simulační knihovny SIMLIB

7.1 Úvod

Simulační knihovna SIMLIB je vyvíjena od roku 1990 na Ústavu informatiky a výpočetní techniky FEI VUT Brno. Knihovna je implementována na počítačích třídy PC pod MS-DOS (překladače Borland C++ nebo GNU C++) a pod operačním systémem Linux (překladač GNU C++). Knihovna je přenositelná i na jimé platformy, vyžaduje však úpravu v modulu implementujícím procesy. SIMLIB poskytuje základní prostředky pro popis spojitých, diskrétních i kombinovaných modelů a prostředky pro řízení simulace. Při tvorbě simulačních modelů a experimentování s nimi lze použít různá integrovaná prostředí, která umožňují interaktivní tvorbu a ladění modelů.

Knihovna usnadňuje efektivní popis modelů přímo v jazyce C++, není tedy nutný překladač simulačního jazyka. Tato koncepce má své výhody i nevýhody. Je možné používat všech ostatních prostředků vytvořených v C++ (např. grafické knihovny a uživatelská rozhraní). Uživatel také není nijak omezován při případném doplňování prostředků knihovny. Za nevýhodu lze považovat nemožnost dodatečných syntaktických a sémantických kontrol, které by se mohly provádět při použití překladače simulačního jazyka. Na straně uživatele se předpokládá základní znalost programování v jazyce C++.

Následující odstavce vysvětlují základní principy použití SIMLIB pro modelování diskrétních, spojitých a kombinovaných modelů. Výklad je doplněn příklady s popisem funkce jednotlivých objektů modelu.

7.2 Objektově orientovaná simulace

Model je v SIMLIB chápán jako množina prvků (entit), které jsou spolu navzájem propojeny vazbami. Tyto vazby spolu s chováním prvků určují chování systému jako celku. Podobný přístup je podstatou objektově orientovaného programování, jehož principy se poprvé objevily v šedesátých letech v jazyce SIMULA 67. Objektově orientovaný program je tvořen množinou objektů, které spolu navzájem komunikují — posílají si zprávy.

Rozdělení systému na jednotlivé objekty je závislé na účelu modelu. Mezi objekty modelu můžeme najít takové, které mají shodné podstatné charakteristiky a ty pak můžeme zařadit do jedné třídy objektů. Třída definuje vnitřní strukturu objektů, reakce objektů na vstupy

(zprávy) a vlastní chování objektu v čase.

Objekty modelu provádějí určité akce jako odezvu na přijímané zprávy a současně provádějí jiné akce, které jsou nezávislé na přijímaných zprávách. Akce mění stav objektu, jenž je dán obsahem jeho vnitřních datových struktur.

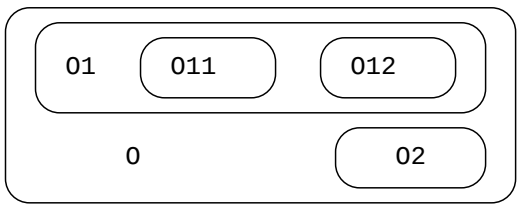
Každý objekt má definované akce, které realizuje když je vytvořen (inicializuje se) a když je rušen. Tyto akce se nazývají konstruktory a destruktory. Stejně jako ostatní akce, související s popisem chování objektu, patří k takzvaným metodám. Přijetí zprávy odpovídá vyvolání metody a metoda pak popisuje akce, kterými má objekt na tuto zprávu reagovat.

Mezi třídami objektů modelu lze nalézt jisté vzájemné vztahy. Tyto vztahy mají obvykle hierarchický charakter (částečné uspořádání). Některé třídy popisují obecné vlastnosti objektů, jiné je více konkretizují. Toho lze výhodně využít při návrhu tříd modelu tak, aby třídy konkrétní mohly využít všeho, co již definovaly třídy obecné. Tato hierarchie tříd je definována relací dědičnosti. Třída může zdědit vlastnosti své bázové třídy, přidávat nové vlastnosti a případně modifikovat vlastnosti zděděné. Takto pojatá hierarchie tříd umožňuje v maximální míře využívat již existujících tříd, což zjednodušuje a zpřehledňuje implementaci modelů. Hierarchie tříd je v podstatě uspořádáním abstrakcí, obvykle od nejobecnějších po konkrétní. Například:

```
Objekt
   Dopravní prostředek
      Automobil
         Nákladní automobil
         Osobní automobil
   Osoba
      Učitel
      Žák
```

Postup od obecných tříd ke konkrétním odpovídá postupu shora dolů, lze však postupovat i jinak. Vytvoření vhodné hierarchie tříd obvykle závisí na řešeném problému.

Hierarchická struktura existuje i na úrovni objektů modelu. Jde o vzájemný vztah objektů z hlediska jejich zahrnutí do jiných objektů jako jejich částí. Tato relace se označuje "patří do" ("is part of"), má jiné použití než dědičnost ("is kind of") a definuje hierarchii objektů.



Obrázek 7.1: Příklad hierarchického uspořádání objektů

Tento způsob popisu problémů je velmi přirozený, protože lze vyjádřit jednoznačný vztah mezi objekty modelu (programu) a objekty modelovaného systému. Metod pro analýzu problémů a návrh objektově orientovaných programů lze využít i v oblasti modelování a simulace. Proto zde uvedeme základní pojmy objektově orientovaného programování.

Objektově orientované programování je metoda implementace, při níž jsou programy tvořeny spolupracujícími skupinami objektů, z nichž každý reprezentuje instanci některé

třídy, a třídy jsou navzájem v relaci dědičnosti. Základní principy objektově orientovaného programování:

Abstrakce ukazuje na podstatné charakteristiky objektu, které jej odlišují od všech ostatních druhů objektů a tedy poskytuje přesně definované konceptuální hranice, vztažené k perspektivě pozorovatele.

Zapouzdření (encapsulation) Zapouzdřením se rozumí ukrývání všech detailů objektu, které nepřispívají k jeho podstatným charakteristikám, viditelným vně objektu.

Modularita vyjadřuje vlastnost systému, jenž byl dekomponován do množiny modulů s přesně definovanými vazbami.

Hierarchie je uspořádání nebo seřazení abstrakcí. Příkladem je hierarchie dědičnosti tříd v objektově orientovaném programování.

Tyto principy jsou pro objektově orientované programování podstatné, následující tři jsou méně důležité.

Typování je takové uplatnění třídy objektu, že objekty různých tříd nemohou být zaměňovány, nebo nanejvýš mohou být zaměňovány jen velmi omezeným způsobem.

Paralelismus (concurrency) je vlastnost, která odlišuje aktivní objekt od neaktivního. Při simulačním modelování dynamických systémů se tato vlastnost stává podstatnou.

Persistence je vlastnost objektu, která definuje dobu jeho existence v čase a prostoru. Například globální proměnné existují pouze při běhu programu, ve kterém jsou definovány, naopak soubory existují, i když program neběží.

Tyto principy nejsou v oblasti programování nové, ale objektově orientované programování je vhodným způsobem kombinuje. Požadavky, které jsou kladeny na objektově orientovaný jazyk můžeme shrnout do následujících bodů:

- jazyk musí podporovat objekty jako datové abstrakce; rozhraní objektů je definováno pomocí pojmenovaných operací, a každý objekt má (skrytý) vnitřní stav
- každý objekt je určitého typu (patří do určité třídy)
- třídy mohou dědit atributy z nadtříd

Existuje mnoho objektově orientovaných programovacích jazyků, největšího rozšíření dosáhly především jazyky Smalltalk a C++. Bližší informace o objektově orientovaném programování jsou dostupné v literatuře.

7.3 Struktura simulačního programu

Obecnou strukturu simulačního programu v C++ znázorňuje příklad:

```
#include "simlib.h"

<definice tříd>
<definice funkcí>
<deklarace globálních objektů>

<definice funkce  main  - popis experimentu>
```

Každý model musí obsahovat dovoz rozhraní simulační knihovny direktivou `#include`, potom následuje popis modelu a popis experimentu. V případě rozsáhlých modelů můžeme rozdělit popis modelu a experimentu do několika souborů (modulů), z nichž každý má tuto strukturu. Popis experimentu (funkce `main`) smí být uveden pouze v jednom modulu.

Deklarace tříd, objektů a funkcí mohou být v libovolném pořadí; platí pouze zásada, že objekt, funkci nebo třídu nelze použít před příslušnou deklarací.

7.3.1 Popis modelu

Model je tvořen množinou objektů, které jsou navzájem propojeny. Propojení objektů umožňuje jejich vzájemnou komunikaci, která definuje chování modelu.

Vytváření a rušení objektů

V objektově orientovaném popisu systémů je třeba používat různé druhy objektů podle způsobu jejich vytvoření. Podle požadavků na dobu existence objektu lze použít různé způsoby vytvoření objektů podle toho, má-li být objekt statický nebo dynamický. Předpokládáme-li třídu Zakaznik, můžeme vytvořit objekty této třídy například takto:

```
(1)  new Zakaznik;

(2)  Zakaznik *ptr;
     ptr = new Zakaznik;

(3)  Zakaznik Z;
```

V prvním případě je dynamicky vytvořen nový objekt třídy Zakaznik a takto vytvořený objekt není nijak identifikovatelný. Proto je tento způsob použitelný pouze pro objekty, na které nebude třeba se explicitně odkazovat. Druhý případ je doplněn o ukazatel na objekt třídy zákazník. Tento ukazatel potom slouží k identifikaci konkrétního zákazníka při komunikaci s ním. Dynamické objekty lze vytvářet i rušit v průběhu simulace. Třetí způsob odpovídá vytvoření globálního statického objektu s identifikací jménem (identifikátorem). Tento objekt existuje po celou dobu běhu simulačního programu.

Objekty, definované tvůrcem modelu, mohou být globální nebo lokální z hlediska jejich umístění v jiných objektech. Objekty globální jsou dostupné vždy, objekty uvnitř třídy (lokální) jsou dostupné pouze když to tato třída dovolí specifikací public. Lokální objekty jsou součástí jiných tříd objektů:

```
class X : public Process {
  Histogram H;
  double  StartTime;
  int i;
public:
  X() : H("X.H",0,0.1,10), i(0) {};
  ...
};
```

Z ukázky je patrná syntaxe volání konstruktoru lokálního objektu H. Každý objekt třídy X obsahuje histogram H, který je nedostupný objektům jiné třídy (má implicitně specifikaci private).

Další členění objektů je z hlediska časového. Objekty, které v modelu existují po celou dobu simulace, nazveme statické a objekty, které vznikají a zanikají v průběhu simulace, budeme nazývat dynamické.

Protože objekty modelu je nutné identifikovat (pro čitelný výstup), musí se při vytváření objektů některých tříd uvést jejich textové pojmenování:

globální objekty:

```
Facility  F1("Zařízení 1");
Store     S1("Sklad 1", Kapacita);
Histogram H("Četnost hodnot x",Od,Krok,10);
```

dynamicky vytvořené objekty:

```
Store *ptr;
ptr = new Store("Sklad X",10000);
```

objekty vnořené ve třídě:

```
class X {
  Facility F;
public:
  X() : F("Lokální zařízení")  {}
  ...
};
```

Pro zrušení objektu odvozeného od třídy Process lze použít metodu Cancel, která provede potřebné operace. Tatáž operace se provede implicitně po ukončení popisu chování objektu v metodě Behavior. Dynamicky vytvořené objekty lze rušit operátorem delete aplikovaným na ukazatel na objekt:

```
delete objptr;
```

Statické globální objekty nelze rušit explicitně, ruší se automaticky po ukončení simulačního programu. Jazyk C++ volá automaticky při rušení každého objektu speciální metodu — destruktor, který může provést případné operace, nutné pro jeho korektní odstranění z modelu.

7.3.2 Řízení simulace

Řízení simulačního experimentu popisuje funkce `main`. Zde se provádí inicializace modelu, vytvoření objektů modelu a jejich aktivace (tj. start procesů, které probíhají uvnitř objektů). Prototypová verze popisu experimentu je uvedena v příkladu:

```
int main() {
  <příkazy1>
  Init(<počáteční čas>,<koncový čas>);
  <příkazy2>
  Run();
  <příkazy3>
  return 0;
}
```

Na počátku funkce `main` lze provést příkazy, nesouvisející přímo s modelem a jeho inicializací (příkazy1). Zde je vhodné otevřít výstupní soubor, případně zobrazit úvodní informace o modelu.

Funkce `Init` zahajuje inicializační fázi experimentu. Nastaví počáteční modelový čas a zaznamená koncový čas pro simulaci. Zároveň inicializuje všechny objekty systému pro řízení simulace (např. kalendář). V této fázi experimentu uživatel inicializuje svoje objekty (příkazy2). Typickým příkazem v této části popisu experimentu je vytvoření, inicializace a aktivace objektů modelu.

Potom následuje vlastní simulační běh vyvolaný funkcí `Run`. Po ukončení simulace následuje obvykle tisk a vyhodnocování výsledků experimentu (příkazy3).

Posloupnost `Init(); <příkazy2> Run(); <příkazy3>` lze mnohokrát opakovat v cyklu s různými parametry modelu, což je vhodné především pro optimalizační experimenty. Je však zapotřebí opatrnosti při inicializaci objektů modelu tak, aby výchozí stav nebyl ovlivněn předchozí simulací, pokud to není žádoucí. Musí se inicializovat všechny objekty modelu, systém řízení simulace inicializuje pouze kalendář událostí, seznam pro `WaitUntil` a svůj stav.

7.3.3 Výstupy modelu

Výstup informací z modelu lze provádět několika způsoby. Použití prostředků C++ je jedním z nich. Běžně se však používá standardních prostředků knihovny SIMLIB. Většina tříd definuje metodu Output, která zapisuje stav objektu v textovém tvaru do výstupního souboru. Tímto souborem je implicitně standardní výstup. Funkce `SetOutput` umožňuje přesměrování tohoto výstupu do souboru se zadaným jménem.

Prohlížení a tisk výsledků (především spojité) simulace v různých formátech je možné programem GNUplot.

Pro výstup spojitých průběhů do výstupního souboru slouží třída `Graph`. Objekty této třídy provádí periodický zápis hodnot svého vstupu do zvláštního výstupního souboru. Perioda zápisu se určuje při vytváření objektu a je možné ji dynamicky měnit.

Příklad:

```
Integrator x(vstup);
Graph Gx("x", x, 0.01);
```

7.4 Modelový čas

V knihovně SIMLIB je modelový čas reprezentován globální proměnnou `Time`. Tato proměnná je typu `double` (čas je nezáporné reálné číslo) a její počáteční a koncová hodnota je specifikována v popisu experimentu funkcí `Init`. Interpretace jednotky modelového času je závislá na modelovaném problému. Hodnota proměnné je nastavována systémem pro řízení simulace a nelze ji měnit přiřazovacím příkazem. Použití příkazu

```
Time = 10; // chyba!
```

vyvolá chybu při překladu modelu. Pro blokové výrazy je použitelný blok `T`, který reprezentuje blokový ekvivalent proměnné `Time`. Použití `Time` v blokovém výrazu je chyba, kterou neodhalí překladač, ani kontrola při běhu programu.

7.5 Diskrétní simulace

Knihovna obsahuje standardní třídy pro popis diskrétního chování objektů, pro modelování standardních obslužných zařízení a pro sběr statistických údajů. Chování objektů lze popsat dvěma způsoby, buď s použitím událostí, nebo procesů.

7.5.1 Prostředky pro práci s náhodnými veličinami

Při simulaci diskrétních stochastických systémů je zapotřebí modelovat náhodné jevy. K tomu je nutné mít generátory náhodných čísel pro všechna potřebná rozložení. Knihovna SIMLIB obsahuje všechny obvyklé generátory ve formě funkcí, zde uvedeme pouze ty nejdůležitější:

Random

```
double Random();
```

Funkce Random generuje pseudonáhodná čísla s rovnoměrným rozložením na intervalu $< 0, 1)$. Tento generátor je použit jako základní generátor pro ostatní rozložení.

Exponential

```
double Exponential(double E);
```

Generátor exponenciálního rozložení se střední hodnotou E.

Normal

```
double Normal(double M, double S);
```

Normální rozložení se střední hodnotou M a rozptylem S.

Uniform

```
double Uniform(double L, double H);
```

Rovnoměrné rozložení na intervalu $< L, H)$.

Příklady použití generátorů

```
Wait(Exponential(10));

if(Random()<0.33) S1();
else              S2();

int pole[10] = { 10,10,20,20,20,30,30,30,30,30 };
//...
X = pole[Random()*10];

Wait(Uniform(100,150));
```

7.5.2 Události

Pro popis jednorázových dějů, které se mohou periodicky opakovat, je určena abstraktní třída `Event`. Třídy objektů, odvozené z této třídy, musí definovat chování události v metodě `Behavior`, podobně jako u procesů. Na rozdíl od procesů však popis události není přerušitelný. Pro naplánování události na čas `t` lze použít metodu `Activate(t)`.

7.5.3 Procesy

Pro popis třídy objektů s vlastním dynamickým chováním je v SIMLIB definována abstraktní třída `Process`. Každá třída, která zdědí třídu Process, musí specifikovat chování objektů této třídy v čase. Chování se popisuje v metodě `Behavior` posloupností příkazů. Základní struktura popisu třídy je uvedena v příkladu:

```
class Zakaznik : public Process {
  <atributy zákazníka>
  void Behavior()
  {
    <příkazy popisující chování objektu v čase>
  }
public:
  Zakaznik(<parametry>) { <inicializace atributů> }
  ~Zakaznik(<parametry>) { <rušení atributů> }
};
```

Popis chování objektů v metodě `Behavior` připomíná popis procedury, může však, na rozdíl od ní, obsahovat i příkazy, které způsobují čekání. To znamená, že popis chování je v modelovém čase na určitou dobu přerušitelný[1]. Po aktivaci objektu se začne provádět posloupnost operací v metodě `Behavior` stejně, jako při provádění obvyklé procedury. V okamžiku, kdy se narazí v tomto popisu činnosti objektu na příkaz, který představuje čekání, je provádění příkazů pozastaveno. Pokud proces čeká, mohou běžet ostatní procesy.

Příkaz `Wait` použitý v popisu chování objektu provádí potlačení činnosti objektu na určitou dobu (tato činnost odpovídá čekání objektu). Příkaz má tvar:

```
Wait( <aritmetický výraz> );
```

Hodnota výrazu udává dobu, po kterou bude pozastavena aktivní činnost objektu. Jestliže tedy objekt O v modelovém čase t provede příkaz Wait(d) ve svém popisu chování, pak se stane po dobu d pasivním a příští aktivace (obnovení činnosti) nastane v modelovém čase $t + d$. Okamžik $t + d$ budeme nazývat reaktivačním okamžikem procesu nebo také kritickým okamžikem procesu.

Příští kritický okamžik je skrytým atributem každého objektu. Čas příštího kritického okamžiku je stanoven buď explicitně příkazem `Wait(d)` (pak je roven `Time + d`, kde `Time` je aktuální hodnota modelového času), nebo implicitně příkazem `WaitUntil(B)` (pak je roven nejbližší hodnotě modelového času, ve kterém se predikát B stane pravdivým).

Každý objekt je charakterizován třídou, svým stavem a identitou (například jménem). V průběhu simulace může objekt vzniknout, zaniknout, může být právě aktivní a nebo může čekat (pasivní stav procesu) na určitou událost nebo na určitý stav modelu.

[1] Přerušitelné procedury jsou implementovány s využitím funkcí setjmp a longjmp ze standardní knihovny jazyka C. Musí být doplněny několika řádky kódu v jazyku symbolických instrukcí, proto SIMLIB není zcela přenositelná.

Po vzniku je objekt v neaktivním stavu. Odstartování procesu (jeho aktivace) se provede metodou `Activate`. Objekt se může sám aktivovat tím způsobem, že provede svou aktivaci jako akci v konstruktoru. Poznamenejme, že k aktivaci nového objektu může dojít až po přerušení právě probíhajícího procesu.

Po provedení posledního příkazu v popisu chování objektu tento objekt automaticky zaniká. Zánik objektu lze případně kontrolovat tím, že definujeme destruktor s požadovanými operacemi.

Kvaziparalelní provádění procesů v SIMLIB

I když sémantika modelu je postavena na paralelně probíhajících procesech, nelze ignorovat skutečnost, že vlastní výpočet (simulace) probíhá na jednom reálném procesoru. Z toho plyne nutnost řešit zpracování simulačního programu kvaziparalelně.

Principy kvaziparalelního zpracování procesů jsou popsány v kapitole **??**. Popisu chování objektu třídy odvozené ze třídy Process odpovídá příslušná metoda Behavior. Tato metoda obsahuje příkazy, které mohou měnit stav daného objektu (změnou atributů) nebo stav ostatních objektů modelu (pokud je to dovoleno). Právě běžící proces provádí akce, popsané v metodě Behavior právě aktivního objektu, který je identifikován ukazatelem Current.

Priorita procesu je definována atributem `Priority`. Při vzniku objektu je možné zadat jeho prioritu, implicitně je nejnižší, tj. nulová. Prioritu probíhajícího procesu můžeme dynamicky měnit přiřazovacím příkazem:

```
Priority = <aritmetický výraz>;
```

V případě plánování reaktivace procesů na stejný modelový čas se nejdříve provede událost procesu s vyšší prioritou (vyšší hodnotou atributu `Priority`). V případě shodných priorit procesů se dříve provede proces, který byl naplánován dříve.

7.5.4 Zařízení

Obslužné zařízení popisuje třída `Facility`[2]. Zařízení je objekt, určený k popisu specifického typu interakcí procesů, označovaného jako výlučný (exkluzívní) přístup. Problém výlučného přístupu lze formulovat takto: každý z m procesů systému požaduje takový přístup k abstraktnímu zařízení nebo zdroji Z, který vylučuje, aby v kterémkoli okamžiku sdílel zařízení Z více, než jeden proces.

S příklady výlučného přístupu se setkáváme téměř v každém systému hromadné obsluhy. Příkladem může být benzinové čerpadlo, poštovní úředník u přepážky, nebo terminál počítače. Rovněž samotný mechanismus kvaziparalelního provádění procesů, kdy pouze jediný z procesů může být aktivním, je ukázkou výlučného přístupu procesů k zařízení — reálnému procesoru. Deklarace zařízení má tvar:

```
Facility <identifikátor> ("<jméno zařízení>");
```

Příklad deklarace:

```
Facility Fac("Fac");
```

Zařízení je buď obsazeno některým objektem modelu nebo je volné. Stav zařízení je možné testovat metodou–predikátem `Busy`, která vrací nenulovou hodnotu (TRUE) v případě, že zařízení je obsazeno:

[2]Zařízení jsou navržena tak, aby se co nejvíce podobala zařízením v jazyce SOL

```
if(Fac.Busy()) Print("obsazeno\n");
```

Základní operace zařízení jsou obsazení (metoda `Seize`) a uvolnění (metoda `Release`). Třída Process má definovány také metody Seize a Release, jejich parametrem je zařízení, se kterým se pracuje:

```
Seize(<identifikátor zařízení>[,<výraz>]);
Release(<identifikátor zařízení>);
```

Každé zařízení má vstupní frontu pro požadavky na obsazení, které nelze okamžitě uspokojit. V případě, že není uveden výraz jako druhý parametr Seize, je sémantika této operace jednoduchá: je-li zařízení volné, pak se obsadí, je-li zařízení obsazeno, pak se proces pozastaví a zařadí do vstupní fronty u zařízení. Uvolnění zařízení může provést pouze ten proces, který je obsadil. V případě, že vstupní fronta je neprázdná, zařízení je uvolnění znovu obsazeno prvním objektem (procesem) z fronty.

Poznámka: Současná implementace zařízení není aktivní, tj. vstoupí-li nějaký proces přímo do fronty a zařízení je volné, nedojde k jeho obsazení (k tomu dochází jen při provádění operace Seize, resp. Release).

Příklad:

```
Seize(Fac);
Wait(20);
Release(Fac);
```

V případě uvedení druhého parametru Seize tento parametr znamená *prioritu obsluhy*. Rozsah priority obsluhy je od nuly do 255. Předchozí varianta příkazu Seize je ekvivalentní příkazu Seize s nulovou prioritou obsluhy.

Je-li zařízení Z obsazeno procesem p_1 s prioritou obsluhy o_1 a požaduje-li obsluhu zařízením Z další proces p_2 s prioritou obsluhy o_2, pak mohou nastat dva případy:

$o_1 < o_2$ způsobí přerušení obsluhy procesu p_1 a zařízení je přiděleno procesu p_2. Po skončení pokračuje obsluha p_1, pokud nedošlo k dalšímu přerušení.

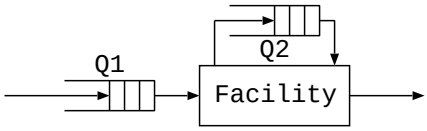
$o_1 >= o_2$ proces p_2 se zařadí do vstupní fronty zařízení Z

Z toho plyne existence další fronty u zařízení — fronty přerušených procesů. Obě fronty jsou řazeny podle těchto kritérií:

1. priority obsluhy
2. priority procesu
3. FIFO

Každé zařízení automaticky uchovává statistiky, potřebné k výpočtu průměrného využití. Výstup statistik zařízení lze provést metodou `Output`. Do standardního výstupního souboru se tisknou tyto informace:

- využití zařízení (je v rozsahu od nuly do jedné)
- maximální délka vstupní fronty
- průměrná délka vstupní fronty
- průměrná doba čekání ve frontě



Obrázek 7.2: Zařízení

7.5.5 Sklad

Na rozdíl od zařízení, pro které je charakteristický výlučný přístup, umožňuje sklad (objekt třídy `Store`) popisovat simultánní přístup ke zdroji s určitou kapacitou. Jako příklad skladu můžeme uvést parkoviště nebo paměť počítače. Sklad může obsadit více procesů v závislosti na kapacitě skladu a na požadavcích těchto procesů. Proces, který požaduje méně jednotek kapacity než je volné místo, může obsadit požadovanou část kapacity a volné místo se tím zmenší. Pokud proces požaduje více, než je volná kapacita, musí čekat až bude požadované místo volné. Zařízení lze tedy považovat za sklad s kapacitou jedna s tou výjimkou, že sklad nemá možnost přerušovat obsluhu. Deklarace skladu má tvar:

```
Store <identifikátor> ( "<jméno skladu>", <výraz-kapacita> );
```

Příklad:

```
Store Sto("Sto",100);
```

Sklad má metody pro zjištění volné kapacity (`Free`) a predikáty pro testování, je-li prázdný (`Empty`) nebo plný (`Full`). Procesy obsazují sklad operacemi `Enter` a `Leave`.

```
Enter(<identifikátor skladu>, <výraz>);
Leave(<identifikátor skladu>, <výraz>);
```

Výraz udává obsazovanou, resp. uvolňovanou kapacitu skladu. Je chybou, když požadovaná kapacita je větší, než deklarovaná kapacita skladu. Příklad ukazuje obsazení a uvolnění deseti jednotek kapacity skladu `S`:

```
Enter(S,10);
Wait(10);
Leave(S,10);
```

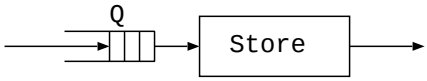
Příkaz `Enter` může způsobit čekání procesu na volnou kapacitu. Čekající procesy se řadí do fronty podle priorit, první je proces s nejvyšší prioritou. Příkaz `Leave` uvolňuje zadanou kapacitu a v případě neprázdné vstupní fronty obsazuje sklad první objekt z fronty[3].

Sklad automaticky uchovává statistiky, potřebné k výpočtu průměrného využití. Výstup statistik skladu lze provést metodou `Output`. Do standardního výstupního souboru se tisknou tyto informace:

- deklarovaná kapacita
- maximální použitá kapacita
- průměrná použitá kapacita

[3]Současná implementace se chová jinak. Podívejte se do zdrojového textu modulu 'store.cc'

- maximální délka vstupní fronty
- průměrná délka vstupní fronty
- průměrná doba čekání ve frontě



Obrázek 7.3: Sklad

7.5.6 Sběr statistik

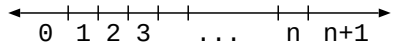
Typickou činností při simulaci diskrétních stochastických systémů je sběr a vyhodnocování statistických informací, získaných z modelu. K tomu jsou určeny standardní třídy `Histogram`, `Stat`, `TStat`.

Histogramy

Objekty třídy `Histogram` slouží k záznamu četností zapisovaných hodnot v zadaných intervalech. Deklarace objektu této třídy má tvar:

```
Histogram <identifikátor>("<jméno>",<od>,<krok>,<kolik>);
```

Význam jednotlivých parametrů je patrný z obrázku 7.4.



Obrázek 7.4: Parametry histogramu

Zápis do histogramu se provádí příkazem:

```
<identifikátor histogramu>(<výraz-hodnota>);
```

Výstup histogramu do standardního výstupního souboru provede metoda `Output`. Tiskne tabulku četností a statistiku vstupních hodnot (tj. minimální vstupní hodnotu, maximální vstupní hodnotu, počet vstupních hodnot, průměrnou hodnotu a směrodatnou odchylku).

Příklad:

Histogram `H` sledující četnost hodnot ve 100 intervalech od nuly s krokem 0.1 :

```
Histogram  H("Histogram1",0,0.1,100);
double  x;
  ...
  H(x);           // záznam hodnoty proměnné x
  ...
  H.Output();     // výstup histogramu
```

Statistiky

Objekty třídy `Stat` uchovávají tyto hodnoty:

- součet vstupních hodnot
- součet čtverců vstupních hodnot
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet zaznamenaných hodnot

Metoda `Output` vytiskne tyto hodnoty a navíc průměrnou hodnotu a směrodatnou odchylku.

Příklad:

Testujeme střední hodnotu generátoru exponenciálního rozložení:

```
Stat  TestStat("St1");
  ...
  for(int i=0; i<10000; i++)
    TestStat(Exponential(10));  // záznam hodnoty
  TestStat.Output();            // tisk statistiky
```

Časové statistiky

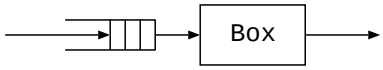
Objekty třídy `TStat` sledují časový průběh vstupní veličiny. Používají se k výpočtu průměrné hodnoty vstupu za určitý časový interval. Objekty třídy `TStat` uchovávají tyto hodnoty:

- sumu součinů vstupní hodnoty a časového intervalu
- sumu součinů čtverce vstupní hodnoty a časového intervalu
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet vstupních hodnot
- počáteční čas

Metoda `Output` tiskne kromě uložených hodnot také průměrnou hodnotu vstupu za čas od inicializace statistiky metodou `Clear` do okamžiku volání metody `Output`.

Příklad:

```
TStat  S("TStat1");
  ...
  S(x);        // záznam hodnoty v čase Time
  ...
  S.Output();  // výstup statistiky
```



Obrázek 7.5: Obslužné středisko Box

7.5.7 Příklad diskrétního modelu

Uvažujme jedno obslužné středisko se vstupní frontou, kterým procházejí zákazníci:

```
//  model  MODEL1

#include "simlib.h"

//  deklarace  globálních  objektů
Facility  Box("Linka");
Histogram Tabulka("Tabulka",0,50,10);

class Zakaznik : public Process { // třída zákazníků
  double Prichod;                 // atribut každého zákazníka
  void Behavior() {               // popis chování zákazníka
    Prichod = Time;               // čas příchodu zákazníka
    Seize(Box);                   // obsazení zařízení Box
    Wait(10);                     // obsluha
    Release(Box);                 // uvolnění
    Tabulka(Time-Prichod);        // doba obsluhy a čekání
  }
};

class Generator : public Event {  // generátor zákazníků
  void Behavior() {               // popis chování generátoru
    new Zakaznik->Activate();     // nový zákazník v čase Time
    Activate(Time+Exponential(1e3/150)); // interval mezi příchody
  }
};

//  popis  experimentu
int main()
{
  Print("***** MODEL1 *****\n");
  Init(0,1000);              // inicializace experimentu
  new Generator->Activate(); // generátor zákazníků, aktivace
  Run();                     // simulace
  Box.Output();              // tisk výsledků
  Tabulka.Output();
  return 0;
}
```

Na začátku popisu modelu musíme použít příkaz `#include`, který definuje rozhraní simulační knihovny. Dále následují deklarace globálních objektů modelu, v tomto příkladu je deklarováno zařízení `Box` a histogram `Tabulka`.

Následuje definice třídy zákazníků, kteří mají chování popsáno v metodě `Behavior`. Každý zákazník má atribut `Prichod`, kterým je doba jeho příchodu do modelovaného systému. Zákazník obsadí zařízení `Box` na dobu 10 časových jednotek (není důležité, jde-li o

hodiny či sekundy) a potom zařízení uvolní. Je zajištěno, že v případě již obsazeného zařízení bude zákazník čekat ve frontě, která se vytvoří u zařízení.

Po uvolnění zařízení se do histogramu `Tabulka` poznamená doba, strávená zákazníkem v systému (doba obsluhy plus doba strávená čekáním ve frontě u zařízení). Potom zákazník opouští námi sledovaný systém, a proto je po ukončení procesu automaticky zrušen.

Vytváření zákazníků je realizováno objektem třídy `Generator`, který periodicky se opakující událostí modeluje příchody zákazníků do systému tak, že vytváří nové zákazníky a aktivuje je.

Popis experimentu je součástí funkce `main`. Je inicializován model a nastavena doba simulace funkcí `Init` od času nula do 1000. Potom je zajištěno vytvoření generátoru příchodů zákazníků do modelu. Po inicializaci spustíme vlastní simulaci voláním funkce `Run`. Po ukončení experimentu se vytisknou informace, získané v histogramu `Tabulka`.

7.6 Spojitá simulace

Spojité chování modelu popisujeme v SIMLIB propojením objektů, které reprezentují integrátory, stavové bloky a různé nelinearity. Propojení objektů se realizuje při jejich vytváření. Konstruktor dostává jako první parametr odkaz na vstupní objekt. Tento odkaz se používá při vyhodnocování objektu. Každý objekt má definovanou metodu Value, která vrací hodnotu objektu. Pokud je k výpočtu hodnoty objektu zapotřebí hodnota vstupu, je objekt na vstupu požádán o svou hodnotu opět metodou Value. Takto proběhne výpočet všech potřebných hodnot objektů. V případě, že vznikne cyklický odkaz (rychlá smyčka), může být detekována.

Pro zvýšení efektivity výpočtu si některé objekty po vyhodnocení pamatují výslednou hodnotu, aby se při několika požadavcích na vyhodnocení v témže modelovém čase nemusely opakovat tytéž výpočty.

Reprezentace výrazu, který je obvykle na vstupu bloku, se dynamicky vytváří při volání konstruktoru. Této vlastnosti bylo dosaženo přetížením obvyklých aritmetických operátorů tak, aby při operandech typu blok dynamicky vytvořily odpovídající grafovou strukturu výrazu.

Příklad:

Uvažujme kmitavý článek:

```
class TEST : public aContiBlock {
  Integrator i1;
  Integrator i2;
public:
  TEST() : i1(-i2,1), i2(i1*0.5) {}
  double Value() { return i2.Value(); }
};
```

7.6.1 Standardní třídy pro spojitou simulaci

Třída `aContiBlock` definuje obecný blok spojitého modelu s operací vyhodnocení metodou `Value`. Všechny třídy spojitých bloků jsou odvozeny z této třídy a definují metodu `Value` pro vyhodnocení, metodu `Init` pro inicializaci a konstruktor pro definici propojení objektů. SIMLIB obsahuje třídy pro základní aritmetické operace (+,−,∗,/) a některé funkce

(Abs,Sin,Log,...). Uživatel SIMLIB může též definovat libovolné vlastní bloky, které jsou rovnocenné standardním blokům.

Třída Integrator

Třída `Integrator` slouží k implementaci integračního mechanismu spojité simulace. Integrátor má definovány tři základní operace:

- numerickou integraci
- nastavení počáteční podmínky (inicializaci)
- nastavení hodnoty (skokovou změnu stavu)

Numerická integrace vstupní hodnoty je nejdůležitější operací integrátoru, provádí se automaticky v průběhu simulace. Nastavení počáteční hodnoty lze provést více způsoby. Zadání druhého parametru konstruktoru objektu je nejčastější případ.

```
Integrator <identifikátor>(<obj-výraz>,<číselný výraz>);
```

Při inicializaci modelu v inicializační části popisu experimentu je použitelná metoda `Init`.

```
<identifikátor>.Init(<číselný výraz>);
```

Zadanou počáteční hodnotu si integrátor pamatuje a nastaví ji při startu simulace ve funkci `Run` automaticky. Nastavení hodnoty integrátoru je proveditelné při běhu simulace buď přiřazovacím příkazem, nebo metodou `Set`.

```
<identifikátor> = <číselný výraz>;
<identifikátor>.Set(<číselný výraz>);
```

Konstruktor má jako první parametr odkaz na objektový výraz — vstup. Lze zadat volitelný druhý parametr s počáteční hodnotou integrátoru. Deklarace

```
Integrator <identifikátor integrátoru>(<objekt-výraz>);
```

vytvoří integrátor se vstupem zadaným objektovým výrazem s implicitně nulovou počáteční hodnotou. Získání hodnoty integrátoru provedeme voláním metody Value příslušného integrátoru:

```
x = <identifikátor integrátoru>.Value();
```

Nelinearity

Pro modelování nelineárních bloků jsou v SIMLIB definovány standardní třídy, které jsou podrobněji popsány v referenční příručce. Nelineární bloky je nutné deklarovat, formát deklarace má tvar:

```
<třída bloku> <identifikátor>(<vstup>,<parametry>);
```

Typickým příkladem je blok omezení (třída Lim):

```
Lim  omez(x+y,-1,1);
```

Výstupem objektu omez je součet hodnot objektů x a y omezený na interval mezi -1 a 1. Blok omez je použitelný jako vstup jiného objektu.

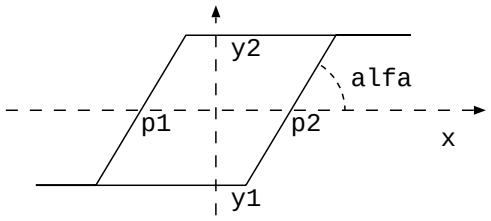
Stavové bloky

Třída Status

Třída `Status` popisuje vlastnosti stavových proměnných. Je to bázová třída pro všechny třídy objektů s vnitřním stavem, kromě třídy `Integrator`. Jako příklad si uvedeme pouze relé a hysterezi, ostatní nelineární bloky jsou stručně popsány v referenční příručce a jejich použití je stejné jako u uvedených objektů.

Třída Hyst

Objekty třídy `Hyst` mají charakteristiku podle obrázku 7.6.



Obrázek 7.6: Charakteristika hystereze

Konstruktor

```
Hyst(Input x, double p1, double p2,
              double y1, double y2, double tga);
```

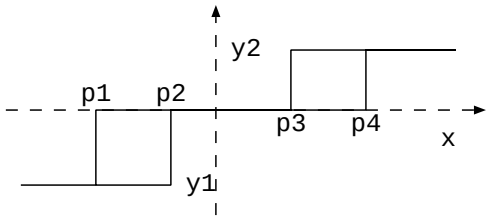
vytvoří objekt se zadaným vstupem x a s příslušnými parametry (viz obrázek 7.6). Parametr `tga` je tangentou úhlu alfa. Hodnotu výstupu tohoto objektu získáme voláním metody `Value`.

Příklad:

```
Hyst H(x,-1,1,-5,5,3.5);
```

Třída Relay

Třída `Relay` popisuje objekty s reléovou charakteristikou:



Obrázek 7.7: Charakteristika relé

Konstruktor

```
Relay(Input i, double p1, double p2, double p3, double p4,
                double y1, double y2);
```

vytvoří objekt se zadanými parametry.

Poznámka: U objektů s nespojitými charakteristikami (relé, hystereze,...) nastává při simulaci problém 'dokročení' numerické integrace na zlom charakteristiky (například do okamžiku, kdy relé sepne). Pro řešení tohoto problému[4] se používá metod, uvedených v odstavci o kombinované simulaci.

7.6.2 Příklad spojitého modelu

Jako příklad spojité simulace budeme uvažovat systém kola automobilu. Experiment bude sledovat odezvu kola na jednotkový skok. Rovnice, popisující tlumené kmitání kola:

$$M\ddot{x} + D\dot{x} + kx = F(t)$$

kde
v je rychlost pohybu kola,
y je výchylka kola z klidové polohy,
$F(t)$ je vstupní budicí funkce (např. jednotkový skok) a
k, D, M jsou konstantní parametry systému kola
Rovnici převedeme na soustavu diferenciálních rovnic prvního řádu

$$\dot{v} = (F - Dv - ky)/M$$

$$\dot{y} = v$$

a potom můžeme přímo psát simulační program:

```
// model KOLO.CPP - Model tlumení kola automobilu v C++

#include "simlib.h"

const double F = 1.0;

class Kolo {
  Graph G;              // výstup polohy kola
  Integrator v,y;       // stav systému kola
public:
  Kolo(Input F, double M, double D, double k):
     G("Výchylka",y,0.01),
     v((F-D*v-k*y)/M),
     y(v) {}
};

Kolo k1(F, 2, 5.656, 400);

int main() {                  // popis experimentu
  Print(" Model tlumení kola automobilu v C++ \n");
  OpenOutputFile("kolo.out");
  Init(0,1.5);                // inicializace parametrů experimentu
```

[4] třída Realy v SIMLIB skutečně umí řešit tento problém

```
  SetStep(1e-3,0.1);          // krok integrace
  SetAccuracy(0.001);         // max. povolená rel. chyba integrace
  Run();                      // simulace
  Print(" Konec simulace \n");
  return 0;
}
```

Spojitý model popisujeme propojením funkčních bloků — objektů modelu. Každý objekt je inicializován tak, že prvním parametrem jeho konstruktoru je jeho vstup. Na místě vstupu může být výraz, ve kterém lze použít objekty (jako proměnné) nebo číselné hodnoty (jako konstanty).

Výstup informací o chování modelu probíhá prostřednictvím objektu třídy Graph. Tato třída zabezpečuje rovnoměrné vzorkování vstupu objektu (v našem případě s periodou 0.01) a výstup do výstupního souboru, který lze prohlížet výstupním editorem.

Celý model je tvořen jedním objektem třídy `Kolo` (globální objekt `k1`). Při vytváření objektu musíme zadat jeho vstup a parametry. Řízení experimentu ve funkci `main` zajišťuje otevření výstupního souboru, inicializaci pro modelový čas od nuly do 1.5 sekundy, nastavení povoleného rozsahu kroku numerické integrace (`SetStep`) a nastavení požadované přesnosti numerické integrace (`SetAccuracy`). Vlastní simulace proběhne v rámci volání funkce `Run`.

7.7 Kombinovaná simulace

Kombinovaná simulace předpokládá použití spojitého i diskrétního přístupu, přináší však navíc některé problémy, spojené se vzájemnou interakcí spojité a diskrétní části modelu. Změny, způsobené diskrétními událostmi ve spojité části modelu, nepřinášejí téměř žádné problémy. Po skokové změně stavu spojité části modelu je zapotřebí pouze znovu inicializovat integrační metodu a je možné pokračovat v simulaci. Složitější situace nastává při potřebě vyvolat diskrétní událost při dosažení určitého stavu spojité části modelu. Řešení tohoto problému je náplní dalších odstavců.

7.7.1 Stavové podmínky a stavové události

Reakce na změny ve spojité části modelu jsou popsány formou stavových podmínek a stavových událostí. Stavová podmínka může být specifikována například booleovským výrazem. Akci, která je podmíněna změnou pravdivostní hodnoty stavové podmínky, nazveme stavová událost. Příkladem může být událost, která má nastat při překročení nastavené maximální teploty místnosti v modelu automaticky řízeného vytápění domu.

V C++ lze stavové podmínky implementovat třídami, které definují chování objektů - bloků citlivých na změnu vstupu. Vstupem takového bloku může být booleovský výraz. Pro zjištění času změny vstupní hodnoty takových podmínek lze proto použít pouze metodu půlení intervalu, tj. zkracování kroku integrace na polovinu. Existují i jiné metody, například metoda Regula-Falsi, ty však vyžadují spojitý vstup podmínky.

Můžeme také požadovat, aby stavová podmínka byla citlivá pouze na některé změny pravdivostní hodnoty vstupu (například pouze na změnu FALSE na TRUE, případně na překročení mezní hodnoty směrem nahoru). Protože C++ nedefinuje speciální Booleovský typ, používá se místo něj typ int. Pravdivostní hodnota TRUE potom odpovídá nenulové hodnotě typu int, pravdivostní hodnota FALSE odpovídá nulové hodnotě.

Při numerické integraci, kdy výpočet probíhá po krocích, nemusí dojít k detekci některých stavových událostí. Tato situace nastane v případě, že krok integrace je příliš dlouhý a dojde při něm k 'překročení' několika změn stavových podmínek. Podobný problém může nastat v

důsledku nepřesnosti numerické integrace, kdy při nevhodně zvolené podmínce nemusí dojít k její změně a tím k vyvolání požadované události.

Třída BoolCondition

Bázová třída BoolCondition popisuje pouze chování, potřebné pro detekci změn vstupní podmínky. Uživatel musí tuto třídu zdědit a doplnit metodu, popisující vstup podmínky a metodu, popisující akci při změně podmínky. Vstup podmínky popisuje metoda Test, která vrací pravdivostní hodnotu. Pro detekci změny pravdivostní hodnoty používáme metodu půlení intervalu. Při změně stavu podmínky se integrační krok zkracuje tak dlouho, až dosáhne své minimální povolené hodnoty (proměnná MinStep). Tím zajistíme 'dokročení' v čase, který se nejvíce blíží skutečnému okamžiku změny podmínky. Teprve potom nastane reakce na tuto změnu vyvoláním metody Action. Metoda Mode umožňuje nastavení citlivosti podmínky na různé změny pravdivostní hodnoty vstupu podmínky.

Příklad:

```
class MyCondition : BoolCondition {
    Input inp;
    int Test()    { return inp.Value()<0; }
    void Action() { Print("změna znaménka z + na -"); }
  public:
    MyCondition(Input i) : inp(i) { Mode(DetectUP); }
};

Integrator x(vstup);
MyCondition  Test(x);
```

Příklad popisuje podmínku, reagující na změnu znaménka vstupní hodnoty z plus na minus. Na vstupu objektu Test je integrátor x.

7.7.2 Příklad kombinovaného modelu

Jako příklad kombinovaného modelu použijeme model skákajícího míčku. Spojité chování míčku odpovídá volnému pádu, pro detekci dopadu míčku používáme stavovou podmínku. Stavová událost (dopad míčku) vyvolá skokovou změnu rychlosti míčku.

```
//  model  MICEK.CPP -  skákající  míček

#include "simlib.h"

const double g = 9.81;       // gravitační zrychlení

class Micek : BoolCondition {
  Graph G;
  Integrator v,y;
  int Test() { return y.Value()<0; }
  void Action() {
      Print(" Odraz míčku v čase t=%g \n",Time);
      G();
      v = -0.8 * v.Value();
      y = 0;  // nutné pro detekci následujícího dopadu
```

```
    }
  }
public:
  Micek(double position) :
     G("Výška míčku",y,0.05),
     y(v,position), v(-g)  {
    Mode(DetectUP);
  }
};

Micek  micek(1);             // vytvoření objektu micek

int main() {                 // popis experimentu
  Print(" Model skákajícího míčku v C++ \n");
  OpenOutputFile("micek.out");
  Init(0,5);                 // inicializace experimentu
  SetStep(1e-10,0.5);        // krok integrace
  SetAccuracy(1e-5,0.001);   // max. povolená chyba integrace
  Run();                     // simulace
  Print(" Konec simulace \n");
  return 0;
}
```

Chování míčku je popsáno jako volný pád. Integrátor v integruje tíhové zrychlení g a jeho hodnota je rovna rychlosti míčku. Integrátor y integruje rychlost a jeho hodnota představuje výšku míčku nad zemí.

Pro detekci okamžiku dopadu míčku na zem ($y = 0$) je použito stavové podmínky `y.Value()<0` v metodě `Test`. Tato metoda je volána systémem řízení simulace při každém kroku numerické integrace. V případě změny hodnoty podmínky se krok zkracuje tak, abychom okamžik dopadu míčku určili s maximální přesností (přesnost určení doby dopadu je dána minimální délkou kroku, tj. hodnotou proměnné `MinStep`). V okamžiku dopadu míčku se provede akce, popsaná v metodě Action, tj. obrácení a zmenšení vektoru rychlosti. Tímto způsobem modelujeme ztrátu energie při dopadu míčku.

Řízení experimentu popisuje funkce `main`. Po volání Init následuje nastavení povoleného rozsahu kroku integrace (SetStep) a určení požadované přesnosti integrace (SetAccuracy). Běh simulace se odstartuje voláním funkce Run. Výstup se řeší podobně, jako u spojité simulace, je zde však nutné zajistit výstup též v okamžicích dopadu míčku na zem.

7.8 SIMLIB-3D rozšíření

Pro usnadnění popisu modelů, které vyžadují popis vektorovými diferenciálními rovnicemi byla SIMLIB doplněna o 3D abstrakce.

Prostorové modely jsou popsatelné vektorovými dif. rovnicemi. Například pohyb hmotného bodu v gravitačním poli je možné popsat rovnicí:

$$\ddot{\vec{r}} = \frac{\vec{F}}{m}$$

kde: $\vec{r}$ je vektor — pozice bodu
$\vec{F}$ je vektor síly působící na hmotný bod
m je hmotnost bodu
SIMLIB dovoluje popis tohoto systému jak skalárními prostředky, tak vektorově.

Vektorový popis je výrazně kratší:

```
class MassPoint {  // model hmotného bodu
  const double m;  // hmotnost
  Integrator3D v;  // rychlost
  Integrator3D r;  // pozice
 public:
  MassPoint(Input3D F, Value3D ini_pos) :  // konstruktor:
     m(1000),       // hmotnost je konstantní
     v(F/m),        // rychlost je integrál zrychlení
     r(v, ini_pos)  // pozice je integrál rychlosti
     {}
};
```

Z uvedeného příkladu je zřejmé, že třírozměrný popis problému je stejně jednoduchý jako popis skalární (operace jsou vektorové a v roli operandů jsou vektory místo skalárů).

7.8.1 Hierarchie tříd 3D

```
aBlock
  aContiBlock3D       - bázová třída
    Constant3D        - konstanta
    Expression3D      - blokový výraz
    Integrator3D      - vektorový integrátor
    Function3D        - obecná vektorová funkce
       _Add3D, _Mul3D - (skryté) třídy pro operátory
    Parameter3D       - parametr modelu

Value3D               - hodnota
Input3D               - odkaz na blok
```

Třída `Value3D` definuje vektorovou hodnotu a má tři složky (x,y,z) typu `double`. Používá se na předávání a uchovávání vektorových hodnot.

Třída `Input3D` definuje odkaz na objekt–blok. Použití odkazu v objektovém výrazu je transparentní, tj. blok na jehož vstupu je uveden tento odkaz si poznamená cíl tohoto odkazu a nikoli odkaz samotný (pozdější změny odkazu již nic neovlivní). Vpřípadě, že nám toto chování nevyhovuje, je možné použít třídu `Expression3D`, která se chová jako blok–identita.

Třída Integrator3D obsahuje tři skalární integrátory, které jsou napojeny na speciální objekty pro transformaci rozhraní 3D/skalární. To je možné proto, že integrace je lineární operátor.

Ostatní třídy definují konstanty, parametry modelu a funkce podobně jako jejich skalární ekvivalenty.

7.8.2 Blokové výrazy

Operace $+ - */$ jsou implementovány jako operátory, které dynamicky vytvoří příslušný objekt a zapojí jeho vstupy. Situace je stejná jako u běžných skalárních operátorů.

Přehled definovaných operátorů:

```
// binární operátory:
Input3D operator + (Input3D a, Input3D b);  // součet vektorů
```

```
Input3D operator - (Input3D a, Input3D b);  // rozdíl
Input3D operator * (Input3D a, Input3D b);  // součin
Input3D operator * (Input3D a, Input b);    // vektor * skalár
Input3D operator * (Input a, Input3D b);    // skalár * vektor
Input3D operator / (Input3D a, Input b);    // vektor / skalár

// unární operátory:
Input3D operator - (Input3D a);             // unární -
```

Funkce v blokových výrazech 3D jsou definovány pouze pro základní operace s vektory:

```
// funkce:
Input Abs(Input3D x);                       // absolutní hodnota vektoru
Input3D UnitVector(Input3D x);              // jednotkový vektor
Input ScalarMultiply(Input3D x, Input3D y); // skalární doučin x.y

Input Xpart(Input3D a); // složka x vektoru
Input Ypart(Input3D a); // složka y vektoru
Input Ypart(Input3D a); // složka z vektoru
```

7.8.3 Příklad

Uvažujme systém Země–Měsíc a družici, která má vhodně zvolenou počáteční pozici a rychlost. Takový systém můžeme popsat takto:

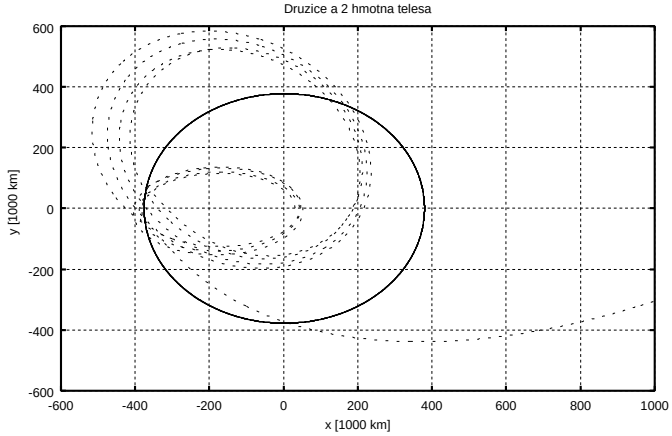
```
// druzice.cc -- model Země--Měsíc--družice
#include "simlib.h"
#include "simlib3D.h"

typedef Value3D Position, Speed, Force;

const double   gravity_constant = 6.67e-11; // gravitační konstanta
const double   m0 = 1000;           // hmotnost družice
const double   m1 = 5.983e24;       // hmotnost Země
const double   m2 = 7.374e22;       // hmotnost Měsíce
const Position p0(36.0e6, 0, 0);    // poloha družice
const Position p2(384.405e6, 0, 0); // poloha Měsíce
const Speed    v0(0, 4.5e3, 0);     // počáteční rychlost družice
const Speed    v2(0, 1022.6, 0);    // oběžná rychlost Měsíce

Constant3D Zero(0,0,0); // pomocný objekt

struct MassPoint {
  double m;              // hmotnost
  Expression3D inforce;  // vstupní síla
  Integrator3D v;        // rychlost
  Integrator3D p;        // pozice
  MassPoint(const double mass, Position p0, Speed v0=Speed(0,0,0)) :
    m(mass),
    inforce(Zero),
    v(inforce/m, v0),
    p(v,p0) {}
   void SetInput(Input3D i) { inforce.SetInput(i); }
};
```

```
struct MyWorld {  // digitální svět
  enum { MAX=10 };
  MassPoint *m[MAX];
  unsigned n;
  MyWorld();
};

MyWorld *w;  // vznikne až později :-)

// gravitační síla pusobící na hmotný bod p
class GravityForce : public aContiBlock3D {
  MassPoint *p;
  MyWorld *w;
 public:
  GravityForce(MassPoint *_p, MyWorld *_w) : p(_p), w(_w) {}
  Force Value() {
    Force f(0,0,0);  // gravitační síla
    for(int i=0; i < w->n; i++) {
      MassPoint *m = w->m[i];
      if (m == p) continue;
      Value3D distance = m->p.Value() - p->p.Value();
      double d = abs(distance);  // vzdálenost
      f = f + (distance * gravity_constant * p->m * m->m / (d*d*d)) ;
    }
    return f; // součet všech gravitačních sil
  }
};

typedef MassPoint Planet, Satelite; // pohyblivé planety

MyWorld::MyWorld() {  // --- vytvoříme digitální svět
  n=0;
  m[n++] = new Planet(m1,Position(0,0,0),Speed(0,0,0));
  m[n++] = new Planet(m2,p2,v2);
  m[n++] = new Satelite(m0,p0,v0);
  for(int i=0; i<n; i++)  // --- zapneme silové působení
    m[i]->SetInput(new GravityForce(m[i],this));
}
```

Třída `MassPoint` definuje model hmotného bodu. Vstupem tohoto modelu je síla, která způsobí jeho pohyb. Protože model vytváříme postupně, je implicitně tato síla nulová a nastaví se až po vytvoření všech hmotných bodů v systému (viz konstruktor třídy `MyWorld`). Celý model je tvořen třídou `MyWorld`, která obsahuje seznam všech hmotných bodů.

Třída `GravityForce` modeluje gravitační sílu a je pro každý hmotný bod definována jako součet všech gravitačních sil působících na hmotný bod. Gravitační síla se počítá se podle gravitačního zákona.

Výsledná dráha družice a Měsíce je uvedena na obrázku 7.8.



Obrázek 7.8: Družice v gravitačním poli

Kapitola 8

Simulace číslicových systémů

V *kap. 2* jsme poznali klasifikaci obecných systémů na systémy spojité, diskrétní a kombinované. Klasifikaci můžeme ale provést i s ohledem na charakter vlastností a vztahů, které nás na systému zajímají a pro které vytváříme model. Potom můžeme hovořit o systémech, resp. modelech technických, biologických, ekologických, ekonomických apod. Číslicové systémy můžeme zařadit mezi technické systémy. Budeme je neformálně chápat jako systémy pracující s číslicovou informací.

Pro simulaci výše uvedených speciálních tříd systémů se v praxi zpravidla využívají univerzální simulační jazyky a systémy, případně se vytvářejí specializované simulační prostředky, které však bývají specializované pouze s ohledem na uživatelské rozhraní tak, aby uživatel-odborník mohl při modelování a simulaci pracovat s pojmy a způsobem, na které je ze své odborné praxe zvyklý.

V případě číslicových systémů je situace poněkud odlišná. Jak uvidíme v *kap. 8.1* lze modelovat číslicový systém na různých úrovních. Ve většině případů se simulační modely vytvářejí použitím specializovaných simulačních prostředků, které se vyznačují nejen specializovaným uživatelským rozhraním, ale i použitím speciálních simulačních technik.

Simulace je v současné době základním prostředkem analýzy při návrhu číslicových systémů. Praxe, kdy spočívalo těžiště ověřování návrhu v experimentování s prototypem navrhovaného systému, resp. jeho dílčích částí, je nyní naprosto nepřijatelná. Pokrok v technologii, zejména integrovaných obvodů, a široké používání tzv. aplikačně specifických integrovaných obvodů (ASIC) umožnil integrovat na jednom čipu složité funkce, které byly dříve realizovány na řadě desek osazených standardními obvody malé a střední integrace, mohly být snadno ověřovány na prototypu a změny prováděny bez větších problémů. Za stávající situace je třeba snížit na minimum riziko nesprávného návrhu. Zatím nejvýhodnějším prostředkem pro tyto účely je simulace. Proto nastal v 70. a 80. letech bouřlivý rozvoj simulačních prostředků a technik a vznikla řada simulačních jazyků a systémů orientovaných na simulaci číslicových systémů.

V závislosti na etapě návrhu, ve které se simulace používá, můžeme rozlišit tyto základní oblasti použití:

- hodnocení alternativ návrhu,
- ověření funkční správnosti,
- ověření časového chování (časování),
- ověření kvality diagnostických testů (simulace poruch),

- ověření některých vlastností konstrukčního návrhu.

V této kapitole se seznámíme s nejdůležitějšími pojmy a technikami používanými při simulaci číslicových systémů. V závěru kapitoly jsou velice stručně charakterizovány logický simulátor OrCAD/VST jako představitel moderního poměrně jednoduchého simulátoru pro osobní počítače a simulační jazyk VHDL, který se stává standardem v oblasti simulace číslicových obvodů.

8.1 Úrovně popisu číslicových systémů

Návrh číslicového systému, jeho popis, vytváření modelu a simulace, může probíhat na různých *úrovních*, které lze hierarchicky uspořádat. V *kap. 2* jsme viděli, že každý systém můžeme charakterizovat na základě jeho struktury (prvky a vazby mezi nimi) nebo jeho chování. S ohledem na tato dvě hlediska lze zavést čistě strukturní *hierachii úrovní* nebo naopak hierarchii s ohledem na chování. V praxi se však často obě hlediska prolínají. V procesu návrhu totiž na základě požadavků specifikujeme nejprve chování systému, ale postupně přidáváme strukturní informaci. Potřebujeme tedy specifikovat jak strukturu, tak chování. V souladu s procesem návrhu budeme rozlišovat tyto základní úrovně:

- chování systému,
- funkční(architektury)
- strukturní(logická),
- fyzická

Přechody mezi těmito úrovněmi v procesu návrhu se nazývají *etapy návrhu*. Hovoříme o návrhu systémovém, logickém a fyzickém. Vymezení výše uvedených čtyř základních úrovní však nesmíme chápat jako striktní v tom smyslu, že se návrh popisuje a ověřuje vždy pouze a přesně na těchto úrovních. Je zřejmé, že jinak bude probíhat návrh počítače, mikroprocesoru nebo desky se standardními integrovanými obvody, odlišné bude i využití simulace v těchto návrzích. Přechod mezi uvedenými úrovněmi obecně pro složité systémy představuje transformaci, která zahrnuje řadu dílčích kroků a tedy i řadu dílčích úrovní, na kterých lze systém nebo jeho části popsat a simulovat. Naopak jednodušší návrhy mohou zahrnovat např. pouze popis na strukturní a fyzické úrovni.

Uvažujme příklad návrhu počítače. Na základě požadavků je definováno jeho chování. V procesu *systémového návrhu* je provedeno rozčlenění na základní podsystémy. Úloha simulace by zde spočívala především v hodnocení různých variant řešení. Vzhledem k tomu, že číslicový systém bývá na této úrovni často modelován jako systém hromadné obsluhy, lze k hodnocení využít běžné jazyky a systémy pro diskrétní simulaci. Vznikly však i speciální jazyky vhodné pro použití v etapě systémového návrhu. Pro strukturní popis počítačů například vznikl jazyk *PMS* (*Processors, Memories, Switches*), jehož hlavním cílem bylo poskytnout vhodnou notaci pro popis a srovnávání různých výpočetních systémů. Ve skutečnosti tento jazyk sloužil výhradně k popisným a dokumentačním účelům, k výstavbě simulačních modelů přímo nesloužil. Příklad použití je uveden v [6].

Na přelomu 70. a 80. let vznikla řada jazyků (např. *LALSDII*, *ADLIB*, *AIDE*), které se někdy označují jako jazyky úrovně architektury, protože jsou vhodné pro popis systému na úrovni architektury. Tyto jazyky mají zpravidla prostředky pro popis struktury i chování. Poskytují uživateli řadu údajových typů, případně předdeklarovaných strukturních prvků, které usnadňují vytváření modelů. Prostředky k popisu chování jsou často orientovány na

procesy. Tyto jazyky mívají zabudovány specializované prostředky pro hodnocení výkonnosti.

Výsledkem systémového návrhu v našem příkladě by byla architektura počítače. Pokud se omezíme na návrh technických prostředků, byly by stanoveny základní podsystémy, jejich chování a vazby mezi nimi. Dále by již mohl probíhat *logický návrh* podsystémů samostatně, muselo by být pouze zajištěno požadované chování. V závislosti na složitosti podsystému by mohl návrh zahrnovat několik dílčích kroků. Např. návrh procesoru by opět mohl sestávat z návrhu jeho architektury, tentokrát specifikované např. množinou registrů, operací arimeticko-logické jednotky, algoritmem činnosti řadiče apod. Úroveň číslicového systému, na níž je chování systému specifikováno především definováním přenosů mezi registry, se označuje jako *úroveň meziregistrových přenosů*. Jde o významnou úroveň z hlediska návrhu i simulace, proto bývá v klasifikaci úrovní často uváděna samostatně. Přestože je známa poměrně dlouho, nebyla dosud přesně definována. Budeme ji chápat jako úroveň, na které jsou základními prvky především registry a funkční přenosy mezi nimi. Registry tvoří hlavní sekvenční prvky, jako další se uvádějí čítače, pole registrů, různé typy pamětí apod. Kombinační prvky realizují funkční transformace při přenosech mezi sekvenčními prvky. Jako primitivní kombinační prvky se uvádějí nejčastěji multiplexory, demultiplexory, kodéry, dekodéry, obvody realizující aritmetické a logické operace apod. Jedním z charakteristických rysů úrovně meziregistrových přenosů je rozlišování řídicí a datové informace, resp. signálu.

Pro vyjádření funkce prvků a jejich propojení se na úrovni meziregistrových přenosů používají nejčastěji různá bloková schémata, vývojové diagramy a programovací jazyky. Dále se zaměříme výhradně na použití programovacích jazyků, protože mají ve srovnání s jinými způsoby popisu řadu výhod, především slouží jako simulační jazyky. Zpravidla jde o problémově orientované programovací jazyky, označované zkratkou *RTL (Register Transfer Languages)*, které jsou podmnožinou obecnějších jazyků pro modelování technických prostředků (*HDL — Hardware Description Languages*). Protože jde z hlediska modelování a simulace o významnou skupinu jazyků, budeme se jimi podrobněji zabývat v *kap. 8.7.*

Vraťme se nyní k našemu příkladu. Po případných korekcích by návrh pokračoval dalším zpřesňováním struktury (např. návrh řadiče jako konečného automatu), až bychom dostali strukturu tvořenou prvky, které máme k dispozici. Zde by opět byla využita simulace pro ověřování návrhu a po návrhu diagnostických testů i k posouzení jejich kvality.

Na této strukturní úrovni je systém plně definován popisem struktury, neboť chování prvků je pevně dáno. Typickým prostředkem popisu je zde logické nebo funkční schéma. Může být vyjádřeno textově(ve tvaru seznamu prvků a spojů - tzv. *netlist*) nebo graficky. Protože jde o diskrétní systém (změny signálu jsou chápány na dané úrovni abstrakce jako skokové), bylo by možné použít pro simulaci některý univerzální simulační jazyk. Popis schématu by však v tomto případě mohl být značně komplikovaný, stejně jako popis chování prvků. Především však časová náročnost simulace vyžaduje použití speciálních technik. Proto se výhradně používají problémově orientované simulační jazyky a systémy. Simulace využívaná v etapě logického návrhu se zpravidla označuje jako *logická simulace*.

Poznámka: V případě, že jsou primitivními prvky struktury základní logické členy (hradla), hovoří se často o *simulaci na úrovni hradel*. Simulátory úrovně hradel byly běžné dříve, dnes má většina simulátorů k dispozici i modely složitějších prvků.

Dosud jsme předpokládali, že návrh probíhá metodou shora-dolů, tj. postupným zjemňováním struktury a že skončí na úrovni prvků, které máme k dispozici. Mlčky jsme předpokládali, že jde přinejmenším o základní logické členy. Avšak i jejich návrh musel někdy proběhnout a při něm návrhář již nemusel vystačit s chápáním signálů jako diskrétních, nýbrž v určité fázi návrhu musel tuto abstrakci opustit a pracovat se signály jako spojitými,

tedy i systém chápat jako spojitý. V takovém případě hovoříme o *elektrickém*, resp. *obvodovém návrhu* a také o *simulaci na elektrické*, resp. *obvodové úrovni*. I zde se používá k ověření návrhu simulace.

Protože systém chápeme na této úrovni jako spojitý, lze ho popsat soustavou diferenciálních a transcendentních rovnic vycházejících ze základních zákonů a vztahů pro elektrické obvody s respektováním náhradních zapojení složitějších prvků. V zásadě by bylo možné použít pro simulaci libovolný univerzální simulační systém nebo program pro spojitou simulaci. Nevýhodou tohoto přístupu by však byla potřeba sestavit příslušné rovnice tvořící matematický model. Je žádoucí, aby uživatel, v tomto případě návrhář, mohl používat prostředky, které jsou mu blízké. Popis obvodu by tedy měl odpovídat tomu, jak jej návrhář chápe, tj. propojení aktivních a pasivních prvků. Z tohoto důvodu se výhradně používají jazyky a systémy problémově orientované na simulaci na úrovni obvodů.

Tyto systémy slouží obecně k modelování elektrických obvodů a zahrnují modely nejen jednoduchých prvků jako jsou tranzistory, ale i např. operačních zesilovačů. Hovoří se zpravidla o *elektrické, obvodové* nebo někdy také *analogové simulaci*, aby se zdůraznila základní odlišnost od logické simulace, kterou je chápání signálů jako spojitých (analogových).

Simulační jazyky v tomto případě poskytují prostředky pro popis struktury, pro zadávání parametrů aktivních a pasivních prvků, definici časových průběhů zdrojů signálů, pro specifikaci sledovaných signálů a formátu jejich zobrazení apod. S rozvojem grafických interakčních zařízení, pracovních stanic a osobních počítačů se dnes výhradně používá grafický vstup, jehož jádrem je editor umožňující popsat obvod nakreslením jeho schématu. Zadávání parametrů se provádí interakčně, stejně jako řízení simulace. Také výsledky simulace se zobrazují v grafickém tvaru, podobně jak bychom je viděli na osciloskopu. Běžně lze provádět přechodovou, stejnosměrnou a frekvenční analýzu, v některých případech i toleranční, případně citlivostní analýzu. K nejpoužívanějším obvodovým simulátorům na osobních počítačích a pracovních stanicích patří simulátory PSpice a Spice.

8.2 Základní pojmy a techniky používané při modelování

V této kapitole budou vysvětleny některé základní pojmy a techniky používané při logické simulaci. Všimneme si především způsobu popisu obvodu, používaných modelů signálů, zpoždění, simulačních algoritmů a některých technik používaných k implementaci simulačních systémů.

8.2.1 Prostředky pro popis obvodu

V předchozí kapitole jsme viděli, že různé úrovně vyžadují různé prostředky pro vytváření modelu systému. Systém může být modelován z hlediska své struktury nebo chování, totéž platí pro prvky systému. Obecně proto potřebujeme mít k dispozici prostředky jak pro popis struktury, tak chování. To odpovídá používání logických schémat, blokových diagramů, funkčních tabulek, stavových diagramů automatu apod. Můžeme rozlišit dva základní přístupy:

a) Jazyk má pouze prostředky pro popis struktury Uživatel má k dispozici určitou množinu prvků s definovaným chováním. Nejčastěji jde o základní logické členy nebo konkrétní integrované obvody. Modely těchto obvodů jsou zabudovány v simulátoru a uživatel je nemá možnost měnit. Často lze však parametrizovat hodnoty některých

atributů (počet vstupů, zpoždění apod.), což umožňuje přizpůsobení konkrétním potřebám.

Moderní logické simulátory, implementované na pracovních stanicích nebo osobních počítačích, používají běžně ke tvorbě strukturního modelu grafické editory pro kreslení schémat. Modely prvků jsou přímo součástí simulátoru a chce-li uživatel použít nějaký jiný prvek, musí vytvořit jeho strukturní model použitím prvků primitivních. U starších simulátorů s textovým vstupem k tomuto účelu sloužily zpravidla makrodefinice. Princip spočíval v tom, že určitou strukturu prvků bylo možné označit jako nový typ prvku a tak s ním také pracovat. Například uživatel definuje nový prvek typu RS jako propojení dvou logických členů NAND. Registr vytvořený z těchto klopných obvodů potom již nepopisujeme jako propojení prvků NAND, nýbrž jako propojení prvků typu klopný obvod RS. Jde vlastně o hierarchii struktur.

Moderní simulátory s grafickým vstupem používají zpravidla organizované knihovny prvků, z nichž si simulátor potřebné vybírá. Pro tvorbu modelů bývá k dispozici nějaký zpravidla velice jednoduchý jazyk. Protože tvorba nových modelů je v tomto případě poměrně pracná, bývají rozsáhlé knihovny prvků přímo součástí dodávky simulačního, resp. návrhového systému. Tato koncepce je použita i u systému OrCAD/VST.

b) Jazyk má prostředky pro popis struktury i chování Tento způsob je pružnější, neboť není vázán na používání nějaké omezené množiny prvků, jejichž chování umíme simulovat. Tuto množinu si vlastně vytváří uživatel sám (nebo již má k dispozici její základ ve formě knihovny) použitím prostředků pro popis chování. Uvažujme nějaký prvek (obecně systém), jehož model vytváříme. Pokud poskytuje jazyk, který máme k dispozici, prostředky pro popis chování i struktury, záleží na naší znalosti struktury systému, jaký model vytvoříme. Často jazyk připouští v rámci jednoho modelu použít buď pouze prostředky pro popis struktury a vytvořit tak čistě strukturní model nebo naopak vytvořit čistý model chování, ve kterém abstrahujeme od konkrétní struktury. Příkladem jazyka tohoto typu je jazyk SFDL [6].

Protože v procesu návrhu návrhář postupně zpřesňuje strukturní informaci, je žádoucí, aby bylo možné prostředky pro popis struktury a chování směšovat. Tento způsob popisu se označuje jako *architektonický* (*architectural description*). Je běžný u klasických jazyků meziregistrových přenosů, jak uvidíme v *kap. 8.7*. Explicitně se u nich deklarují některé prvky modelované struktury, především registry, jiné jsou skryty v operacích transformací při přenosech mezi registry. Celkové chování je potom také určeno způsobem vyhodnocování příkazů.

Požadavek libovolného kombinování popisů struktury a chování byl uplatněn i při návrhu jazyka VHDL (viz *kap. 8.8*).

8.2.2 Model signálu

Logická simulace předpokládá, že se hodnoty signálů mění diskrétně. Použitý model signálu je určen oborem simulačních hodnot. *Simulační hodnotou* rozumíme reprezentaci hodnoty (stavu) signálu. Při modelování se zde může uplatnit různý stupeň abstrakce.

Nejjednodušším případem s nejvyšší abstrakcí je použití pouze dvou hodnot reprezentujících stav *logické nuly* a *logické jedničky*. Nejčastěji se označují 0 a 1. *Dvouhodnotových modelů* se v moderních simulátorech používá málo, protože mají některé vážné nedostatky. Především je to problém inicializace simulovaného systému. Problém spočívá v tom, že i když se signály v modelovaném obvodě po připojení napájecího napětí dostanou do stavu logické nuly nebo jedničky, nelze předem říci, do kterého. Při návrhu je potřeba tuto skutečnost respektovat

a zajistit uvedení do jednoznačně definovaného stavu. Pokud používáme dvouhodnotový simulátor, musíme na začátku simulace definovat hodnoty všech signálů, resp. využít implicitního nastavení. To nám prakticky neumožňuje odhalit chyby návrhu související s neošetřením nedefinovaného stavu. Druhým nedostatkem je obtížnost modelování některých hazardních stavů, kdy opět nejsme schopni jednoznačně určit hodnotu signálu. Naopak výhodou dvouhodnotových simulátorů je jednoduchost, nízká režie a vysoká rychlost. Proto se někdy používají tam, kde uvedené nedostatky příliš nevadí, např. v některých případech při simulaci poruch.

Vzhledem k nedostatkům dvouhodnotových simulátorů se v praxi často zavádí simulační hodnota reprezentující *neurčený* (*nedefinovaný*) *stav*. Nejčastěji se označuje X nebo U a interpretuje se jako neznámý, ale stabilní stav (0 nebo 1). Tím lze odstranit prvý výše uvedený nedostatek. Pro odstranění druhého se někdy interpretuje hodnota X obecněji jako neznámý stav, který může být nestabilní. S *tříhodnotovou simulací* se můžeme setkat u jednoduchých logických simulátorů, při simulaci na úrovni meziregistrových přenosů a při simulaci poruch.

Vzhledem ke skutečnosti, že řada technologií umožňuje uvést některé své výstupy do *stavu vysoké impedance*, používá řada simulátorů hodnotu označovanou nejčastěji Z právě k reprezentaci tohoto stavu.

Při vyšších nárocích na přesnost simulace se zavádějí další simulační hodnoty. Mezi nejužívanější patří hodnoty reprezentující *přechod z logické nuly* do *jedničky* (zpravidla R — Rise) *a naopak* (F — Fall). Tyto hodnoty mohou být interpretovány jako skutečná doba náběhu, resp. doběhu signálu, častěji však vymezují oblast, ve které k přechodu dochází, ale z nějakého důvodu, např. rozplylu zpoždění, nelze okamžik přechodu určit přesně.

Moderní logické simulátory často také umožňují modelovat různé budicí schopnosti signálů, jinými slovy, že různé signály mají různou "sílu", a pokud se v nějakém místě obvodu setkávají (montážní funkce, sběrnice), je výsledný signál určen hodnotou signálu "nejsilnějšího". Typickým případem jsou MOS obvody. Uvažujme jednoduchý příklad invertoru v techologii NMOS (*obr.* **??**). Je-li na vstupu I logická nula, je tranzistor T_1 uzavřen a na výstupu bude hodnota logické jedničky. Její "síla" je dána odporem tranzistoru T_2. Proto je žádoucí, aby odpor tranzistoru T_2 byl co nejmenší, aby se zatěžovací kapacita, která ovlivňuje dynamické vlastnosti invertoru, co nejdříve nabila. Uvažujme druhý stav. Je-li na vstupu I logická jednička, bude tranzistor T_1 otevřen, zatěžovací kapacita se se přes něj vybije a hodnota na výstupu O bude určena napětím na odporovém děliči tvořeném tranzistory T_1 a T_2. Protože na výstupu invertoru má být logická nula, musí být odpor tranzistoru T_1 několikanásobně menší, než tranzistoru T_2. Protože tento odpor určuje "sílu" signálu, bude logická nula na výstupu O "silnější" než logická jednička a pokud by měly dva takové invertory, jeden s výstupem v jedničce a druhý v nule, výstupy spojeny, byl by výsledný signál roven nule.

Proto tyto logické simulátory zavádějí jako atribut modelu signálu *sílu signálu*. Např. simulátor OrCAD/VST, se kterým se podrobněji seznámíme v *kap. 8.5* můžeme charakterizovat jako tříhodnotový $\{0, 1, X\}$ rozlišující čtyři úrovně síly signálu (stav vysoké impedance je zde vyjádřen úrovní síly signálu, nikoliv simulační hodnotou).

Dosud uvedené simulační hodnoty patří k nejpoužívanějším. Někdy se můžeme setkat u simulátorů používaných pro velmi přesnou simulaci, především s ohledem na detekci různých hazardních stavů, s dalšími simulačními hodnotami např. pro reprezentaci *statických* nebo *dynamických hazardů*.

Závěrem je třeba zdůraznit, že s růstem počtu simulačních hodnot roste přesnost simulace, zpravidla však za cenu složitějšího simulátoru, poklesu rychlosti simulace a vyšších nároků na paměť. Poměrně jednoduchá je situace u simulátorů se zabudovanými modely úrovně základních logických členů, neboť lze snadno definovat základní logické operace nejen pro hodnoty 0 a 1. Složitější je však situace při modelování na vyšší úrovni abstrakce. Uva-

žujme čtyřbitovou sčítačku se vstupy A a B, vstupem přenosu CI a výstupem součtu S a přenosu CO. Předpokládejme, že máme k dispozici simulátor poskytující prostředky pro popis chování (včetně operace aritmetického součtu). V případě dvouhodnotové simulace je situace jednoduchá. Rozšířením oboru simulačních hodnot o hodnotu X by měl simulátor zajistit správné šíření této hodnoty na výstup prvku, tj. je-li např. hodnota některého bitu vstupu B nedefinovaná nastavit na výstupech S a CO hodnotu X pouze tam, kde ji skutečně nelze jednoznačně určit. Způsob, jak by to bylo možné provést, ponecháváme na čtenáři. Zde pouze uvedeme, že vyhodnocení je v takových případech obecně značně časově náročné, a proto se často zavádějí určitá zjednodušení. V našem příkladě bychom nastavili hodnoty všech bitů S i výstupu CO na X.

8.2.3 Modely zpoždění

Otázka zpoždění úzce souvisí s otázkou oboru simulačních hodnot. Tak, jak vyjadřuje obor simulačních hodnot jistý stupeň abstrakce tvaru signálu, různá zpoždění vyjadřují různou abstrakci chování systému v čase. Podobně, jako v případě volby oboru simulačních hodnot, je i zde nutno vyjít z oblasti použití simulátoru, především má-li simulátor sloužit i pro ověření časového chování systému, tj. k odhalení případných chyb v časování.

Nejvyšším stupněm abstrakce je *nulové zpoždění*. Tento model zpoždění je vhodný pouze v případech, kdy simulátor slouží k ověření funkční správnosti bez ohledu na časování nebo při modelování synchronních systémů, kdy mohou probíhat činnosti pouze v pevných, ekvidistantních okamžicích.

Některé simulátory pracují s *jednotkovým zpožděním*. Předpokládají, že zpoždění každé operace trvá jednu časovou jednotku. U těchto simulátorů lze sice odhalit některé hazardní stavy, používají se však spíše při modelování na vyšších úrovních (např. meziregistrových přenosů), případně při simulaci poruch, jak uvidíme později. Při nulovém a jednotkovém zpoždění stačí pro modelování dvou- nebo tříhodnotová simulace.

Má-li být simulátor použitelný i pro analýzu časového chování systému, používá se přesnějších modelů zpoždění. Častým případem je použití *přiřaditelného zpoždění*, kdy lze přiřadit každé operaci, resp. prvku zpoždění. Zpravidla se navíc rozlišují různé hodnoty zpoždění pro přechod z nuly do jedničky a naopak. V případě, že jde o simulátor se zabudovanými modely, je zpoždění již přímo součástí modelu (jde-li např. o modely konkrétních integrovaných obvodů), častěji však lze modely prvků parametrizovat hodnotou zpoždění. Přiřazení zpoždění operacím je typické pro modelování na vyšší úrovni abstrakce při popisu chování, jak uvidíme u jazyka VHDL v *kap. 8.8*.

Použití přiřaditelného zpoždění umožňuje modelovat jevy, které se v reálných obvodech vyskytují, ale při modelování s jednotkovým zpožděním se neprojeví. Typickým příkladem je filtrace úzkých impulsů na vstupech vlivem setrvačného zpoždění, které se projevuje různou hodnotou zpoždění pro přechod z nuly do jedničky a naopak.

Výhodou přiřaditelného zpoždění ve srovnání s jednotkovým je, že nevyžaduje zvýšení počtu simulačních hodnot. Za zvýšení přesnosti simulace se platí nutností použití nějaké struktury pro práci s modelovým časem (viz *kap. 8.2.5*).

Nejpřesnější modely jsou obvykle chakterizovány rozsahem zpoždění, tj. jeho minimální a maximální hodnotou. V tomto případě hovoříme o *přesném zpoždění*. Z daného intervalu je možné náhodně generovat hodnotu zpoždění a pracovat s ní jako s přiřaditelným zpožděním. Častěji se však tento rozsah interpretuje jako interval, v jehož některém okamžiku dochází k přechodu z jedné logické hodnoty na druhou, ale nevíme přesně, ve kterém. Je zřejmé, že zde již nevystačíme s tříhodnotovou simulací, nýbrž je nutné rozšířit obor simulačních hodnot o hodnoty, které jsme si v předchozí kapitole označili R a F.

Pokud vytváříme strukturní model systému, vzniká otázka modelování *zpoždění na spojích*. Buď se zanedbává nebo jej simulátor umožňuje modelovat jako obyčejné dopravní zpoždění, charakterizované jedinou hodnotou. Určitým problémem je, že dostatečné informace o zpoždění na spojích jsou k dispozici až po ukončení fyzického návrhu. To je typické především pro návrh integrovaných obvodů vysokých hustot, kde zpoždění na spojích může i převyšovat zpoždění samotných prvků. Proto simulátory, které jsou součástí profesionálních návrhových systémů pro návrh integrovaných obvodů, umožňují odsimulovat navržený obvod po ukončeném logickém návrhu se zanedbáním zpoždění na spojích nebo se zpožděním, odhadnutým z návrhářem provedeného předběžného rozmístění prvků na ploše čipu a další simulaci provést až po definitivním návrhu rozmístění a propojení prvků s již přesným zpožděním na spojích, vypočítaným ze skutečného tvaru spojů (tzv. *postlayout simulation*).

O použitém modelu zpoždění platí obecně totéž co o oboru simulačních hodnot. Čím je složitější, tím je simulace přesnější za cenu nižší rychlosti a vyšších nároků na paměť. Proto lze v některých simulačních systémech volit jeden z několika různých režimů, lišících se počtem simulačních hodnot a modelem zpoždění. Typickým příkladem je režim simulace poruch, kdy se z časových důvodů používají jednodušší modely signálů i zpoždění.

Součástí ověření správnosti návrhu je i kontrola časování, která má zjistit dodržení potřebných časových relací v systému. Nejčastěji se kontroluje dodržení předepsaných dob předstihu, přesahu, minimální šířky impulsu apod. Při návrhu plně synchronních systémů se používají pro tyto účely specializované prostředky, běžně se však používá i simulace. V takovém případě je nutné použit simulátor pracující s dostatečně přesným modelem zpoždění, navíc jsou obvykle k dispozici prostředky pro hlídání předepsaných relací; v případě zabudovaných modelů jsou kontroly součástí vyhodnocení modelu prvku, lze-li popisovat i chování prvků, musí být k dispozici příkazy pro tyto kontroly.

8.2.4 Simulační algoritmus

Simulačním algoritmem rozumíme postup při šíření změn hodnot signálů modelem. Existují dva základní algoritmy:

- řízení událostmi (event-driven),
- uspořádání po úrovních (levelized scheduling)

Uvažujme velmi jednoduchý příklad obvodu podle *obr.* **??***a)*. Předpokládejme hodnoty na vstupech $A = 1$, $B = 1$, $C = 1$, které se v čase 0 změní na $A = 1$, $B = 0$, $C = 1$, tj. hledáme odezvu na vstupní vektor 101. Zpoždění na spojích zanedbáme.

Při *řízení událostmi* je pořadí vyhodnocování prvků určeno časovou posloupností událostí, přičemž událostí pro nás bude změna hodnoty signálu. Realizace události bude spočívat ve vyhodnocení prvku, ke kterému je daný signál veden. Následkem vyhodnocení může vzniknout nová událost pro signál na výstupu prvku.

Pro odezvu na vstupní vektor bychom mohli algoritmus řízení událostmi vyjádřit takto:

Algoritmus 8.1 - Řízení událostmi

```
naplánuj události pro nový vstupní vektor;
while je plánována nějaká událost do
begin
  for each plánovanou událost do
  begin vezmi událost;
     aktualizuj hodnotu signálu;
```

```
    zapamatuj prvky, ke kterým signál vede;
  end;
  for each zapamatovaný prvek do
  begin vyhodnoť prvek;
    if mění se hodnoty na jeho výstupech then plánuj události;
  end
end
```

Uvedený algoritmus se někdy označuje jako *dvoufázový*, neboť sestává ze dvou základních kroků. V prvém se aktualizují hodnoty signálů a zapamatují prvky pro vyhodnocení, ve druhém probíhá vlastní vyhodnocení prvků a případné plánování událostí pro jejich výstupy. Vyhodnocení prvků je odloženo proto, aby vždy proběhlo s aktuálními hodnotami na vstupech. Za určitých okolností lze použít i tzv. *jednofázový algoritmus*, kdy je prvek vyhodnocen hned.

Simulace našeho příkladu řízená událostmi je shrnuta v *tab. 8.1*, časové průběhy signálů jsou na *obr.* **??***b).*

Tabulka 8.1: Řízení událostmi v příkladu z obr. **??**

Čas	0	t_1	t_2	t_3
Změna signálu	B	S1, S3	S2, F	F
Vyhodnocované prvky	H1, H3	H2, H4	H4	-
Plánování události	S1, S3	S2, F	F	-
Hodnota na výstupu F	1	1	0	1

Všimněme si dvou významných rysů algoritmu řízení událostmi:

- V průběhu simulace jsou vyhodnocovány pouze *aktivní prvky*, tj. ty, které jsou bezprostředně ovlivněny změnami signálů. Tato technika se nazývá *selektivní sledování (selective trace).*
- V závislosti na použitém modelu zpoždění poskytuje simulace i informaci o časovém chování. V našem příkladě zjistíme přítomnost nulového impulsu na výstupu F.

Tyto dva rysy patří mezi hlavní výhody algoritmu řízení událostmi a spolu s obecností algoritmu přispěly k jeho značnému rozšíření. Algoritmus má však i některé nevýhody, které snižují jeho efektivnost. Především vyžaduje implementovat nějakou datovou strukturu pro práci s modelovým časem (blíže si této problematiky všimneme v *kap. 8.2.5*), což především při simulaci na úrovni hradel výrazně zvyšuje časové nároky. Další nevýhodou je, že při zjišťování odezvy na vstupní vektor mohou být některé prvky vyhodnocovány několikrát ($H4$ v našem příkladě), což sice může poskytnout informace o časovém chování, ale to nás např. u složitých synchronních obvodů nemusí zajímat, neboť používáme simulaci pouze pro ověření funkční správnosti a k verifikaci časování, která odhalí případné hazardy, se používají jiné prostředky. V takovém případě vystačíme s nulovým zpožděním a zajímá nás pouze výsledná hodnota na výstupech.

Algoritmus s uspořádáním po úrovních bychom mohli charakterizovat těmito třemi rysy:

- vyžaduje uspořádání prvků po logických úrovních,
- vyhodnocuje vždy všechny prvky bez ohledu na jejich aktivitu

- nepoužívá žádnou datovou strukturu pro práci s modelovým časem.

Použití algoritmu pro náš příklad je uvedeno v *tab. 8.2.*

Tabulka 8.2: Řešení příkladu z obr. **??** uspořádáním po úrovních

	Vyhodnocované prvky	Hodnoty signálů						
		A	B	C	S1	S2	S3	F
Počáteční stav		1	1	1	0	1	0	1
Vstupní vektor		1	0	1				
Úroveň 1	H1,H3				1		1	
Úroveň 2	H2					0		
Úroveň 3	H4							1

Skutečnost, že algoritmus předpokládá možnost uspořádání prvků ve směru toku signálu, tj. od vstupů k výstupům, představuje vážné omezení použitelnosti. Tuto podmínku splňují pouze obvody bez zpětných vazeb. Pokud chceme použít algoritmus pro obvod se zpětnými vazbami, je nutné vyhodnocovat prvky ve zpětnovazebních smyčkách opakovaně, až do dosažení ustáleného stavu.

Uspořádání prvků po úrovních včetně, detekce částí se zpětnými vazbami, se provádí v rámci zpracování zdrojového tvaru modelu. Jde o problematiku obdobnou seřazování bezpaměťových bloků, jak jsme ji poznali v *kap.3.5.*

Na rozdíl od řízení událostmi se u tohoto algoritmu při zjišťování odezvy na nějaký vstupní vektor vyhodnocují jedenkrát všechny prvky bez ohledu na jejich aktivitu s výjimkou prvků ve zpětnovazebních smyčkách, které jsou v rámci iterací vyhodnocovány vícekrát. Tím sice klesá efektivnost algoritmu, ale na druhé straně díky nulovému zpoždění, které algoritmus používá, nevyžaduje časově náročnou práci s modelovým časem.

Nyní se podíváme na *techniky implementace* výše uvedených algoritmů. Zdrojový tvar modelu je zpracován překladačem, který provádí potřebnou analýzu. Generuje cílový kód, který budeme nazývat *přeloženým modelem.* Existují dva základní tvary přeložených modelů:

- tabulkový model,
- kompilovaný model.

V prvém případě se hovoří o *simulaci řízené tabulkami (table-driven)*, případně *interpretační (interpretive)*, ve druhém o *simulaci řízené kompilátorem (compiler-driven)* nebo také *kompilované simulaci (compiled simulation).*

Princip *tabulkového modelu* spočívá v tom, že překladač generuje údajové struktury (tabulky), obsahující informace o prvcích a jejich propojení. Tyto informace jsou v průběhu simulace interpretovány řídicím programem, který realizuje vlastní simulační algoritmus. Jde výhradně o řízení událostmi. Simulační systémy využívající tabulkových modelů proto zahrnují kromě překladače řídicí program, který se často označuje jako simulátor (ve skutečnosti nemusí jít o samostatné programy). Simulátor kromě údajů tabulkového modelu zpravidla interpretuje i příkazy nějakého jazyka, kterým se popisuje simulační experiment (průběhy signálů na vstupech, sledované signály atd.). Podrobněji se budeme simulací řízenou tabulkami zabývat ještě v *kap. 8.2.5.*

Podstata *kompilovaných modelů* je odlišná. Překladač v tomto případě generuje proveditelný kód, tedy spustitelný program, který sám realizuje simulační algoritmus. Kompilované

modely nejčastěji implementují algoritmus uspořádání po úrovních, ale existují i kompilované modely, implementující řízení událostmi.

Uvažujme opět náš příklad z *obr.* **??** a předpokládejme, že překladač generuje pascalovský program. Mohl by mít tvar:

```
program CIRCUIT(input, output);
type SIMVAL = (ZERO, ONE, UNDEFINED);
var A, B, C, F, S1, S2, S3;
function INVERT(I: SIMVAL): SIMVAL;
begin ... end;
function NAND2(I1, I2: SIMVAL): SIMVAL;
begin ... end;
   :
   :
    { tělo programu}
begin
    { inicializace}
         :
         :
  S1 := INVERT(B);           { 1. úroveň}
  S3 := NAND2(B, C);
  S3 := NAND2(A, S1);        { 2. úroveň}
  F  := NAND2(S3, S2);       { 3. úroveň}
         :
         :
end.
```

V programu předpokládáme tříhodnotovou simulaci. V těle programu by po inicializaci, která nastaví hodnoty všech signálů na UNDEFINED, mohl být naprogramován cyklus, v jehož těle se přečte ze souboru vstupní vektor, voláním funkcí modelujících prvky se vypočítají nové hodnoty signálů a ty se zase uloží do souboru. Je zřejmé, že v souladu s algoritmem uspořádání po úrovních se vždy vyhodnocují všechny prvky.

Prvky simulovaného obvodu nemusí být jenom základní logické členy. Navíc zde může kompilátor zařazovat iterační výpočty pro části se zpětnými vazbami. Uvažujme, že kompilátor detekoval v obvodu zpětnovazební smyčku tvořenou prvky H_1 a H_2 podle *obr.* **??**. Mohl by pro tuto část obvodu v daném místě programu vygenerovat příkaz cyklu:

```
I := 0;
repeat
  I    := I + 1;
  POM:= QN;
  Q    := NAND2(SN, QN);
  QN   := NAND2(RN, Q)
until ((QN = POM) or (I = MAXITER));
if (I = MAXITER) then Error;
```

Iterace jsou ukončeny při dosažení ustáleného stavu nebo překročením maximálního povoleného počtu iterací. Druhý případ se považuje za chybu, případně by bylo možné nastavit Q a QN na UNDEFINED.
Většina logických simulátorů, se kterými se v praxi můžeme setkat, využívá tabulkových modelů a řízení událostmi. Rostoucí nároky na výkonnost simulačních systémů a změny

v metodologii návrhu však vedou i k využívání kompilované simulace v návrhových systémech. Umožňuje to skutečnost, že v současné době jsou složité číslicové systémy, především s ohledem na snadnou testovatelnost, přenositelnost mezi technologiemi a využití zřetězeného zpracování, navrhovány jako synchronní. Uvažujme strukturovaně navržený obvod podle *obr.* **??**, kde *KL1*, *KL2*, *KL3* značí kombinační logiku a *PAM1*, resp. *PAM2* paměťové prvky. Protože u synchronního návrhu nejsou podstatné přechodné stavy na výstupech kombinační logiky, nýbrž až hodnoty platné v okamžiku synchronizace, lze za předpokladu uspořádání synchronizačních signálů v čase snadno vytvořit kompilovaný model, protože v obvodu neexistují zpětné vazby, kde by ve smyčce nebyl alespoň jeden paměťový prvek. V našem případě by se při aktivních hodinách h_1 vyhodnocovaly prvky *KL1* (uspořádané po úrovních) a paměťové prvky *PAM1*, při hodinách h_2 by se vyhodnocovaly postupně prvky *KL2*, *PAM2* a *KL3*.

Kompilovaná simulace používaná pro synchronní návrhy se také označuje jako *synchronní simulace*. Dosahuje se u ní rychlosti alespoň o rád vyšší, než u srovnatelného tabulkového simulátoru. Použitím kompilovaného modelu sice neodhalíme případné problémy v časování, ale pro tyto účely existují specializované prostředky pro verifikaci časování.

8.2.5 Simulátory řízené tabulkami

V předchozí kapitole jsme se seznámili s podstatou tabulkových modelů. Nyní se podrobněji podíváme na informace uchovávané v tabulkách modelu a jejich interpretaci při řízení událostmi. Pro jednoduchost se omezíme na případ simulátoru úrovně hradel se zabudovanými modely jednoduchých logických členů s jedním výstupem, které lze parametrizovat počtem vstupů a hodnotou zpoždění. Zpoždění na spojích předpokládáme nulové.

Vstupem by mohlo být logické schéma, na jehož základě překladač vytvoří tabulkový model. Budeme předpokládat, že obsahuje tři typy tabulek:

- *TP* - tabulka prvků. Obsahuje řádek (záznam) pro každý prvek schématu, pro každý primární vstup a výstup. Byly by zde informace o typu prvku, zpoždění, počtu vstupů, počtu spojů vedoucích z výstupu a ukazatele na tabulku vstupů a výstupního větvení příslušející danému prvku. Dále by zde bylo místo pro uložení hodnoty na výstupu, případně další informace využívané při simulaci.

- *TV* - tabulka vstupů. Obsahuje pro každý vstup každého prvku ukazatel do tabulky prvků na řádek příslušející prvku, který je k danému vstupu připojen. Řádky pro vstupy téhož prvku po sobě bezprostředně následují a můžeme je chápat jako dílčí tabulky vstupů jednotlivých prvků. Tabulka vstupů bude sloužit při simulaci ke zpřístupnění hodnot na vstupech prvku při jeho vyhodnocování.

- *TVV* - tabulka výstupního větvení. Obsahuje informaci o propojení prvků, která je využívána při šíření nových hodnot signálů. Mohla by obsahovat pro každý spoj v obvodu řádek s ukazatelem na prvek, ke kterému spoj vede (ve smyslu šíření signálu). Opět po sobě bezprostředně následují řádky pro spoje vedoucí z výstupu téhož prvku, resp. téhož primárního vstupu a tvoří tak dílčí tabulku výstupního větvení.

Vazba mezi tabulkami je naznačena na *obr.* **??**. Na *obr.* **??***a)* je nějaký prvek P se dvěma vstupy a třemi spoji vedoucími z výstupu. Příslušely by mu tabulky podle *obr.* **??***b)*. Ukazatele v *TV*, resp. *TVV* by ukazovaly do tabulky prvků na řádky pro prvky, které jsou připojeny ke vstupům, resp. výstupu prvku P.

Jako konkrétní příklad si ukážeme tvar tabulek pro obvod z *obr.* **??**. Tabulky jsou uvedeny na *obr.* **??**. Řádky tabulek, na které je někde odkaz, jsou označeny symbolickou adresou

začínající znakem @. Všechny informace jsou statické, tj. v průběhu simulace se nemění, s výjimkou hodnoty na výstupech prvků. Je vidět, že zpětné vazby nezpůsobují u tabulkového modelu žádné komplikace. Nyní se podívejme, jak jsou informace v tabulkách využívány při simulaci. Již jsme uvedli, že se výhradně využívá řízení událostmi. S použitím algoritmu *8.1* můžeme zapsat algoritmus simulátoru řízeného tabulkami ve tvaru:

Algoritmus 8.2 - Simulace řízená tabulkami

```
naplánuj události pro řízení simulace;
repeat
  nastav novou hodnotu modelového času;
  for each plánovanou událost do
  begin vezmi událost;
     if jde o událost řízení simulace then proveď odpovídající akci
      else
      begin aktualizuj hodnotu primárního vstupu, resp.~výstupu prvku v TP;
        for each řádek v TVV tohoto primárního vstupu, resp.~prvku do
          do zásobníku prvků ulož hodnotu ukazatele z TVV;
      end
  end;
  for each ukazatel v zásobníku prvků do
  begin použitím  informace v TP vyhodnoť prvek~a~urči~hodnotu~na jeho~výstupu;
      if mění se hodnota na výstupu then plánuj událost pro výstup;
  end
until není plánována žádná událost;
```

V algoritmu *8.2* používáme pro zapamatování prvků, které se mají vyhodnotit, zásobník prvků. V předchozí kapitole jsme uvedli, že simulátor nejen interpretuje informace v tabulkách modelu, ale i příkazy nějakého jazyka pro řízení simulace, který definuje časové průběhy signálů na primárních vstupech, signály, jejichž hodnoty chceme sledovat, ukončení simulace apod. Tuto skutečnost jsme zohlednili v algoritmu předpokladem existence nějakých speciálních událostí pro řídicí akce.

V našich úvahách jsme se omezili na velmi jednoduchý tabulkový simulátor. Zrušení některých našich omezení by představovalo modifikaci zavedených tabulek, případně zavedení dalších. Pokud by šlo o simulátor, jehož vstupní jazyk poskytuje nejen prostředky pro popis struktury, jak jsme předpokládali, ale i prostředky pro popis chování prvků, bylo by nutné tyto informace v nějakém tvaru do tabulkového modelu také umístit. Tyto úvahy necháváme čtenáři ke cvičení.

Již v předchozí kapitole jsme uvedli, že algoritmus řízení událostmi vyžaduje použití údajové struktury pro práci s modelovým časem. Existují dvě základní techniky práce s modelovým časem:

- technika s pevným krokem času,
- techika řízení příští událostí

Technika s pevným krokem času předpokládá rovnoměrné dělení časové osy na intervaly vymezující okamžiky, v nichž může nastat nějaká událost. Intervaly mají délku zvanou *časový krok*. Posun modelového času znamená zvýšení okamžité hodnoty vždy o časový krok. *Řízení příští událostí* předpokládá možnost výskytu události v libovolném časovém okamžiku. Dělení časové osy není pevné a nová hodnota při posunu modelového času je určena okamžikem příští plánované události.

Implementace obou technik vyžaduje údajovou strukturu se základními operacemi — posun modelového času, výběr události a naplánování události.

Implementace techniky s pevným krokem závisí na používaném modelu zpoždění. Pro nulové a jednotkové zpoždění lze použít dva zásobníky událostí: zásobník současné aktivity pro současný čas a budoucí aktivity pro příští časový okamžik. Výběr události znamená vyjmutí záznamu o události, plánování událostí, umístění záznamu o události do zásobníku nových událostí a posun modelového času záměnou obou zásobníků. Při složitějších modelech zpoždění se nejčastěji používá cyklické pole nazývané *časová mapa*. Prvek pole odpovídá jednomu časovému okamžiku a obsahuje ukazatel na zřetězený seznam záznamů o událostech, plánovaných na tento okamžik. Základním problémem techniky s pevným krokem času je určení velikosti časového kroku a délky časové mapy s ohledem na zpoždění prvků v obvodu. Naopak výhodou je jednoduchost a tím i malá časová náročnost požadovaných operací.

Údajovou strukturou pro implementaci řízení příští událostí je struktura zvaná *kalendář událostí* nebo *seznam událostí*. Nejčastěji jde o seznam vzestupně uspořádaný podle plánovaného času výskytu události. Jeho výhodou je obecnost (nejsou zde problémy jako u časové mapy), nevýhodou složitost a tedy i časová náročnost požadovaných operací.

Při logické simulaci se můžeme setkat i se *smíšenou technikou*, která vzniká kombinací obou předchozích. Podstata spočívá v tom, že se postupuje vždy s pevným časovým krokem, avšak pro události, jejichž okamžik výskytu je relativně vzdálený, se vytváří kalendář událostí. Implementace této techniky vyžaduje jak časovou mapu, tak kalendář událostí, sloužící jako "přetoková oblast" pro události plánované na okamžik, který padne mimo časovou mapu. Události z kalendáře se přesouvají do seznamů časové mapy tehdy, až okamžik výskytu událostí padne do aktuálního intervalu časové mapy. Časová náročnost požadovaných operací se blíží technice s pevným krokem tím více, čím méně událostí je zařazováno do kalendáře. Výhodou oproti technice s pevným krokem je lepší využití paměťového prostoru při velkých rozdílech ve zpoždění prvků, problém optimální velikosti časové mapy a kroku času však zůstává.

8.3 Zvláštnosti modelování unipolárních obvodů

Unipolární tranzistory mají některé vlastnosti, které vyžadují specifický přístup k modelování na logické úrovni. Patří mezi ně především schopnost obousměrného přenosu tranzistorem a skutečnost, že popis funkce jednoduchých zapojení nemusí být zcela triviální. Typickým příkladem, u kterého se projevují obě vlastnosti, je tranzistor ve funkci *přenosového hradla* (také *průchozí tranzistor* (*transfer, transmission gate, pass transistor*)) — viz *obr.* **??**. Pokud je na řídicí elektrodě G napětí, které otevře tranzistor, závisí směr přenosu na napětí na vývodech S a D, přesněji řečeno na budicích schopnostech signálů na těchto vývodech.

Příklad přenosového hradla zároveň ukázuje další podstatné rysy obvodů MOS, které musíme při modelování respektovat. Jde o kapacitní jevy a vliv odporu tranzistoru. Obě tyto vlastnosti určují budicí schopnosti signálů, které jsou rozhodující pro určení směru šíření signálu. Této problematiky jsme se již dotkli v kapitole o modelech signálů a víme, že pro vyjádření budicích schopností se používá zpravidla atribut "síla" signálu.

Vzhledem k tomu, že v době nástupu unipolárních technologií se již logická simulace běžně využívala, byla snaha modifikovat simulátory tak, aby byly schopny modelovat i zvláštnosti unipolárních obvodů. To vyžadovalo především vhodně rozšířit model signálu, aby vyjadřoval i budicí schopnosti, a zavedení některých pomocných modelů (sběrnice, jednosměrného přenosového hradla apod.). Zásadním omezením, které brání přirozenému způsobu modelování unipolárních obvodů konvenčními logickými simulátory, je předpoklad

jednosměrného šíření signálů od vstupů k výstupům. V *kap. 8.2.5* jsme viděli, že algoritmus tabulkových simulátorů předpokládá vždy šíření signálu z výstupu prvku k připojeným vstupům. Proto je třeba u takových simulátorů vkládat zvláštní modely, kterým neodpovídá žádný prvek modelovaného obvodu. Například obousměrné přenosové hradlo se modeluje pomocí dvou jednosměrných hradel a speciálního modelu uzlu, který zohledňuje budicí schopnosti signálů.

Mnohem přirozenějším způsobem modelování zvláštností obvodů MOS je *modelování na úrovni spínačů (switch level)* [31], [32]. Základní princip spočívá v tom, že obvod je na této úrovni chápán jako propojení tranzistorů, které se modelují jako spínače. Situaci ilustruje *obr.* **??**. Na *obr.* **??***a)* je zapojení logického členu NOR a na *obr.* **??***b)* jeho model na úrovni spínačů. Unipolární tranzistory jsou modelovány více či méně idealizovaným spínačem. Podstatným rysem, který odlišuje simulaci na úrovni spínačů od běžné elektrické simulace je, že se používají diskrétní modely prvků (hodnoty odporů a kapacit jsou diskretizovány). V našem příkladě je proto zatěžovací tranzistor modelován odporem s hodnotou 1 a zatěžovací kapacita je vyjádřena hodnotou 2.

Simulátory úrovně spínačů chápou hodnotu signálu jako dvojici:

$$h =< v, s >$$

kde v značí *logickou úroveň* a s *sílu signálu*.
Pro logickou úroveň se používá nejčastěji tříprvková ($\{0, 1, X\}$) nebo čtyřprvková množina ($\{0, 1, X, Z\}$) jako u běžné simulace, úrovní síly bývá 2 až 5. Hodnota signálu slouží k vyjádření jak logické úrovně, tak síly signálu v uzlu obvodu. Rozlišují se *uzly vstupní*, v nichž mají signály nejvyšší budicí schopnost, a *uzly paměťové* (všechny ostatní), kde předpokládáme schopnost uchovávat náboj.

Tranzistor se modeluje jako spínač, jehož odpor v sepnutém stavu určuje ohodnocení tranzistoru. Podobně kapacitní vlastnosti paměťových uzlů určují jejich ohodnocení. Vstupní uzly mají nejvyšší hodnotu síly.

Při průchodu signálu tranzistorem ovlivňuje ohodnocení tranzistoru sílu signálu přicházejícího do uzlu. V uzlu se tento signál kombinuje obecně jednak s původní hodnotou v uzlu, jednak s ostatními signály přicházejícími do uzlu. Klíčový význam na úrovni spínačů má proto *funkce spojení (connection function)*, která určuje výslednou hodnotu kombinace různých signálů. Je zřejmé, že nová hodnota se obecně šíří opět do sousedních uzlů, a proto je možné bez problémů modelovat obousměrné prvky.

Protože nalezení ustáleného stavu šířením signálů mezi uzly by bylo pro rozsáhlé obvody časově náročné, je důležitou fází většiny simulačních systémů a programů úrovně spínačů *rozčlenění modelovaného obvodu* na množinu propojených a vzájemně komunikujících komponent. *Komponentou* budeme rozumět část obvodu tvořenou tranzistory vzájemně spojenými prostřednictvím emitorů a kolektorů spolu s uzly tvořícími tato spojení. Formálně můžeme převést hledání komponent na řešení problému nalezení komponent neorientovaného grafu (tzv. *graf kanálů*), jehož vrcholy odpovídají uzlům obvodu a hrany kanálům tranzistorů. Cílem je rozčlenit celý modelovaný obvod na podobvody, pro něž se bude hledat šířením signálů ustálený stav, zatímco mezi komponentami se signály budou šířit stejným způsobem jako při konvenční logické simulaci mezi logickými prvky, kdy stačí uvažovat pouze logickou úroveň. Situaci ilustruje *obr.* **??**, kde na *obr. a)* je rozčlenění na komponenty a na *obr. b)* graf kanálů.

Další výhodou rozčlenění obvodu na komponenty je, že umožňuje snadno implementovat *smíšené simulátory*, u kterých jsou části obvodu modelovány na úrovni spínačů, zatímco jiné mohou být modelovány na vyšších úrovních abstrakce.

Simulační algoritmus může být podobný algoritmu řízení událostmi s tím, že prvkům nyní odpovídají komponenty. Důležitým krokem je *vyčíslení komponenty*. Odpovídá vyhodnocení

prvku v*alg. 8.1.* Znamená určení nových hodnot v uzlech komponenty na základě původních hodnot v paměťových uzlech, hodnot ve vstupních uzlech a hodnot na řídicích elektrodách tranzistorů. U většiny simulátorů se určuje nový ustálený stav a jeho hodnoty se nastavují v uzlech v závislosti na modelu zpoždění. Používá se zpoždění nulové , jednotkové, různé pro nárůst a pokles signálu, případně přesné zpoždění určené z RC modelu.

Pro výpočet ustáleného stavu se používají různé metody. Většina simulátorů používá nějakou *iterační metodu*. Každý iterační krok představuje šíření hodnoty v jednom uzlu přes tranzistor do sousedního uzlu, kde se kombinuje s hodnotou zde již přítomnou. Proces pokračuje tak dlouho, dokud se nedosáhne ustáleného stavu. K implementaci lze použít přístup podobný řízení událostmi. Kromě simulátorů využívajících iteračních metod se můžeme setkat i s programy, které generují kód obdobný *kompilovaným modelům* pro logickou simulaci (např. rozčlení obvod na komponenety a pro ně definují soustavu booleovských rovnic pro výpočet hodnot v uzlech).

8.4 Simulace poruch

Jednou z důležitých oblastí využití logické simulace je simulace poruch. Používá se pro vyhodnocování kvality diagnostických testů. Ta se nejčastěji vyjadřuje *pokrytím*, tj. procentem poruch detekovaných daným testem z množiny všech uvažovaných poruch.Úlohu simulace poruch bychom mohli formulovat takto:

Je dán obvod, množina poruch, jejichž výskyt v obvodu předpokládáme, a diagnostický test (posloupnost testovacích vektorů). Zjisti, které poruchy z uvažované množiny jsou daným diagnostickým testem detekovány.

Zjistit, zda je určitá porucha detekovatelná diagnostickým testem v daném místě, můžeme tak, že odsimulujeme bezporuchový obvod, vytvoříme model obvodu s poruchou, který také odsimulujeme pro zadaný diagnostický test, a srovnáme výsledky. Pokud se hodnoty signálu v místě, kde chceme poruchu detekovat, pro oba simulační běhy liší, je daná porucha v tomto místě detekovatelná.

Důležitou otázkou diagnostiky logických obvodů jsou *modely poruch*. Fyzikální poruchy, ke kterým dochází v číslicovém zařízení, ovlivňují jistým způsobem jeho chování. Modelem poruchy rozumíme logickou reprezentaci fyzikální poruchy, která umožňuje s požadovanou přesností modelovat logické chování obvodu při jejím výskytu. Volba modelu poruchy úzce souvisí s úrovní abstrakce, na níž je číslicové zařízení popsáno. Metody diagnostiky jsou nejvíce propracovány pro úroveň základních logických členů, pro níž se zpravidla používají modely označované *trvalá nula (t0)* a *trvalá jednička (t1)*. Tyto poruchy reprezentují zejména pro bipolární technologie velmi dobře fyzikální poruchy.

Již bylo uvedeno, že simulace poruch vyžaduje schopnost simulovat jak bezporuchový obvod, tak obvod s poruchou (nejčastěji se předpokládá výskyt pouze jediné poruchy v obvodu). Má-li být simulační systém použitelný pro simulaci poruch, musí zajistit některé akce, které se nevyskytují u bezporuchové simulace. Jde především o *specifikaci, vkládání (injekci), šíření a detekci poruch* a o *zpracování výsledků*. Specifikace poruch spočívá v určení, které poruchy a jakých typů budou simulovány. Zpravidla se simulují všechny poruchy, které jsme schopni v daném obvodu modelovat. Vkládání poruch představuje transformaci modelu bezporuchového obvodu na obvod s poruchami. Určuje se typ poruchy a místo výskytu. Simulační systém musí být schopen zajistit šíření vlivu vložených poruch a rozhodnout o detekovatelnosti poruchy srovnáním hodnot pro bezporuchový obvod a obvod s poruchou. Systémy pro simulaci poruch musí na závěr určit pokrytí testu, případně poskytnout výsledky dalším programům, které získají další důležité informace pro diagnostiku. Vzhledem k časové náročnosti bývá často pro simulaci poruch určen zvláštní režim simulátoru, případně

jde o specializovaný simulátor poruch s co nejjednodušším modelem signálu (tříhodnotový, čtyřhodnotový) i zpoždění. Triviálním případem simulace poruch je tzv. *sekvenční (sériová) simulace poruch*. Umožňuje odsimulovat v jednom simulačním běhu chování jediného obvodu s poruchou. Injekce poruch se provádí modifikací reprezentace struktury obvodu (rozpojení/vytvoření spoje, připojení spoje na zdroj konstantní logické úrovně apod.), případně modifikací popisu chování prvků (abstraktnější funkční poruchy). Vzhledem k tomu, že potřebujeme zjišťovat detekovatelnost všech poruch z určité množiny, jejíž mohutnost je typicky značná, bylo by zapotřebí při simulaci uvedeným způsobem provést velké množství simulačních běhů. Proto byly vyvinuty speciální techniky, jejichž hlavním cílem je redukovat počet simulačních běhů a tím i zvýšit rychlost simulace. Existují tři základní techniky simulace poruch redukující počet simulačních běhů:

- paralelní (parallel),
- deduktivní (deductive),
- souběžná (concurrent)

Paralelní simulace poruch je vhodná pro úroveň základních logických členů, jejichž chování lze popsat pomocí běžných logických operací. Ke snížení počtu simulačních běhů se používá tzv. *bitového kódování* simulačních hodnot, které spočívá v uložení simulační hodnoty na několika bitech slova hostitelského počítače. Uvažujme dvouhodnotovou simulaci a délku slova W. K zakódování simulační hodnoty nám stačí 1 bit a zbývajících $W-1$ bitů můžeme využít k uložení dalších simulačních hodnot a to téhož signálu, ale pro obvod s různými vloženými poruchami. Lze tedy paralelně simulovat bezporuchový obvod a $W-1$ obvodů s poruchou (tzv. poruchových obvodů).

Uvažujme jednoduchý příklad z *obr.* **??**. Předpokládejme, že chceme simulovat všechny poruchy typu $t0$ a $t1$. Je jich celkem 10 a jsou v obrázku očíslovány. Použijeme-li slovo délky 8 bitů, můžeme za předpokladu nejjednoduššího dvouhodnotového modelu signálu současně simulovat bezporuchový obvod a 7 obvodů poruchových. Nejsme tedy schopni odsimulovat všechny poruchy současně, nýbrž budeme muset uskutečnit dva simulační běhy - pro bezporuchový obvod a obvod s poruchami 1 až 7 a potom bezporuchový obvod a obvody s poruchami 8 až 10. V obrázku jsou uvedeny obsahy slov představující signály A, B, C, D a E. V nejpravějším bitu je uložena hodnota pro bezporuchový obvod. Hodnoty pro výstup prvku se počítají logickými operacemi nad celými slovy s respektováním případných poruch pro výstup.

Výhodou paralelní simulace poruch je jednoduchost vyhodnocování prvků a algoritmů pro vkládání, šíření i detekci poruch. Je zřejmé, že její efektivnost roste s délkou slova hostitelského počítače (někdy se zpracovává i několik slov).

Pro úroveň základních logických členů je vhodná také *deduktivní simulace poruch*. Vychází z myšlenky, že jsou-li známy hodnoty na vstupech logického členu obvodu bez poruch a vliv poruch v obvodu na tyto hodnoty, lze určit (dedukovat) hodnotu na výstupu prvku pro bezporuchový obvod i vliv poruch na ni. Postup tohoto výpočtu se nazývá *deduktivní algoritmus*.

Důležitým pojmem deduktivní simulace je *seznam poruch*, který je připojen ke každému signálu na vodiči modelovaného obvodu a vyjadřuje vliv všech simulovaných poruch na hodnotu signálu na tomto vodiči. Chápeme ho jako seznam jmen všech simulovaných poruch v obvodě, při jejichž výskytu se v daném čase pro právě simulovaný krok testu liší hodnota signálu oproti bezporuchovému stavu. Deduktivní algoritmy pro dvouhodnotovou simulaci a členy AND a OR lze nalézt v [6].

Výhodou deduktivní simulace ve srovnání s paralelní je, že se současně simulují všechny uvažované poruchy. Na druhé straně paměťové nároky jsou vyšší a také režie operací se seznamy je značná. Při použití deduktivní simulace musí existovat deduktivní algoritmus pro každý typ prvku modelované struktury. Proto je tato technika použitelná pouze u simulátorů se zabudovanými modely, navíc musí jít o modely jednoduchých prvků, pro které jsme schopni tento algoritmus určit.

Souběžná simulace poruch vychází ze skutečnosti, že poruchy představují malé změny ve struktuře logického obvodu, které vedou zpravidla k malým změnám stavu obvodu. Jinými slovy, tyto změny ovlivňují pouze malou část obvodu. Z toho plyne, že je možné souběžně simulovat řadu takových "podobných" obvodů tak, že se vytvoří tzv. *referenční model* a řada tzv. *souběžných modelů*, reprezentovaných pouze odlišnostmi od modelu referenčního.

I souběžnou simulaci můžeme charakterizovat jako orientovanou na seznamy. V případě, že nějaká porucha způsobí změnu hodnot na vstupech, vytvoří se kopie prvku, avšak se změněnými hodnotami. Tuto kopii nazýváme *poruchovým prvkem (faulty machine)*. Ke každému prvku referenčního modelu je proto připojen seznam poruchových prvků. Pro šíření poruch se nepoužívá žádná algebra, nýbrž všechny poruchové prvky se explicitně simulují.

Souběžná simulace poruch využívá tabulkových modelů a řízení událostmi. Referenční model je představován kompletním tabulkovým modelem obvodu, zatímco souběžné modely pouze tabulkami poruchových prvků. Poruchové prvky navíc v průběhu simulace dynamicky vznikají a zanikají podle toho, zda se u daného prvku v daném modelovém čase liší hodnoty na vstupech a výstupech obvodu s poruchou reprezentovaného daným souběžným modelem od referenčního modelu.

Simulační algoritmus má podstatně vyšší režii ve srovnání s obyčejnou bezporuchovou simulací právě z důvodu potřebné analýzy seznamů poruchových prvků. Souběžná simulace poruch klade poměrně vysoké nároky na paměť a má značně větší režii ve srovnání se sekvenční simulací. Díky tomu, že se simulují všechny poruchy současně, je však pro rozsáhlejší obvody s velkým množstvím simulovaných poruch podstatně rychlejší. Ve srovnání s paralelní a deduktivní simulací má zásadní výhodu v tom, že je použitelná nejen pro úroveň základních logických členů, nýbrž i pro složitější modely a především modely prvků vytvářené uživatelem.

I přes použití speciálních technik je časová náročnost simulace poruch pro složité obvody značná. Proto se hledají alternativní přístupy, které nejčastěji využívají k určení pokrytí *pravděpodobnosti* a *statistických odhadů* [33]. Používá se obyčejná bezporuchová simulace, v průběhu simulace se navíc pouze sbírají některé statistiky, které potom slouží pro odhad pokrytí. Zpravidla se využívá pojmů *říditelnost* (*controlability*) a *pozorovatelnost* (*observability*) v logické jedničce a nule. Prvý vyjadřuje pravděpodobnost, že pro náhodně vybraný vstupní vektor bude na daném spoji hodnota 1, resp. 0. Druhý vyjadřuje pravděpodobnost, že daný spoj, je-li na něm 0, resp. 1 bude pozorovatelný na primárním výstupu. Statistické metody jsou velmi atraktivní pro rychlý a dostatečně přesný odhad pokrytí.

8.5 Simulační systém OrCAD/VST

V této kapitole popíšeme prostředí komponenty VST (Verification and Simulation Tools) systému OrCAD [34]. Smyslem je ukázat prostředky, jaké poskytuje moderní jednoduchý logický simulátor, implementovaný na osobním počítači. Popis bude zaměřen uživatelsky, aby mohl být využit při cvičeních u počítače. Budeme předpokládat základní znalost práce s komponentou SDT (Schematic Design Tools) systému OrCAD, především kreslení schémat programem DRAFT.

8.5.1 Základní vlastnosti simulátoru

Základem komponenty VST je *simulátor* SIMULATE, sloužící k ověření správnosti návrhu obvodu, jehož schéma je popsáno programem ve formátu EDIF. Simulátor lze charakterizovat těmito základními rysy:

- Je *řízen událostmi*.
- *Model signálu* rozeznává tři logické úrovně (nízká - 0, vysoká - 1, nedefinovaná - X) s použitím čtyř úrovní síly signálu. Nejvyšší budicí schopnosti má napájení (supply). Je použito pro budicí signály - stimuly (stimulus). Další úrovní je buzení (driving), které se používá pro výstupy logických členů budících nějaký uzel. Další úroveň je označena jako odporová (resistive). Tuto úroveň síly mají např. výstupy logických členů s otevřenými kolektory (pro výstupní signál vysoká). Nejnižší úrovní síly je vysoká impedance (Z).
- Maximální *rozsah modelovaných obvodů* závisí na řadě okolností, např. použité knihovně, délkách identifikátorů signálů atd. Uvádí se odhad asi 10000 ekvivalentních dvouvstupých hradel.
- *Budicí signály* lze vytvářet využitím editoru stimulů, který je součástí prostředí simulátoru, nebo použitím souboru testovacích vektorů.
- *Časová jednotka* je typicky 1ns, pro ECL 0.1ns a je určena knihovnou.
- Prostředí simulátoru zahrnuje tři editory: *editor stimulů* (*stimulus editor*), *editor sledovaných signálů* (*trace editor*), *editor podmínek zastavení* (*breakpoint editor*).
- *Výsledky simulace* lze zobrazovat graficky na obrazovce i ukládat do souboru. Zobrazovat lze až 50 kanálů (jednoduchých nebo sběrnic), celkem maximálně 250 různých signálů. Zobrazovaný průběh může být definován jako signál nebo sběrnice, hodnota na sběrnici může být vyjádřena v binárním, oktalovém, decimálním nebo hexadecimálním tvaru. Grafické výstupy výsledků simulace mají podobu průběhů, snímaných logickým analyzátorem. Pro prohlížení průběhů lze použít čtyř časových měřítek a lze provádět měření intervalů s využitím až tří značek (marker), kterými je možné označit časové okamžiky. Je zajištěn vertikální i horizontální posun.
- Editorem podmínek zastavení lze nastavit až 10 podmínek.
- Simulaci lze kdykoliv znovu spustit od času 0.
- Lze pracovat s minimálním nebo maximálním *zpožděním*.
- Definice stimulů, podmínek zastavení i sledovaných signálů lze uložit na disk, podobně výsledky simulace k pozdějšímu prohlížení.
- Pro použití simulátoru musí být dodrženy určité konvence, týkající se především jmen sběrnic, signálů a portů při vytváření schématu editorem DRAFT.

8.5.2 Příprava pro simulaci

Základní postup je naznačen na *obr.* **??**. Simulátor vyžaduje jako vstup seznam spojů (netlist) ve formátu EDIF popisující strukturu obvodu.

V prostředí komponenty OrCAD/SDT jej můžeme vygenerovat programem NETLIST příkazem:

```
NETLIST obvod.sch obvod.net /p
```

`obvod.sch` je soubor s výkresem vytvořený programem DRAFT, `obvod.net` je soubor, do kterého ukládá program NETLIST seznam spojů ve formátu EDIF (implicitní formát programu NETLIST). Nejvhodnější je na tomto místě uvést úplnou cestu do pracovního adresáře simulátoru, např. `..\\VST\\DESIGN\\OBVOD.NET`
klíč `/p` indikuje požadavek na použití čísel špiček místo jmen.

Simulátor vyžaduje jako vstup seznam spojů, definice budicích stimulů (`obvod.stm`), a seznam sledovaných signálů (`obvod.trc`). Poslední dva soubory máme uloženy na disku od předchozí simulace nebo definice vytvoříme interakčně použitím editorů integrovaných v prostředí simulátoru. Budicí signály mohou být také definovány ve tvaru budicích vektorů (obvod.tsv). Od předchozí simulace můžeme mít na disku uloženy i podmínky zastavení simulace, definované opět zabudovaným editorem. Při výstavbě simulačního modelu využívá simulátor knihovny modelů (`model.lib`).
Simulátor se spouští příkazem ve tvaru:

```
SIMULATE obvod.net [/c]
```

`obvod.net` označuje soubor ve formátu EDIF, ze kterého vytváří simulátor simulační model. Implicitně se předpokládá, že soubor leží v pracovním adresáři simulátoru (DESIGN).

Při zadání klíče `/c` lze provádět změny v konfiguraci (přístupová cesta k ovládačům, ovládač displeje, tiskárny, cesta ke knihovně, pracovnímu adresáři, adresáři pro přechodné soubory, nastavení barev apod.).

8.5.3 Příkazy simulátoru

Základní ovládání simulátoru je stejné jako ovládání editoru schémat DRAFT. V některých případech je třeba zadávat *jména signálů*. U jednoduchého spoje je jméno totožné s odpovídajícím návěštím spoje (u schématu organizovaného jako skupina výkresů (flat file) je součástí jména ještě podtržítko a číslo výkresu). Odkaz na signál na některém bitu sběrnice je tvaru:

```
jm_sběrnice:index
```

Lze se také odkazovat na vývod prvku ve tvaru:

```
.jm_výskytu-č_vývodu
```

Následuje abecední seznam příkazů simulátoru:

- **Again** - opakuje předchozí příkaz hlavního menu.
- **Breakpoint** - umožňuje definovat až 10 podmínek pro zastavení simulace. Je třeba si uvědomit, že použití podmínek zastavení simulace díky časté nutnosti testování snižuje rychlost simulace.
- **Disable Breakpoints** - globální pozastavení sledování podmínek zastavení simulace.
- **Enable Breakpoints** - globální povolení sledování podmínek zastavení simulace.
- **Breakpoint Edit** - vyvolání editoru podmínek zastavení simulace. Podmínka je definována logickým součinem nebo součtem až 16-ti signálů. Operace je vždy společná pro všechny signály. Signál v podmínce může být negovaný, což se označí znakem před jménem signálu.

- **Edit** - editace zvolené podmínky zastavení. Kurzor je ve tvaru zvýraznění, řádkové menu obsahuje položky v závislosti na místě, kde se nachází kurzor. Lze individuálně povolit/zakázat (enable/disable) sledování editované podmínky (Status), zadat typ operace pro podmínku (And/Or), nastavit úroveň hierarchie pro signály v podmínce (Context) a editovat signály podmínky (Breakpoint Bit). Signály lze přidávat (Add), rušit (Delete), editovat (Edit), vkládat nad kurzor (Insert). Lze se dostat také přímo do menu příkazu Macro, což umožňuje práci s makry i na této úrovni. K návratu slouží příkaz Return.
- **Macro** - přístup k menu příkazu Macro.
- **Quit** - ukončení činnosti editoru podmínek zastavení.
- **Read** - načtení podmínek zastavení ze souboru (implicitně z pracovního adresáře nastaveného v konfiguraci s rozšířením .brk).
- **Status** - povolení/zákaz sledování podmínky, na níž právě ukazuje kurzor.
- **Use** - zpracování podmínek zastavení simulace pro použití simulátorem. Při chybě se vypíše zpráva a chybná specifikace je zvýrazněna. Tento příkaz je nutno dát na závěr práce s editorem podmínek, mají-li být použity.
- **Write** - zápis podmínek zastavení do souboru (implicitně do pracovního adresáře nastaveného v konfiguraci s rozšířením .brk).
- **Conditions** - zobrazí se informace o velikosti oblastí paměti pro model, stimuly a makra.
- **Delete Marker** - zruší nastavenou značku (příkazem Place Marker). Je-li jich více, požaduje se zadání čísla značky (1, 2 nebo 3).
- **Edit Stimulus** - vyvolání editoru stimulů pro definování budicích signálů.
- **Add** - připojuje další položku k seznamu budicích signálů (max.200 položek včetně signálů definovaných souborem testovacích vektorů).
- **Delete** - zruší zvýrazněnou položku ze seznamu.
- **Edit** - editace detailní specifikace. Kurzor je ve tvaru zvýraznění, řádkové menu obsahuje položky v závislosti na místě, kde se nachází kurzor. Lze nastavit úroveň hierarchie pro signál, jehož průběh definujeme (Context), zadat jméno signálu (Signal Name), počáteční stav signálu (Initial Value) - 0, 1, X, Z, a definovat tvar budicích signálů (Stimulus Specification). Každá položka seznamu definice průběhu sestává z času a funkce. Zadávání času může být nastaveno pro absolutní hodnoty (Abs.Mode) nebo relativní (Rel.Mode). Funkce nabývá hodnoty 0, 1, X, Z, T (překlopení do opačné hodnoty) nebo GOTO (nekonečný cyklus). V případě GOTO se požaduje zadání čísla položky seznamu, na níž má být skok proveden. Položky definičního seznamu lze přidávat (Add), rušit (Delete), editovat (Edit), vkládat nad kurzor (Insert). Lze se dostat také přímo do menu příkazu Macro, což umožňuje práci s makry i na této úrovni. K návratu slouží příkaz Return.
- **Insert** - vkládá novou položku seznamu budicích signálů nad kurzor.
- **Macro** - přístup k menu příkazu Macro.

- **Quit** - ukončení činnosti editoru stimulů.
- **Read** - načtení definice stimulů ze souboru (implicitně z pracovního adresáře nastaveného v konfiguraci s rozšířením .stm).
- **Set** - nastavuje způsob zadávání času - absolutní (Absolute Mode) nebo relativní (Relative Mode).
- **Test Vector Editor** - specifikace formátu souboru testovacích vektorů. Soubor testovacích vektorů se vytváří nějakým textovým editorem mimo prostředí simulátoru. Ten potřebuje znát pouze formát souboru. Při specifikaci formátu je kurzor ve tvaru zvýraznění, řádkové menu obsahuje položky v závislosti na místě, kde se nachází kurzor. Lze povolit, resp. zakázat načítání vektorů ze souboru v průběhu simulace (Test Vector Input - Enable/Disable), zadat jméno souboru testovacích vektorů (Test Vector File Name - implicitně pracovní podadresář podle konfigurace a rozšíření .tvs) a určit pozice hodnot signálů a času v budicích vektorech. Testovací vektory se zapisují v souboru tak, že na každém řádku je jeden vektor obsahující pole s hodnotu času a pole hodnot budicích signálů v daném čase. Středník na prvé pozici uvozuje řádek poznámky, posloupnost testovacích vektorů je zakončena klíčovým slovem END začínajícím na 1.pozici. Editoru testovacích vektorů se potom zadává začátek pole hodnot času (Time Input Column), šířka pole pro čas (Time Input Width) a dále se zadává seznam jmen signálů (Item Names or Column). Každá položka definuje pozici jednoho signálu ve vektoru (maximální počet sloupců ve vektoru je 200). Položka je tvořena úplným jménem signálu, včetně označení úrovně hierarchie (kořen implicitně .) a pozicí ve vektoru. Položky seznamu lze přidávat (Add), rušit (Delete), editovat (Edit), vkládat nad kurzor (Insert).
- **Use** - zpracování definice budicích signálů pro použití simulátorem. Při chybě se vypíše zpráva a chybná specifikace je zvýrazněna. Chyby z překladu specifikace souboru testovacích vektorů jsou uloženy v souboru s rozšířením .lst.
- **Write** - zápis definice stimulů do souboru (implicitně do pracovního adresáře nastaveného v konfiguraci s rozšířením .stm).
- **Hardcopy** - tisk výsledků simulace na tiskárně nebo uložení do tiskového souboru.
- **Destination** - LPT nebo File (vytiskne se příkazem COPY ... prn:/b).
- **File Mode** - soubor lze přepsat (Replaced) nebo připojit (Appended).
- **Start Time** - tisk od času.
- **End Time** - tisk po čas (implicitně jedna stránka).
- **Make Hardcopy** - uskuteční tisk. Obsah obrazovky lze vytisknout běžně klávesou PrtScr.
- **Width of Paper** - lze vybrat papír šířky 8, resp. 13" (Narrow, resp. Wide).
- **Initialize** - nastavení modelu do počátečního stavu (v čase 0).
- **Macro** - práce s makry. Lze používat funkční klávesy, řídicí klávesy CTRL a SHIFT v kombinaci s ostatními klávesami, a prostřední tlačítko myši(MMB).

- **Capture** - vytváření makra: 1.zadá se definiční klávesa, 2.zadávání příkazů, 3.ukončení definice makra klávesou M.
- **Delete** - zrušení makra na základě definiční klávesy.
- **Initialize** - smazání všech maker.
- **List** - výpis definovaných maker reprezentovaných definičními klávesami.
- **Write** - zápis všech definovaných maker do souboru.
- **Read** - načtení souboru maker.
- **Place Marker** - umístění značek pro měření časových intervalů (hodnota se zobrazuje na řádku hlášení v horní části obrazovky). Implicitně lze zadávat pouze jednu značku. Příkazem SET (viz dále) lze povolit až 3 značky. V takovém případě má značka svoje číslo (1 až 3). Značka zůstává zachována v časovém okamžiku, kterému byla přiřazena, dokud není zrušena příkazem Delete Marker, a to i při restartu simulace.
- **Quit** - ukončení činnosti simulátoru (Abandon Program) nebo vymazání části obrazovky s časovými průběhy (Clear Trace Display).
- **Run Simulation** - spuštění simulace, zadává se délka. Simulace pokračuje od stavu, ve kterém se model nachází. Simulaci lze přerušit stiskem kláves CTRL BREAK.
- **Set** - nastavení různých parametrů a podmínek.
- **AutoPan** - automatický posun v horizontálním směru při dosažení hranice obrazovky.
- **Vertical Screen Step** - nastavení velikosti vertikálního posunu (2, 4, 8, 16 - tj. 1/2, 1/4, atd. vertikálního rozměru obrazovky). Na rozdíl od horizontálního posunu není vertikální posun automatický, nýbrž je ovládán klávesami PgUp a PgDn.
- **Horizontal Screen Step** - nastavení velikosti horizontálního posunu obrazovky. Způsob nastavení je stejný jako u vertikálního posunu.
- **Error Bell** - ovládání zvukového signálu při chybě.
- **Trace Name Dividers** - jména zobrazovaných signálů jsou pro lepší orientaci oddělena podtržením.
- **Multiple Markers** - povolení/zákaz současného nastavení více (max.3) značek.
- **Reticle Display** - zobrazení indikátoru, ukazujícího kolik a kterou část vyrovnávací paměti sledovaných signálů máme zobrazenu na obrazovce.
- **Spool To Disk** - zápis výsledků simulace na disk. Zápis do souboru se provádí pouze v případě, že je povolen příkazem Enable Trace (viz příkaz Trace). Po povolení zápisu na disk lze měnit parametry trasování, ale nelze již přidávat nové signály. V editoru sledovaných signálů (viz příkaz Trace) lze načíst soubor s výsledky jako specifikační pro definici sledovaných signálů a potom si pouze prohlížet výsledky simulace (při spuštění simulátoru je však potřeba zadat tentýž seznam spojů, se kterým probíhala simulace, jejíž výsledky si chceme prohlížet). Při inicializaci simulátoru se soubor s výsledky simulace, do něhož se zapisuje, uzavírá.

- **Trace** - práce se sledovanými signály. Koncepce trasování vychází z činnosti při použití logického analyzátoru (umísťují se sondy do těch míst obvodu, kde nás zajímají signály a ty se potom zobrazují na obrazovce).
- **Change View** - změna periody vzorkování sledovaných signálů (implicitně 1). Při prohlížení výsledků simulace ze souboru má příkaz za následek změnu časového měřítka pro zobrazování.
- **Disable Trace** - potlačení sledování signálů.
- **Enable Trace** - povolení sledování signálů.
- **Set Start Time** - nastavení okamžiku, od kterého mají být signály vzorkovány.
- **Trace Edit** - spuštění editoru sledovaných signálů (trace editor). Lze definovat až 50 trasovacích kanálů (jednoduchých signálů nebo sběrnic (max.16 signálů)). Pro trasování může být definováno celkem až 250 signálů. Podobně jako u ostatních editorů integrovaných v prostředí simulátoru, má kurzor tvar zvýraznění a řádkové menu obsahuje položky v závislosti na místě, kde se nachází kurzor. Sledované signály se definují pomocí těchto položek:
- **Display Name** - jméno, které se vypíše u příslušného signálu na obrazovce (max.16 znaků). Lze používat příkazy:
- **Add** - přidání položky na konec seznamu, přechod do režimu podrobné specifikace (viz dále - Edit).
- **Copy** - zkopírování informací o položce, na kterou ukazuje kurzor do/z vyrovnávací paměti (Copy To Buffer/Copy From Buffer) - vhodné pro reorganizaci seznamu sledovaných signálů.
- **Delete** - zrušení položky seznamu.
- **Edit** - podrobná specifikace sledovaného signálu. Lze editovat:
- **Display Name** - zobrazované jméno signálu (viz nahoře),
- **Type** - typ zobrazení (Binarybus, Decimalbus, Hexbus, Octalbus, Signal).
- **Trace** - individuální povolení/blokování vzorkování signálu (On/Off).
- **Display** - příznak, zda se má sledovaný signál zobrazovat (On/Off).
- **Context** - definování úrovně hierarchie.
- **Signal Name** nebo **Bit # Position** - jméno signálu nebo seznam jmen signálů sdružených pro trasování do sběrnice (lze přidávat - Add, rušit - Delete, vkládat - Insert a editovat - Edit).
- **Insert** - vložení položky nad kurzor, přechod do podrobné specifikace (viz nahoře - Edit).
- **Macro** - přechod do menu pro práci s makry,
- **Quit** - ukončení činnosti editoru sledovaných signálů s návratem do hlavního menu,

- **Read** - načtení specifikace sledovaných signálů ze souboru (implicitně z pracovního adresáře nastaveného v konfiguraci s rozšířením ....trc).
- **Use** - kontrola správnosti specifikace a převod do tvaru vyžadovaného simulátorem. Při chybě se vypíše zpráva a chybná specifikace je zvýrazněna. Pokud se provádí zápis výsledků do souboru a byly přidány nějaké nové signály, zobrazí se chybové hlášení a lze uzavřít soubor s výsledky a ukončit ukládání (Disable spooling), nalézt v seznamu přidané signály (Find New Node), přepsat soubor (Rewrite Spool File) nebo uzavřít stávající soubor a začít zapisovat do nového (Select New Spool File).
- **Write** - zápis specifikace sledovaných signálů do souboru (implicitně do pracovního adresáře nastaveného v konfiguraci s rozšířením ....trc).
- **Type** - typ zobrazení (viz nahoře - Edit).
- **Trace** - individuální povolení/blokování vzorkování signálu (viz nahoře - Edit).
- **Display** - příznak, zda se má sledovaný signál zobrazovat (viz nahoře - Edit).
- **Zoom** - manipulace s časovými průběhy v horizontálním a vertikálním směru. Při zobrazování se využívají výsledky, uložené ve vyrovnávací paměti sledovaných signálů. Pokud došlo při simulaci k přeplnění (uživatel potvrzuje), lze zobrazovat pouze v rozsahu uchovaném v této paměti. Jestliže však jsou ukládány výsledky do souboru na disk, doplňují se potřebné hodnoty do paměti a lze tedy zobrazovat v plném rozsahu.
- **Begin At Cursor** - posun doleva tak, aby byl kurzor na levém okraji obrazovky.
- **End At Cursor** - posun doprava tak, aby byl kurzor na pravém okraji obrazovky.
- **Center At Cursor** - posun doprava nebo doleva tak, aby byl kurzor uprostřed.
- **Up** - vertikální posun tak, aby byl průběh, na který ukazuje kurzor, jako první na obrazovce.
- **Down** - vertikální posun tak, aby byl první průběh na obrazovce na pozici kurzoru.
- **Top** - vertikální posun na začátek seznamu sledovaných signálů.
- **Scale** - horizontální měřítko (1, 2, 4, 8).
- **Next Page** - posun v horizontálním směru o jednu stránku doprava.
- **Previous Page** - posun v horizontálním směru o jednu stránku doleva.
- **Initial Display** - nastavení levé hranice času na obrazovce, např. pro rychlý přesun při prohlížení výsledků nebo na začátku prohlížení výsledků ze souboru.

8.6 Knihovny simulátoru

Modely prvků jsou uloženy v knihovně `model.lib`. Jádrem knihovny jsou modely základních prvků, jejichž pomocí lze vytvářet modely složitější. K přizpůsobování knihovny potřebám slouží program MODELPRRO, který překládá modely prvků ve zdrojovém tvaru do formátu knihovny simulátoru a přidává je ke stávající knihovně. V souboru `model.lst` vytváří seznam prvků, jejichž modely jsou v knihovně k dispozici. Zdrojové tvary modelů jsou uloženy v souborech s rozšířením `.dsf`. Program MODELPRO se spouští příkazem:

```
MODELPRO modely.dsf
```

`modely.dsf` značí soubor se zdrojovými tvary modelů (implicitně na cestě nastavené v konfiguraci simulátoru pro přístup ke knihovně s rozšířením `.dsf`).
Zcela novou zákaznickou knihovnu lze získat tak, že se vytvoří nová knihovna model.lib obsahující pouze modely základních prvků (`primitiv.lib`) a k ní se použitím programu MODELPRO připojí modely požadovaných prvků.
Ke tvorbě modelů slouží jednoduchý jazyk. Formát zdrojového tvaru modelu je:

```
:Jméno_typu       Jméno_knihovny       Počet_pinů
{ zapojení prvků }
%
```

Jméno_prvku je jméno, které se musí shodovat se jménem v knihovně OrCAD/SDT,

Jméno_knihovny je jméno knihovny, které se uvádí ve formátu EDIF a při generování seznamu spojů programem NETLIST odpovídá knihovně použité při kreslení schématu.

Počet_pinů udává počet vývodů prvku, tj. např. u pouzdra se 14 vývody 14.

Tělo modelu prvku je tvořeno popisem zapojení základních prvků (primitiv) tvaru:

```
Typ_primitiva(vstupy; výstupy; zpoždění);
```

Typ_primitiva je jméno některého ze základních prvků, *vstupy* značí seznam vstupů základního prvku, oddělených čárkami, *výstupy* značí seznam výstupů základního prvku, oddělených čárkami, *zpoždění* značí seznam časových charakteristik základního prvku.

Na místě vstupu může být číslo vývodu modelovaného prvku, označení vnitřního uzlu, logického uzlu, logická 0 (ZERO) nebo logická 1 (ONE). Na místě výstupu může být číslo vývodu prvku, označení vnitřního uzlu nebo označení logického uzlu. Pro označování platí konvence:

```
P<číslo>      vývod prvku,
N<číslo>      vnitřní uzel,
L<číslo>      logický uzel.
```

Logické uzly představují výstupy tzv. *logických hradel* (s nulovým zpožděním). Neplánují se pro ně žádné události, nýbrž vypočítaná hodnota je okamžitě k dispozici. Proto na rozdíl od obyčejných hradel závisí u logických hradel na pořadí zápisu. Z tohoto důvodu je třeba dodržovat zásadu, že nepoužijeme logický uzel jako vstup dříve, než byla definována jeho hodnota na výstupu logického hradla. Výhoda logických hradel spočívá v tom, že jsou z hlediska simulace efektivnější.

Zpoždění se zadává v závislosti na počtu časových údajů pro daný typ základního prvku jako dvojice seznamů oddělených čárkami — prvý pro minimální, druhý pro maximální hodnoty.

Mezi *základní prvky* patří hradla AND, NAND, OR, NOR, XOR, XNOR, INV (invertor) a BUF (buffer). Hradla kromě INV a BUF mají 2 až 16 vstupů a jsou charakterizována zpožděním TPLH a TPHL. Logická hradla jsou typu LAND, LNAND, LOR, LNOR, LXOR, LXNOR a LINV. K dispozici jsou i modely základních sekvenčních prvků, např. klopného obvodu typu D (DFFPC), který je charakterizován 10-ti časovými parametry (zpoždění hodiny -¿ výstup, předstih a přesah na vstupu D, minimální šířka hodinového impulsu, zpoždění nastavení, nulování -¿ výstup, minimální šířka nastavovacího nebo nulovacího impulsu, minimální odstup nastavení nebo nulování vzhledem k hodinám).

Kromě základních prvků lze v popisu použít *příkaz SET*, který slouží k určení síly signálu. Formát příkazu je:

```
SET(síla);
```

kde *síla* = {DRIVE, RHIGH, RLOW}.
Implicitní hodnota je DRIVE a znamená, že signály na výstupu mají sílu odpovídající buzení (drive). Použití příkazu SET značí, že od daného místa se síla výstupních signálů určuje podle parametru tohoto příkazu. Hodnota RHIGH značí, že pokud má výstup prvku hodnotu 1, bude chápána jako odporová (resistive) - hodnota 0 zůstává se sílou buzení. Pro hodnotu RLOW je situace opačná. Těchto prostředků lze s výhodou využít např. k modelování montážních funkcí.
Pro ilustraci uvedeme modely prvků 7400 a 7474. Měly by tvar:

```
:7400                   TTL      14
   NAND(P1,P2;P3;22,15,25,18);
   NAND(P4,P5;P6;22,15,25,18);
   NAND(P10,P9;P8;22,15,25,18);
   NAND(P13,P12;P11;22,15,25,18);
%
:7474                   TTL      14
   DFFPC(P2,P3,P4,P1;P5,P6;25,40,20,5,30,37,25,40,30,
         0,28,43,20,5,30,37,28,43,30,0);
   DFFPC(P12,P11,P10,P13;P9,P8;25,40,20,5,30,37,25,40,
         30,0,28,43,20,5,30,37,28,43,30,0);
%
```

8.7 Simulace na úrovni meziregistrových přenosů

Ve světě existuje řada jazyků, které lze označit za jazyky úrovně meziregistrových přenosů. V některých případech vznikly z nějakého univerzálního vyššího programovacího jazyka zavedením jazykových prostředků, usnadňujících popis číslicových zařízení.

Program, zapsaný v jazyce meziregistrových přenosů, má zpravidla podobnou strukturu jako programy ve vyšších programovacích jazycích: můžeme rozlišit deklarační a příkazovou část. V *deklarační části* jsou deklarovány základní strukturní prvky. Uvádí se jejich typ, jméno a dimenze. Jednotlivé jazyky se odlišují zejména typy prvků, které lze deklarovat, dimenzemi a způsobem indexace. Typickým příkladem je deklarace paměťových prvků typu registr nebo paměť. Např. deklarace:

```
register IR[15:0];
```

může deklarovat registr, kterým modelujeme instrukční registr. Navíc lze často označit část deklarované struktury novým jménem. Má-li náš instrukční registr v bitech 15 až 12 operační kód a v bitech 11 až 0 je adresa operandu, mohlo by mít přejmenování tvar:

```
subregister OPK = IR[15:12], ADR = IR[11:0];
```

Podobně lze často skupiny prvků řetězit a takto vytvořenou skupinu nově pojmenovat.

Kromě deklarací paměťových prvků se deklarují obvykle výstupy kombinačních obvodů jako propojení paměťových prvků s případnými transformacemi hodnot. Pro tyto účely se používá nejčastěji klíčové slovo*terminal*. Deklarace tvaru:

```
register A[7:0], B[7:0];
terminal T = A + B;
```

může reprezentovat výstup sčítačky, která sčítá hodnoty registrů A a B.

Různé jazyky umožňují deklarovat ještě další typy prvků. Deklarace neposkytují žádné informace týkající se realizace, neboť jsou nepodstatné na úrovni abstrakce odpovídající úrovni meziregistrových přenosů.

Zatímco deklarační část zachycuje strukturu popisovaného systému, funkce a algoritmus činnosti jsou vyjádřeny v *příkazové části*. V příkazech se vyskytují operace, které určují základní funkční operace, realizované v popisovaném zařízení. Mohou to být např. operace logické, aritmetické, posuvy a rotace apod. Pro jazyky meziregistrových přenosů je charakteristické, že operace jsou definovány nad vektory. Typickým příkazem je *přenos mezi registry*. Jestliže A značí registr a B další registr nebo výstup kombinačního obvodu, potom operaci přenosu hodnoty z B do registru A můžeme zapsat příkazem:

```
A <- B
```

Operátor `<-` značí přenos hodnoty do registru. Napravo od operátoru přenosu může zpravidla být výraz popisující trasformace při přenosu. Příkazy přenosu se u typických jazyků meziregistrových přenosů odlišují od příkazů spojení, které popisují propojení v kombinačních obvodech.

Již jsme uvedli, že na úrovni meziregistrových přenosů lze rozlišit řídicí signály. Při popisu funkce se tato skutečnost projevuje používáním různých podmíněných příkazů. Syntax se značně liší, ale ve většině jazyků jsou k dispozici příkazy odpovídající běžnému konstruktu IF-THEN.

Pro popis funkce je důležitý vztah posloupnosti příkazů příkazové části k posloupnosti provádění operací. Z tohoto hlediska lze rozlišit jazyky:

- neprocedurální
- procedurální

U *neprocedurálního jazyka* není pořadí, v němž jsou příkazy zapsány, podstatné. Výsledek je vždy stejný. Musí existovat mechanismus, který pořadí provádění určí. Jednou z možností je opatření každého příkazu návěštím ve tvaru nějaké podmínky, která rozhodne o tom, zda má být příkaz vykonán. Tento způsob používá např. jazyk *CDL* (*Computer Description Language*). Návěští však může mít také tvar pojmenovaného stavu a provádějí se příkazy odpovídající aktuálnímu stavu. Tento způsob se používá u jazyků vycházejících z pojetí modelovaného zařízení jako konečného automatu. Typickým příkladem je jazyk *DDL* (*Digital Design Language*). Všechny příkazy, jejichž vykonání je povoleno hodnotou návěští, resp. které odpovídají aktuálnímu stavu, jsou provedeny současně (z hlediska modelového času). Pořadí provádění příkazů může být také řízeno událostmi, podobně jako vyhodnocování prvků u logických simulátorů řízených událostmi.

Naproti tomu u *procedurálních jazyků* pořadí zápisu příkazů určuje i pořadí provádění. Mají-li být některé příkazy prováděny současně, musí být tato skutečnost syntakticky označena.

Při simulaci na úrovni meziregistrových přenosů se používají nejčastěji dvou- nebo tříhodnotové modely signálů a jednotkové zpoždění. Simulátory mohou být opět implementovány jako kompilované nebo interpretační.

Mezi typické představitele jazyků meziregistrových přenosů kromě již zmíněných *CDL* a *DDL* patří jazyky *AHPL* (*A Hardware Programming Language*) a *ISP* (*Instruction Set Procssor*). Jde o jazyky, které vznikly již na přelomu 60. a 70. let a od té doby zaznamenaly řadu modifikací. Jejich charakteristiku lze nalézt v [Neuschl]. Zde si pro bližší představu o tvaru programu v jazyce meziregistrových přenosů uvedeme pouze jednoduchý příklad.

Uvažujme obvod, který vytváří sériovým způsobem doplněk čísla, uloženého v nějakém registru R. Podstata algoritmu spočívá v tom, že jednotlivé bity registru R se kopírují nebo negují. Začíná se nejméně významným bitem registru$R$ a bity se kopírují tak dlouho, dokud se nenarazí na prvý nenulový bit. Další bity se potom již negují. Předpokládejme použití osmibitového posuvného registru R, klopného obvodu S indikujícího, zda se má bit kopírovat nebo negovat, a potřebných kombinačních obvodů. Řídicí část zahrnuje hodiny H, čítač posuvů C a stavový klopný obvod T, který je ovládán signálem P, spouštějícím převod.
Program v jazyce CDL by mohl mít tvar:

```
REGISTER, R(0-7), S, C(2-0), T
SWITCH, P(ZAP, VYP)
CLOCK, H
/P(ZAP)/ T=1, C=0, S=0
/T*H/ IF (S .EQ. 0) THEN (S=R(7), R=R(7)-R(0-6))
                    ELSE (R=R(7)'-R(0-6)),
      IF (C .EQ. 7) THEN (T=0) ELSE (C=C .CNTUP.)
END
```

Program je dostatečně srozumitelný. V deklarační části je ukázána možnost různého indexování registrů. Vstupní signál P pro spuštění převodu je modelován spínačem s polohami *ZAP* a *VYP*. V příkazové části je každý příkaz opatřen návěštím (výraz mezi lomítky). Operátor - značí zřetězení a je použit k popisu rotace, resp. rotace s negací (operátor $'$) obsahu registru R.

Při simulaci se vyhodnocují návěští a provádějí příkazy s aktivním návěštím (s hodnotou 1). Každé vyhodnocení všech návěští se nazývá *cyklus návěští*. Důležité je, že všechny příkazy jsou v rámci cyklu návěští zpracovány kvaziparalelně, tj. nové hodnoty jsou nastaveny vždy až na závěr cyklu návěští. Proto má jazyk neprocedurální charakter.

Ve srovnání s příkazy pro řízení simulace na úrovni logických členů prostředky pro řízení simulace na úrovni meziregistrových přenosů odrážejí vyšší abstrakci pohledu na modelovaný systém. Příklad lze nalézt v [**?**].

Jazyky meziregistrových přenosů mají velkou výhodu ve srozumitelnosti popisu. Jejich nevýhodou je, že jsou použitelné pouze pro úroveň meziregistrových přenosů, neboť používají strukturní prvky a operátory typické právě pro tuto úroveň.

Vývoj moderních jazyků pro logickou simulaci je charakteristický snahou pokrýt co největší škálu úrovní popisu systému, pro něž by byl použitelný. V současnosti se stává standardem, který tento požadavek do značné míry splňuje, jazyk *VHDL*.

8.8 Charakteristika jazyka VHDL

Jazyk *VHDL* (*VHSIC Hardware Description Language*) vznikl v USA v rámci stejnojmenného programu, který navazoval na program *VHSIC* (*Very High Speed Integrated Circuits*), který byl zaměřen na výzkum a vývoj technologie vysoce rychlých integrovaných obvodů. Již v počátcích řešení tohoto programu se totiž ukázala potřeba standardního jazyka pro popis obvodů, který by umožňoval snadnou komunikaci návrhových dat. Proto byly zformulovány požadavky na nový jazyk s přihlédnutím k provedené analýze několika používaných jazyků. Jedním z požadavků bylo, aby jazyk co nejvíce používal konstrukcí programovacího jazyka Ada. Na návrhu a implementaci se podíleli pracovníci firem Intermetrics, IBM a Texas Instruments, což spolu se skutečností, že program byl financován ministerstvem

obrany USA, přispělo k jeho značnému rozšíření. Jazyk VHDL byl přijat jako standard ministerstva obrany USA a později i jako standard IEEE [35].

Základním pojmem jazyka VHDL je *návrhová entita*. Modeluje číslicový systém libovolné složitosti. Může jít o hradlo, klopný obvod, ale i počítačový systém. Každá návrhová entita je charakterizována svým *rozhraním* a jedním nebo několika alternativními *těly*.

Rozhraní obsahuje definice společné všem tělům. Část z těchto definic určuje vnější pohled na entitu tím, že specifikuje komunikační kanály, jejichž prostřednictvím entita komunikuje s okolím.

Každé tělo popisuje alternativní pohled na návrhovou entitu. Např. jedno tělo může obsahovat popis chování, další architektonický popis odpovídající činnosti na úrovni meziregistrových přenosů, jiná těla různé implementace této návrhové entity.

Jazyk VHDL je poměrně složitý. V dalším se omezíme pouze na některé jeho rysy, které budeme ilustrovat na velice jednoduchém příkladě jednobitové sčítačky. Složitější příklad s podrobnějším výkladem lze nalézt v [**?**].
Rozhraní sčítačky v jazyce VHDL by mohlo vypadat takto:

```
entity  Full_Adder  is
     port  (X, Y   : in Bit;
     Cin  :  in  Bit = '0';
     Sum  :  out Bit;
     Cout :  out Bit );
end;
```

Rozhraní entity zahrnuje především informace viditelné zvnějšku. Jsou to vstup/výstupní *porty* definující kanály, kterými entita komunikuje s okolím, a tzv. *generické parametry* v případě, že entita je generická, tj. když se některý parametr určující její strukturu, chování nebo prostředí může lišit u jednotlivých výskytů entity.

V našem případě máme definovány čtyři porty. Definice portu obsahuje kromě jména také jeho režim a typ. Režim určuje směr přenášené informace. U nás jsou porty X, Y a *Cin* vstupní a *Sum* a *Cout* výstupní. Ve všech případech jsou porty typu Bit, což je předdefinovaný typ jazyka.
Jazyk VHDL poskytuje dva *typy těla* entity:

- architektonické tělo,
- konfigurační tělo.

Architektonické tělo vyjadřuje vztah mezi hodnotami na vstupních a výstupních portech návrhové entity. Tento vztah může být popsán buď pomocí příkazů vyjadřujících transformace vstupních signálů nebo příkazů, explicitně popisujících propojení prvků a portů. *Konfigurační tělo* obsahuje globální informaci vyjadřující vztah mezi různými entitami (např. které návrhové entity modelují prvky použité v architektonickém popisu a jak se transformují globální signály mezi entitami).
V rámci architektonického těla lze použít dva základní *tvary popisu*:

- *strukturní* — popisuje především propojení prvků,
- *chování*.

Při popisu chování se mohou uplatnit *dva styly*:

- *popis toku dat* — popisuje prováděné transformace dat pomocí příkazů, odpovídajících jazykům meziregistrových přenosů,

- *popis algoritmu* — popisuje prováděné transformace prostřednictvím vyjádření algoritmu určení výstupních odezev na změny na vstupech.

Díky společné sémantice lze v rámci jednoho architektonického těla způsoby popisu libovolně kombinovat.

Všechny tři uvedené formy popisu budeme ilustrovat na architektonickém těle našeho příkladu. Pokud bychom předpokládali strukturu sčítačky podle *obr.* **??**, vypadal by čistě strukturní popis takto:

```
architecture Structural of Ful_Adder is
signal S1, S2, S3: Bit;            --vnitřní signály
component XOR_G                    --deklarace prvků
  port (X1, X2: in Bit; XO1: out Bit);
end component;
component AND_G
  port (A1, A2: in Bit; AO1: out Bit);
end component;
component OR_G
  port (O1, O2: in Bit; OU1: out Bit);
end component;
     -- konfigurační specifikace
for all: XOR_G use entity XOR_Gate(Behave);
     -- je použito implicitní mapování portu (stejná jména a typy)
for all: AND_G use entity AND_Gate(Behave);
for all: OR_G use entity OR_Gate(Behave);
begin
   XOR1: XOR_G port map (X,Y,S1);   -- použito poziční přiřazení portů
   XOR2: XOR_G port map (S1,Cin,Sum);
   AND1: AND_G port map (X,Y,S3);
   AND2: AND_G port map (S1,Cin,S2);
   OR1 : OR_G  port map (O1=>S2,O2=>S3,OU1=>Cout); -- expl. přiřazení portů
end Structural;
```

Strukturní popis sestává ze tří základních částí. Prvou je deklarace typů prvků, tj. typů entit použitých v popisu, druhou konfigurační specifikace udávající, které konkrétní entity z knihovny a která jejich těla mají být použity pro modelování potřebných typů prvků. Třetí část popisuje propojení prvků. Entity komunikují prostřednictvím *signálů*. Jazyk VHDL rozlišuje mezi proměnnou a signálem. Proměnná je zde chápána obvyklým způsobem, signály mohou být obdobných typů jako proměnné, avšak se změnou hodnoty je spojena událost, která má za následek šíření nové hodnoty na patřičná místa.

Při popisu toku dat se používají tzv. *souběžné příkazy* (*concurrent statements*). Každý takový příkaz se provádí při událostech signálů s ním spojených (např. vyskytujících se ve výrazu napravo od operátoru přiřazení u přiřazovacího příkazu). Jako souběžné se označují proto, že při změně signálu, který se vyskytuje v několika souběžných příkazech, jsou tyto příkazy zpracovány souběžně (nové hodnoty se nastavují až po vyhodnocení všech souběžných příkazů). Jazyk je proto v této části neprocedurální.
Pro náš příklad by byl popis toku dat jednoduchý:

```
architecture DataFlow of Full_Adder is
signal S1, S2: Bit;        -- vnitřní signály
```

```
begin
  S1  <= X xor Y;          -- použito implicitní zpoždění
  S2  <= S1 and Cin;       -- provádí se při změně S1 nebo Cin
  Sum <= S1 xor Cin;
  Cout <= S2 or (X and Y); -- provádí se při změně X, Y nebo S2
end DataFlow;
```

V našem příkladě jsme použili souběžné příkazy přiřazení hodnoty signálu. Přiřazení je vždy provedeno s nějakým zpožděním, které může být zadáno explicitně nebo je použito implicitní zpoždění, které je nekonečně malé, ale současně nenulové (přiřazení se provede v aktuálním modelovém čase, ale až po vyhodnocení všech příkazů). Souběžné příkazy mohou mít i tvar podmíněného přiřazení, přepínače apod. Tento styl popisu odpovídá způsobu používanému u jazyků meziregistrových přenosů s tím rozdílem, že tady všechny registry a terminály jsou deklarovány jako signály např. předdefinovaného typu `BIT_VECTOR`.

Čisté chování se v jazyce VHDL popisuje *příkazem procesu*. Obsahuje deklarační a příkazovou část, která zahrnuje pouze *sekvenční příkazy* vyjadřující algoritmus, který specifikuje odezvy na výstupech v závislosti na změnách na vstupech bez jakýchkoliv informací a předpokladů o struktuře. U každého procesu se uvádí seznam signálů, na které je proces citlivý. Sám příkaz procesu je jedním ze souběžných příkazů, a proto se příkazy jeho těla provádějí vždy při změně některého z těchto signálů.

Náš příklad je velice jednoduchý, nicméně i zde můžeme plně abstrahovat od vnitřní struktury a definovat proces, ve kterém použijeme operaci aritmetického součtu. Ta je však definována pro typ INTEGER, proto je potřeba použít vhodnou konverzní funkci. Jazyk VHDL převzal z jazyka Ada koncepci práce s *moduly* (*package*), které obsahují určitý logický celek dat a akcí. Předpokládejme, že máme potřebné konverzní funkce v modulu Convert_Pack. Zpřístupnit je lze použitím klauzule *use*. Současně ukážeme použití příkazu bloku, který kromě entit představuje hlavní mechanismus strukturalizace popisu. Může mít záhlaví, ve kterém lze použít klauzule port a port map, s nimiž jsme se již setkali u strukturního popisu. My použijeme rozhraní bloku právě ke konverzi typu Bit na INTEGER a naopak. Algoritmický popis sčítačky by mohl mít tvar:

```
architecture Functional of Full_Adder  is
use Convert_Pack.all -- zpřístupní vše v modulu
begin
  FF block
      port(A,B,C: in INTEGER; S,CO: out INTEGER);
           -- konverze typů se provádí na portech
      port map (A => Bin_to_Int(X), B => Bin_to_Int(Y),
                C => Bin_to_Int(Cin), Int_to_Bin(S)=>Sum,
                Int_to_Bin(CO) => Cout);
      process (A, B, C) -- seznam signálů, na něž je proces citlivý
      variable Int1: INTEGER;    -- obyčejná proměnná
      begin
        Int1 := A + B + C;       -- sekvenční příkazy
        S    <= Int1 mod 2;
        CO   <= Int1/2;
      end process;
  end block FF;
end Functional;
```

V našem příkladu jsme v rámci těla entity použili vždy jen jeden styl popisu. Syntax

jazyka však umožňuje všechny tři styly používat současně v rámci jednoho těla. To může být vhodné pro složitější systémy, kdy máme již například navrženu část systému (známe strukturu), zatímco o zbývající části máme představu pouze na úrovni chování nebo toku dat.

I malá část prostředků jazyka VHDL, kterou jsme si ukázali, svědčí o tom, že jde o složitý jazyk. Díky své obecnosti je použitelný pro simulaci na různých úrovních (mixed-level). Všimněme si, že i když je předdefinován typ Bit, podporující dvouhodnotovou simulaci, lze nadefinovat i jiné typy pro vícehodnotovou simulaci. Ve srovnání se způsobem práce, který využívá strukturních modelů, jak jsme viděli v případě systému OrCAD/VST, se může zdát VHDL pro návrháře těžko zvládnutelný. Jeho modelovací schopnost je však nesrovnatelně vyšší a zpravidla jsou k dispozici knihovny s moduly, které modelování usnadňují.

Kapitola 9

Stručná referenční příručka SIMLIB

Tato část textu[1] shrnuje stručné informace o standardních třídách, objektech a funkcích obsažených v SIMLIB. Jsou vynechány všechny nepodstatné detaily implementace — aby se popis nemusel s dalším vývojem knihovny příliš měnit. Protože jde o příručku, používáme abecedního řazení položek.

9.1 Hierarchie tříd SIMLIB

(zjednodušeno)

```
SimObject
    aBlock
        Frict
        Graph
        Insv
        Integrator
        Lim
        Qntzr
        Status
            Blash
            Hyst
            Relay
    aCondition
        BoolCondition
    aQueue
        Queue
    Entity
        Process
        Event
    Facility
    Histogram
    Stat
```

[1] Aktualizováno 1998

```
Store
TStat
Variable
```

9.2 Třída aBlock

Bázovou třídou pro všechny třídy objektů se spojitým chováním a jedním výstupem je abstraktní třída aBlock. Tato třída definuje podstatné společné vlastnosti pro všechny spojité bloky, především získání hodnoty bloku metodou Value. Deklaruje též virtuální metody pro vyhodnocení bloku. Bloky lze navzájem propojovat.

Value double Value()=0;

Metoda Value je určena pro získání hodnoty výstupu bloku. Je definována pro všechny třídy odvozené ze třídy aBlock.

9.3 Třída aCondition

Tato třída je abstraktní bázovou třídou pro všechny třídy, popisující stavové podmínky. Definuje základní operace, nutné pro detekci změn stavových podmínek a pro vyvolání příslušných stavových událostí.

9.4 Třída aQueue

Třída aQueue představuje abstraktní frontu objektů. Slouží jako bázová třída pro fronty s různě definovaným uspořádáním. Umožňuje uchovávání objektů tříd, odvozených z třídy Entity. Platí pravidlo, že jeden objekt může být v jednom okamžiku nejvýše v jedné frontě.

Clear

virtual void Clear();

Tato metoda zruší všechny objekty ve frontě a inicializuje frontu.

Empty

int Empty();

Metoda Empty testuje, je-li fronta prázdná.

Get

virtual Entity *Get(Entity *e)=0;

virtual Entity *Get()=0;

Metoda Get vyjme zadaný, resp. frontou definovaný (obvykle první) prvek z fronty.

GetFirst,GetLast

Entity *GetFirst();

Entity *GetLast();

Metody GetFirst a GetLast vyjmou první, resp. poslední prvek z fronty.

Insert

virtual void Insert(Entity *e)=0;

Metoda Insert zařadí prvek e do fronty podle uspořádání, které definuje fronta. Tato metoda je nejčastěji používána pro vstup objektu do fronty.

Length

unsigned Length();

Metoda Length vrací počet objektů, zařazených ve frontě.

9.5 Třída Blash - vůle v převodech

Tato třída popisuje stavové bloky se spojitým chováním a charakteristikou, která je uvedena na obrázku:

Konstruktor

Blash(Input x, double p1, double p2, double tga);

Konstruktor vytvoří blok a inicializuje jeho parametry. Parametr tga je tangentou úhlu alfa.

Value

double Value()=0;

Metoda Value vrací hodnotu výstupu bloku, tj. stav bloku.

9.6 Třída BoolCondition - stavová podmínka

Tato abstraktní třída popisuje chování stavových podmínek, které reagují (voláním metody Action) na změnu hodnoty vstupu (metoda Test), a to buď z hodnoty nula (FALSE) na hodnotu různou od nuly (TRUE), nebo obráceně. Pro detekci změny podmínky je použita metoda půlení intervalu. Metody Test a Action nejsou v této třídě definovány, třída se používá jako bázová pro uživatelské třídy, které tyto metody definují.

Action

virtual void Action()=0;

Metoda popisuje reakci na změnu pravdivostní hodnoty podmínky. Tato metoda musí být definována v odvozené třídě.

Mode

enum DetectionMode DetectUP, DetectDOWN, DetectALL

void Mode(DetectionMode m);

Metoda Mode umožňuje rozlišení různých případů změny podmínky. DetectUP způsobí detekci změny pouze z hodnoty FALSE na hodnotu TRUE, DetectDOWN detekuje opačnou změnu. DetectALL zapne detekci obou změn.

Test

virtual int Test()=0;

Metoda Test slouží pro zápis vstupu podmínky. Vyhodnocuje se průběžně při simulaci a změna pravdivostní hodnoty způsobí vyvolání metody Action. Tato metoda musí být definována v odvozené třídě.

Příklad

Integrator i1(x); class MyCondition : BoolCondition int Test() return i1.Value()¿10; void Action() x = -x.Value(); public: MyCondition() Mode(DetectUP); ;

9.7 Třída Entity

Tato třída je abstraktní bázovou třídou pro všechny prvky modelu, které mají popsáno svoje chování událostmi nebo procesy. Definuje chování společné událostem i procesům, především jde o zařazování do front a plánování akcí. Entita může být zařazena pouze v jedné frontě (nemůže být na dvou místech současně).

Activate

virtual void Activate();

virtual void Activate(double t);

Plánuje aktivaci entity na čas t (implicitně ihned).

Head

aQueue *Head();

Vrací ukazatel na frontu, ve které je objekt zařazen.

Idle

int Idle();

Test, je-li naplánována reaktivace entity. V případě, že není aktivační záznam entity v kalendáři, vrací nenulovou hodnotu (TRUE). To znamená, že entita čeká v některé frontě, nebo čeká na WaitUntil podmínku.

Into

void Into(aQueue *Q);

Zařadí objekt do fronty Q, uspořádání je definováno frontou.

Out

void Out();

Vyjme prvek z fronty, je-li v některé zařazen.

Passivate

virtual void Passivate();

Deaktivuje entitu, tj. vyřadí její aktivační záznam z kalendáře.

Priority

tPriority Priority;

Priorita entity. Používá se při řešení problémů s plánováním více entit na jeden a tentýž okamžik. Typ tPriority má rozsah od nuly do 255. Nejnižší priorita je nulová.

9.8 Třída Event

Tato abstraktní třída popisuje vlastnosti entit, jejichž chování v čase je popsáno událostmi (tj. akcemi, během jejichž provádění nedochází ke změně modelového času - nelze je přerušit). Události jsou ekvivalentní procesům, které nepoužívají přerušovací příkazy. Jejich implementace je efektivnější než u procesů. Používají se především při periodickém sledování stavu modelu, jako generátory objektů nebo pro modelování jednorázových dějů. Dědí vlastnosti třídy Entity.

Activate

virtual void Activate();

virtual void Activate(double t);

Plánuje aktivaci události na čas t (implicitně ihned). Aktivací události se rozumí provedení metody Behavior.

Behavior

virtual void Behavior()=0;

Výkonná funkce, definující akce události. Uživatel ji musí definovat v odvozené třídě. Tato funkce je nepřerušitelná.

Konstruktor

```
Event(tPriority p=DEFAULT_PRIORITY);
```

Vytvoří událost se zadanou prioritou. Implicitní priorita události je rovna nule (nejnižší).

Priority

tPriority Priority;

Priorita události. Používá se při řešení problémů s plánováním více událostí na jeden a tentýž okamžik. Typ tPriority má rozsah od nuly do 255. Nejnižší priorita je nulová.

Schedule

void Schedule(double t);

Plánuje aktivaci události na dobu t. Je ekvivalentní metodě Activate(t).

Terminate

void Terminate();

Zrušení aktivní události.

9.9 Třída Facility

Tato třída představuje zařízení s výlučným přístupem. To znamená, že toto zařízení smí být obsazeno pouze jednou entitou. Zařízení má dvě fronty - frontu vstupní Q1 a frontu přerušené obsluhy Q2.

Busy

int Busy();

Metoda vrací nenulovou hodnotu (TRUE), je-li zařízení obsazeno.

Clear

virtual void Clear();

Inicializace zařízení. Zruší entity ve frontách a fronty, pokud byly vytvořeny dynamicky zařízením.

in - entita v obsluze

Entity *in;

Obsahuje ukazatel na entitu právě obsluhovanou zařízením nebo NULL v případě, že zařízení není obsazeno.

Konstruktor,Destruktor

Facility();

Facility(char *name);

Facility(Queue *queue1);

Facility(char *name, Queue *Queue1);

virtual Facility();

Inicializuje zařízení jménem name, je možné explicitně zadat vstupní frontu. Implicitně je zařízení volné.

Q1,Q2 - fronty u zařízení

Queue *Q1;

Queue *Q2;

Vstupní fronta a fronta entit, jejichž obsluha byla přerušena.

QueueLen

unsigned QueueLen();

Délka vstupní fronty zařízení. Tento údaj je použitelný pro omezení délky fronty.

QueueIn

virtual void QueueIn(Entity *e, tPriority sp);

Metoda QueueIn zařadí entitu e do vstupní fronty Q1 podle priorit v tomto pořadí:

(1) priorita obsluhy (největší je první) (2) priorita entity (3) FIFO

Podle stejných pravidel je řazena i fronta Q2.

Release

virtual void Release(Entity *e);

Uvolní zařízení. Je chybou, provede-li se uvolnění neobsazeného zařízení. Je chybou, uvolňuje-li zařízení jiná entita než ta, která je obsadila.

Seize

virtual void Seize(Entity *e, tPriority sp);

Obsazení zařízení entitou e s prioritou obsluhy sp. Je-li zařízení již obsazeno entitou s menší prioritou obsluhy než je sp, potom přeruší obsluhu této entity, zařadí ji do fronty přerušené obsluhy a obsadí zařízení entitou e. Jinak vstoupí do vstupní fronty zařízení metodou QueueIn. Entity, čekající ve frontě přerušených mají při stejné prioritě obsluhy přednost před entitami, čekajícími ve vstupní frontě. Uspořádání front je popsáno v popisu metody QueueIn.

tstat - statistika zařízení

TStat tstat;

Tato časová statistika umožňuje výpočet průměrného využití zařízení.

SetName, SetQueue

void SetName(char *name);

void SetQueue(Queue *queue1);

Explicitní nastavení jména, resp. vstupní fronty. Používá se především při inicializaci polí zařízení.

Output

virtual void Output();

Operace tisku stavu a statistik zařízení včetně statistik front.

9.10 Třída Frict - tření

Třída Frict implementuje nestavové bloky s charakteristikou tření:

Konstruktor

Frict(Input x, double r1, double r2, double tga);

Value

double Value()=0;

Metoda Value vrací hodnotu výstupu bloku.

9.11 Třída Graph

Tato třída je určena pro záznam hodnot získaných při spojité simulaci do výstupního souboru. Výstup hodnot probíhá automaticky, soubor musí být otevřen voláním funkce OpenOutputFile v popisu experimentu. Tento soubor lze po skončení simulačních experimentů prohlížet výstupním editorem (program OE).

Konstruktor

Graph(char *name, Input x, double step);

Parametrem konstruktoru je jméno grafu, tj. jméno ukládaného průběhu, vstupní objekt, jehož hodnota bude zaznamenávána a interval ukládání vstupních hodnot do souboru.

Příklad:

Integrator Teplota(x,20); Graph G("Teplota",Teplota,0.01);

int main() OpenOutputFile("ZAZNAM.OUT"); Init(0,10); Run();

9.12 Třída Histogram

Objekty třídy Histogram sledují četnost vstupních hodnot v zadaných intervalech. Histogram má tyto parametry:

name jméno histogramu

low dolní mez = začátek prvního intervalu

step krok = šířka intervalů

count počet intervalů

Kromě toho se sledují hodnoty, které jsou menší, než dolní mez prvního intervalu a větší, než horní mez posledního intervalu. Situaci ilustruje obrázek:

Clear

virtual void Clear();

Inicializace histogramu.

Konstruktor,Destruktor

Histogram();

Histogram(double low, double step, int count);

Histogram(char *name, double low, double step, int count);

Histogram();

Vytvoření a rušení histogramu se zadanými parametry.

operator () - záznam hodnoty

void operator ()(double x);

Tato operace zaznamená hodnotu x do histogramu.

operator [] - četnost v intervalu

unsigned operator [](unsigned i);

Tato operace vrací četnost v i-tém intervalu histogramu. Je-li index i mimo meze vrací nulu.

Output

virtual void Output();

Operace tisku histogramu do standardního textového výstupu. Tiskne se formou tabulky.

Příklad:

Histogram H("Histogram1",0,0.1,100); double x; ... H(x); // záznam hodnoty proměnné x H(5); // záznam hodnoty 5 ... H.Output(); // tisk histogramu H

9.13 Třída Hyst - hystereze

Hystereze je nelinearita s vnitřním stavem. Charakteristika hystereze je na obrázku:

Konstruktor

Hyst(Input x, double p1, double p2, double y1, double y2, double tga);

Vytvoří a inicializuje parametry bloku.

Value

double Value()=0;

Metoda Value vrací hodnotu výstupu bloku.

Příklad:

Hyst H(x,-1,1,-5,5,3.5); ... printf("výstup bloku hystereze =

9.14 Třída Input

Objekty této třídy se používají jako vstupy spojitých bloků. Objekt reprezentuje odkaz na spojitý blok.

Konstruktor

Input(aBlock *b);

Inicializuje objekt třídy Input odkazem na blok, zadaný jako parametr.

Value

double Value();

Vyhodnotí objekt na který se odkazuje.

Příklad:
class B1 : aBlock Input in; ... B1(Input i): in(i) ... ;

9.15 Třída Insv - necitlivost

Necitlivost je typická nestavová nelinearita s charakteristikou:
Konstruktor
Insv(Input x, double p1, double p2);
Insv(Input x, double p1, double p2, double tgalfa, double tgbeta);
Implicitně jsou úhly alfa a beta 45 stupňů.
Value
double Value()=0;
Metoda Value vrací hodnotu výstupu bloku.

9.16 Třída Integrator

Třída Integrator implementuje popis numerické integrace při spojité simulaci. Integrátor má definovány tři základní operace:
o inicializace, nastavení počáteční hodnoty
o nastavení hodnoty, tj. skoková změna stavu integrátoru
o numerická integrace (skrytá vlastnost objektu)
Init
void Init(double initvalue);
double operator = (double x)
Metoda Init nastaví inicializační hodnotu integrátoru. Tuto hodnotu integrátor vždy použije pro svou inicializaci před začátkem simulace.
Konstruktor
Integrator(Input x);
Integrator(Input x, double initvalue);
Vytvoření a inicializace integrátoru hodnotou initvalue nebo nulou.
Set
void Set(double value);
double operator = (double x)
Nastaví hodnotu integrátoru. Lze použít k provedení nespojité změny hodnoty integrátoru.
Operátor =
double operator = (double x)
Operátor přiřazení má dva různé významy podle místa použití. Při použití ve stavu INITIALIZATION (tj. mezi voláním funkcí Init a Run v popisu experimentu) nastaví inicializační hodnotu integrátoru (je ekvivalentní metodě Init). Použije-li se ve stavu SIMULATION (probíhá simulace), potom nastavuje hodnotu výstupu integrátoru (je ekvivalentní metodě Set).
Value
double Value();
Metoda Value vrací hodnotu výstupu integrátoru.
Příklad:
Integrator i1(x,1), i2(i1); // vytvoření a inicializace

class Udalost1 : Event void Behavior() i1 = 1.0; // nastavení hodnoty - skoková změna ... ;
main() Init(0,10); i1 = 1.0; // inicializace integrátoru ... Run();

9.17 Třída Lim - omezení

Objekty třídy Lim omezují vstupní hodnotu na zadaný interval od p1 do p2. Strmost charakteristiky je možno zadat jako poslední parametr formou tangenty úhlu alfa. Charakteristika omezení je na obrázku:

Konstruktor
Lim(Input x, double p1, double p2);
Lim(Input x, double p1, double p2, double tgalfa);
Value
double Value();
Metoda Value vrací hodnotu výstupu bloku.

9.18 Třída Process

Abstraktní třída Process popisuje vlastnosti objektů (entit), jejichž chování v čase je popsáno procesem (tím rozumíme proceduru, přerušitelnou některými příkazy). Používá se jako bázová třída pro třídy vytvářené uživatelem. Dědí vlastnosti třídy Entity.

Activate
void Activate(double t);
void Activate();
Plánuje aktivaci entity na modelový čas t, případně ihned. Aktivace znamená buď start popisu chování v případě nově vytvořené entity, nebo pokračování provádění popisu chování v případě, že proces již běžel (a byl například deaktivován příkazem Passivate).

Behavior
virtual void Behavior()=0;
Popis diskrétního chování procesu. Uživatel musí definovat tuto metodu ve třídě, která zdědí třídu Process. Metoda je přerušitelná příkazy Wait, WaitUntil, Seize, Enter, Passivate, Cancel.

Cancel
virtual void Cancel();
Ukončení procesu a zrušení objektu. Nemusí se uvádět v případě, že by šlo o poslední příkaz v metodě Behavior.

Enter
virtual void Enter(Store *s, tCapacity ReqCap);
Obsadí kapacitu ReqCap skladu s, je-li volná. Jinak vstoupí do vstupní fronty skladu s. Uvolnění skladu viz metoda Leave.

Konstruktor,Destruktor

```
Process(tPriority p=DEFAULT_PRIORITY);

virtual ~Process();
```

Vytvoří proces se zadanou prioritou. Implicitní priorita procesu je rovna nule (nejnižší). Destruktor zruší objekt (proces).

Leave

virtual void Leave(Store *s, tCapacity ReqCap);

Uvolní kapacitu ReqCap skladu s. Obsazení skladu viz metoda Enter.

Passivate

virtual void Passivate();

Vyřadí aktivační záznam procesu z kalendáře. Proces čeká až bude opět aktivován. Po aktivaci pokračuje od následujícího příkazu v popisu chování.

Priority

tPriority Priority;

Priorita procesu. Typ tPriority má rozsah od nuly do 255. Nejnižší priorita je nulová.

Release

virtual void Release(Facility *f);

Uvolní zařízení f. Je chybou, provede-li se uvolnění zařízení před jeho obsazením. Je chybou, uvolňuje-li zařízení jiný proces než ten, který je obsadil metodou Seize.

Seize

virtual void Seize(Facility *f);

virtual void Seize(Facility *f, tPriority PrioS);

Obsadí zařízení f, je-li volné. Je-li zařízení obsazeno entitou s menší prioritou obsluhy PrioS, potom přeruší obsluhu této entity, zařadí ji do fronty přerušené obsluhy a obsadí zařízení. Jinak vstoupí do vstupní fronty zařízení f. Entity, čekající ve frontě přerušených mají přednost před entitami, čekajícími ve vstupní frontě. Fronty jsou uspořádány podle priority obsluhy a podle priority entity. Viz též třída Facility.

Wait

virtual void Wait(aQueue *Q);

virtual void Wait(double DeltaTime);

Čekání entity ve frontě Q nebo čekání po dobu DeltaTime. Čekání po dobu DeltaTime znamená plánování reaktivace procesu na čas Time + DeltaTime.

WaitUntil

virtual void WaitUntil(int expr);

Obecné čekání na splnění podmínky expr. Parametr je neustále vyhodnocován a platí-li, proces pokračuje, jinak čeká. K vyhodnocování podmínky expr dochází vždy po skončení události nebo po přerušení procesu. Pokud se čeká na změnu nějaké proměnné, je třeba zajistit přerušení procesu po příkazu, který ji změní.

9.19 Třída Qntzr - kvantizátor

Kvantizátor provádí zaokrouhlení vstupní hodnoty na násobek kvantizačního kroku.

Konstruktor

Qntzr(Input x, double p);

Parametr p je kvantizační krok.

Value

double Value();

Metoda Value vrací hodnotu výstupu kvantizátoru.

9.20 Třída Queue

Třída Queue představuje prostředek pro uchování objektů tříd, odvozených z třídy Entity. Fronta je seřazena podle priorit entit Fronta udržuje statistiku počtu entit ve frontě a statistiku dob pobytu entit ve frontě.

Clear
virtual void Clear();
Zruší všechny entity ve frontě, inicializuje statistiky fronty.
Empty
int Empty();
Test, je-li fronta prázdná.
Get
Entity *Get(Entity *e);
Entity *Get();
Vyjme zadaný, resp. první prvek z fronty.
Insert
virtual void Insert(Entity *e);
Vloží zadanou entitu do fronty podle její priority.
Konstruktor, Destruktor
Queue();
Queue(char *name);
virtual Queue();
Vytvoří prázdnou frontu se jménem name. Destruktor zruší frontu, včetně zařazených objektů.
Length
unsigned Length();
Získá počet entit ve frontě. Je použitelné v případě modelování front s omezenou délkou.
Output
void Output();
Tato metoda vytiskne do standardního textového výstupního souboru statistiky a stav fronty.
SetName
void SetName(char *name);
Nastaví jméno fronty. Používá se při inicializaci polí front. Jméno slouží k identifikaci fronty ve výstupních datech při tisku metodou Output.
StatN, StatDT - statistiky fronty
TStat StatN;
Stat StatDT;
Statistika počtu prvků ve frontě a statistika doby přítomnosti entit ve frontě.

9.21 Třída Relay - relé

Charakteristika relé je uvedena na obrázku:
Konstruktor
Relay(Input x, double p1, double p2, double p3, double p4, double y1, double y2);
Value
double Value();
Metoda Value vrací hodnotu výstupu relé.

9.22 Třída SimObject

SimObject je základní bázovou třídou (kořenem) hierarchie tříd v SIMLIB. Specifikuje základní vlastnosti objektů modelu. Definuje metody pro vytvoření a zrušení objektu. Třída

SimObject je abstraktní bázovou třídou, tj. nelze vytvářet objekty této třídy.

9.23 Třída Stat

Tato třída slouží pro sběr statistických údajů. Uchovává následující informace vypočtené ze vstupních hodnot x:

double sx; // suma hodnot x double sx2; // suma čtverců hodnot x double min; // minimální hodnota double max; // maximální hodnota unsigned long n; // počet záznamů

Clear

virtual void Clear();

Inicializace statistiky.

Konstruktor,Destruktor

Stat(char *name);

Stat();

Inicializace a rušení objektu.

operator () - záznam hodnoty do statistiky

void operator ()(double x);

Záznam hodnoty x do statistiky.

Output

virtual void Output();

Operace tisku statistiky do standardního textového souboru.

Příklad:

Stat Statistika("Statistika"); ... Statistika(x); // záznam hodnoty x ... Statistika.Output(); // tisk statistiky

9.24 Třída Status - stavová proměnná

Třída popisuje vlastnosti stavových proměnných. Jsou z ní odvozeny třídy stavových bloků (např. relé,hystereze). Každý objekt této třídy má atribut, který definuje jeho stav.

Init

void Init(double initvalue);

Metoda Init nastaví inicializační hodnotu stavu objektu.

Konstruktor

Status(Input x);

Status(Input x, double initvalue);

Inicializace stavu objektu nulou nebo zadanou hodnotou initvalue.

operator =

double operator = (double x)

Operátor přiřazení má dva různé významy podle místa použití. Při použití ve stavu INITIALIZATION (tj. mezi voláním funkcí Init a Run v popisu experimentu) nastaví inicializační hodnotu (je ekvivalentní metodě Init). Použije-li se ve stavu SIMULATION (probíhá simulace), potom nastavuje hodnotu objektu (je ekvivalentní metodě Set).

Set

void Set(double value);

double operator = (double x)

Nastaví stav objektu. Lze použít kdekoli v popisu modelu k provedení nespojité změny hodnoty stavu.

Value

double Value();
Metoda Value vrací hodnotu výstupu objektu.

9.25 Třída Store

Tato třída představuje sdílený prostředek s určitou kapacitou (počtem míst). To znamená, že několik entit může současně používat část kapacity skladu. Pokud požadovaná kapacita není k dispozici, musí entita počkat ve vstupní frontě Q.

Capacity
tCapacity Capacity();
Vrací celkovou kapacitu skladu.
Clear
virtual void Clear();
Inicializace skladu, jeho vstupní fronty a statistik.
Empty
int Empty();
Metoda Empty vrací TRUE, je-li sklad prázdný.
Enter
virtual void Enter(Entity *e, tCapacity cap);
Entita e obsadí kapacitu cap skladu, je-li volná. Jinak vstoupí do vstupní fronty skladu voláním metody QueueIn.
Free
tCapacity Free();
Vrací volnou kapacitu skladu.
Full
int Full();
Metoda Full vrací TRUE, je-li kapacita skladu zcela obsazena.
Konstruktory,Destruktor
Store();
Store(tCapacity capacity);
Store(char *name, tCapacity capacity);
Store(tCapacity capacity, Queue *queue);
Store(char *name, tCapacity capacity, Queue *queue);
virtual Store();
Konstruktory umožňují inicializaci skladu, nastavení jeho kapacity a vstupní fronty. Implicitní kapacita je 1, pokud nezadáme frontu, sklad si ji vytvoří. Typ tCapacity je celočíselný (unsigned long).
Leave
virtual void Leave(tCapacity cap);
Uvolní požadovanou kapacitu cap skladu. Je chybou, žádá-li se o uvolnění větší kapacity, než je obsazeno.
Output
virtual void Output();
Tiskne stav skladu, jeho fronty a statistiky.
Q - vstupní fronta
Queue *Q;
Ukazatel na vstupní frontu. Sklad si vytvoří vstupní frontu dynamicky, je-li to zapotřebí.
QueueIn
virtual int QueueIn(Entity *e);

Operace vstupu do fronty u skladu. Zařadí do fronty podle priority entity, při shodě priorit podle strategie FIFO.

QueueLen
unsigned QueueLen();
Zjistí délku fronty u skladu.
SetName,SetCapacity,SetQueue
void SetName(char *name);
void SetCapacity(tCapacity capacity);
void SetQueue(Queue *queue);
Tyto operace nastavují parametry skladu, používají se při inicializaci polí skladů.
tstat - statistika skladu
TStat tstat;
Slouží pro výpočet průměrně obsazené kapacity.
Used
tCapacity Used();
Vrací použitou kapacitu skladu.

9.26 Třída TStat

Tato třída slouží pro sběr statistických údajů. Objekty této třídy sledují časový průběh vstupní veličiny. To umožňuje výpočet průměrných hodnot veličin v zadaném časovém intervalu od inicializace statistiky do okamžiku výstupu. Časová statistika uchovává tyto údaje vypočtené ze vstupních hodnot x:

```
double sxt;        // suma hodnot x*dtime
double sx2t;       // suma čtverců hodnot (x^2)*dtime
double min;        // minimální hodnota x
double max;        // maximální hodnota x
double t0;         // čas inicializace
double tl;         // čas poslední operace
double xl;         // poslední zaznamenaná hodnota x
unsigned long n;   // počet záznamů do statistiky
```

Clear
virtual void Clear();
Inicializace statistiky.
Konstruktor,Destruktor
TStat();
TStat(char *name);
TStat();
Inicializace a rušení objektu.
operator () - záznam hodnoty do statistiky
void operator ()(double x);
Tato operace provede záznam hodnoty x do statistiky.
Output
virtual void Output();
Operace tisku časové statistiky.
Příklad:
TStat TS("TS"); ... TS(x); // záznam hodnoty v čase Time ... TS.Output(); // výstup

9.27 Standardní objekty a proměnné

Součástí každého modelu jsou standardní proměnné, které jsou deklarovány implicitně. Identifikátorů standardních proměnných nelze použít pro označení jiných objektů modelu. Tyto proměnné plní zvláštní úlohu při řízení simulace. Jejich hodnoty nastavují funkce Init, Run, SetStep a SetAccuracy.

AbsoluteError

```
double AbsoluteError;
```

Proměnná AbsoluteError obsahuje požadovanou maximální absolutní chybu numerické integrace. Je nastavována funkcí SetAccuracy. Implicitní hodnota je nulová, to znamená, že v úvahu se bere pouze požadovaná maximální relativní chyba integrace.

Current

```
Entity *Current;
```

Proměnná Current je ukazatel na právě aktivní entitu, tj. objekt s diskrétním chováním, jehož popis chování (metoda Behavior) se právě provádí.

EndTime

```
double EndTime;
```

Proměnná EndTime obsahuje čas ukončení simulace.

MaxStep,MinStep

```
double MaxStep,MinStep;
```

Proměnné MaxStep a MinStep udávají povolený rozsah kroku numerické integrace. Nastavují se voláním funkce SetStep.

Phase

```
enum PhaseEnum {START,INITIALIZATION, SIMULATION, TERMINATION};

PhaseEnum Phase;
```

Proměnná Phase obsahuje označení právě probíhající fáze experimentu, může nabývat hodnot: START,INITIALIZATION, SIMULATION, TERMINATION

RelativeError

```
double RelativeError;
```

Proměnná RelativeError obsahuje požadovanou maximální relativní chybu numerické integrace. Integrační metoda bere tento požadavek v úvahu při volbě velikosti integračního kroku.

StartTime

```
double StartTime;
```

V proměnné StartTime je zaznamenán počáteční modelový čas, nastavený funkcí Init při inicializaci experimentu.

Time

```
double Time;
```

Proměnná Time reprezentuje modelový čas.

Objekt T

9.28 Standardní funkce

Abort

```
void Abort();
```

Funkce ukončí simulační program.
Activate

```
void Activate(Entity *e);
```

Funkce pro aktivaci entit. (viz třída Entity)
Init

```
void Init(double StartTime, double StopTime);
```

Funkce inicializuje kalendář, nastaví rozsah modelového času a proměnnou Time, inicializuje systém řízení simulace.
Passivate

```
void Passivate(Entity *e);
```

Funkce pro deaktivaci entit. (viz třída Entity)
Run

```
void Run();
```

Funkce zahájí simulaci od aktuální hodnoty času a pokračuje až do času, zadaného při předcházejícím volání Init. Simulaci lze předčasně ukončit stiskem klávesy ESC nebo programově funkcí Stop.
SetOutput

```
void SetOutput(char *name)
```

Nastaví standardní textový výstupní soubor pro všechny metody Output.
Stop

```
void Stop();
```

Funkce ukončí právě probíhající simulační běh. Je možno pokračovat v dalších experimentech.

9.29 Generátory náhodných čísel

Základem pro generování různých pseudonáhodných rozložení je generátor rovnoměrného rozložení na intervalu ¡0,1). Tento generátor je realizován funkcí Random, kterou lze předefinovat uživatelem definovanou funkcí. To umožňuje použít vlastní generátor pseudonáhodných čísel s lepšími vlastnostmi.
Exponential
double Exponential(double mv);
Funkce Exponential generuje pseudonáhodná čísla s exponenciálním rozložením a střední hodnotou mv.

Normal

double Normal(double mi, double sigma);

Funkce Normal generuje pseudonáhodná čísla s normálním rozložením, střední hodnotou mi a směrodatnou odchylkou sigma.

Poisson

int Poisson(double lambda);

Funkce Poisson generuje pseudonáhodná čísla Poissonova rozložení se střední hodnotou lambda.

Random

double Random();

Standardní funkce Random generuje pseudonáhodná čísla s rovnoměrným rozložením v intervalu od nuly do jedné. Horní mez intervalu (číslo jedna) negeneruje.

RandomInit

void RandomInit();

Funkce RandomInit inicializuje generátor pseudonáhodných čísel. Po jejím zavolání funkce Random generuje opět stejnou posloupnost pseudonáhodných čísel.

Uniform

```
double Uniform(double l, double h);
```

Funkce Uniform generuje pseudonáhodná čísla s rovnoměrným rozložením v intervalu od l včetně do h. Horní mez není zahrnuta do intervalu.

Ostatní generátory

Uvedeme pouze přehled ostatních generátorů:

```
double Beta(double th, double fi, double min, double max);
double Erlang(double alfa, double beta);
double Gama(double alfa, double beta);
int    Geom(double q);
int    HyperGeom(double p, int n, int m);
double Logar(double mi, double delta);
int    NegBinM(double p,int m);
int    NegBin(double q, int k);
double Rayle(double delta);
double Triag(double mod, double min, double max);
double Weibul(double lambda, double alfa);
```

9.30 Poznámky k implementaci SIMLIB

Současná verze knihovny je 2.12.

9.30.1 MSDOS

Knihovnu lze použít s těmito překladači:

Borland C++ v3.1 Borland C++ v4.0 (vypnout exception handling) Borland C++ v5.0 (vypnout exception handling) DJGPP = GNU C++ pro MSDOS

Paměťové požadavky knihovny samotné se pohybují kolem 100KB.

Pro 16 bitové překladače Borland C++ knihovna používá paměťový model LARGE, v integrovaném prostředí musí být pro každý model vytvořen projekt, tj. seznam modulů a

knihoven, které se budou spojovat do jednoho programu. Běžné modely vystačí s knihovnou SIMLIB a jedním modulem obsahujícím popis modelu i experimentu.

Pro překlad modelu je napsána dávka SIMLIB.BAT.

9.30.2 Linux

V tomto prostředí je prováděn vývoj knihovny. Používá se překladač

GNU C++ v2.7.2

a vývojové prostředí xwpe.

Překlad modelu provede skript SIMLIB.

Literatura

[1] Klir, J. G. (ed.): Trends in General Systems Theory. New York, J. Wiley 1973.

[2] Kalman, R. E., Falb, P. L., Arbib, M. A.: Topics in Mathemetical System Theory. (v ruském překladu Moskva, Mir 1971).

[3] Klír, J., Valach, M.: Kybernetické modelování. Praha, SNTL 1965.

[4] Mesarovič, M. D.: Views on General Systems Theory. (v ruském překladu Moskva, Mir 1966).

[5] Gordon, G.: System Simulation. New Jersey, Prentice-Hall 1969.

[6] Neuschl, Š. a kol.: Modelovanie a simulácia. Bratislava, Praha, Alfa, SNTL 1988.

[7] Haška, J.: Hybridní systémy. [Skriptum.] Praha, SNTL 1982.

[8] Ralston, A.: Základy numerické matematiky. Praha, Academia 1973.

[9] Douša, J.: Modelování a simulace. Praha, ČVUT 1980.

[10] Benyon, P. R.: A Review of Numerical methods for digital simulation. Simulation 1968.

[11] Fowler, M. E., Warten, R. M.: A Numerical Integration Technique for Ordinary Differential Equations with Widely Separated Eigen Values. IBM Journal of Research and Development, vol. 11 (1967), str. 537.

[12] Dahlquist, G. G.: A Special Stability Problem for Linear Multistep Methods. BIT, 1963, 3, str. 27-43.

[13] Nevřiva, P.: Simulace řídicích systémů na číslicovém počítači. Praha, SNTL 1975.

[14] Hanzálková, M.: Metody numerické integrace z hlediska aplikace v simulačních jazycích. Výzk. zpráva stát. úkolu III-2-1/5, FE VUT 1979.

[15] Ventzel, E. S.: Těorija verojatnotěj. Moskva, Nauka 1964.

[16] Saaty, L.: Elements of Queuing Theory With Applicatons. New York, Toronto, London, McGraw-Hill 1961.

[17] Feller, E.: An Introduction to Probability Theory and Its Applications. New York, London, Sydney, John Wiley 1957.

[18] Martin, J.: Design of Real-time Computer Systems. New Jersey, Prentice Hall 1967.

[19] Knuth, D. E.: The Art of Computer Programming. vol. 2, Addison Wesley 1969.

[20] Rogeberg, T.: Simulation and Simulation Languages. Publication No. S 48, Oslo, Norwegian Computing Center 1973.

[21] Fishman, G. S.: Concepts and Methods in Discrete Event Digital Simulation. New York, J.Wiley 1973.

[22] Peterson, J. L.: Modelling of parallel systems. Ph.D. Thesis, Electrical Engineering Department, Stanford University, 1974.

[23] Hušek, R., Lauber, J.: Simulační modely. Praha, Bratislava, SNTL, Alfa 1987.

[24] Češka, M., Hruška, T., Motyčková, L.: Vyčíslitelnost a složitost. Brno, Nakladatelství VUT 1992.

[25] Jensen, K.: Coloured Petri Nets: A High Level Language for System Design and Analysis. In: G. Rozenberg (ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science, Springer-Verlag 1991.

[26] Huber, P., Jensen, K., Shapiro, R. M.: Hierarchies in Coloured Petri Nets. In: G. Rozenberg (ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science, Springer-Verlag 1991.

[27] Schalkoff, R. J.: Artificial Intelligence: An Engineering Approach. New York, McGraw-Hill 1990.

[28] Booch, G.: Object Oriented Design. Redwood City, CA, The Benjamin/Cummings Publishing Company 1991.

[29] Stroustrup, B.: The C++ Programming Language. Addison-Wesley 1986

[30] Borland International: Borland C++ 3.0 Programmer's Guide. Scotts Valley, CA 1991

[31] Hayes, J. P.: An Introduction to Switch-level Modeling. IEEE Design & Test, August 1987, s.18-25.

[32] Bryant, R. E.: A Survey of Switch-level Algorithms. IEEE Design & Test, August 1987, s.26-40.

[33] Jain, K. S.: Agrawal, V. D.: STAFAN: An Alternative to Fault Simulation. In: Proc. of 21st Design Automation Conference 1984, s.18-23.

[34] OrCAD/VST. User's Guide. 1989. Zendulka, J., Schwarz, J.: Automatizace projektování číslicových systémů. Skriptum VUT Brno, Nakladatelství VUT 1992.

[35] IEEE Standard VHDL. Language Reference Manual, IEEE Std.1076-1987. IEEE 1988.

[36] Calahan, D.A.: Computer aided Network Design. McGraw-Hill Book Company 1968.

[37] Benda, Z., Staudek, J.: Programování v jazyku Simula 67. Praha, SNTL 1978.

[38] Černý, P.: Programové prostředky pro simulaci spojitých systémů na profesionálních osobních počítačích. In: Proc. of 22. seminář Simulace systémů a vědeckotechnické výpočty, Kopřivnice 1988, s.26-33.

[39] Kindler, E.: Simulační programovací jazyky. Praha, SNTL 1980.

Příloha A

Model víceprocesorového počítačového systému

A.1 Vymezení modelu

Uvažujme víceprocesorový počítačový systém, který je schematicky znázorněn na *obr A.1.*

Systém se skládá z n procesorů (*cpus*), které sdílejí společnou paměť (*memory*), z m diskových jednotek (*disks*) a z k kanálů (*chnnls*), které přenášejí data mezi disky a pamětí.

Jednotlivé práce (*job*) přicházejí do systému v náhodných okamžicích; interval mezi dvěma příchody má exponenciální rozložení se střední hodnotou *ARRTM.* Zpracování každé práce vyžaduje určitou dobu činnosti procesorů, určitý prostor paměti a provedení určitého počtu vstup/výstupních operací.

Předpokládejme, že celková doba činnosti *tcpu* procesorů pro jednu práci je exponenciálně rozloženou náhodnou veličinou se střední hodnotou *PROCTM.* Paměťový prostor pro jednu práci je rovnoměrně rozložená celočíselná náhodná veličina *mem* z intervalu 20..60 reprezentující počet požadovaných paměťových jednotek. Dalším atributem každé práce je počet bloků (*recs*); každý z nich vyžaduje provedení I/O operací. Tento počet bloků je úměrný požadované době *tcpu*; na jednu sekundu činnosti procesoru připadá náhodně 2..10 bloků.

Jakmile je práci přidělený požadovaný paměťový prostor, obsazuje kterýkoli z volných procesorů až do okamžiku, kdy se má provést vstup/výstupní operace. V tomto okamžiku práce uvolňuje procesor, který pak může být použit pro jinou práci. Po provedení požadovaných vstup/výstupních operací obsadí práce opět některý z volných procesorů a pokračuje zpracováním bloku. Délka zpracování bloku je náhodná veličina s exponenciálním rozložením; střední hodnota *tio* je rovna průměrné době $tcpu/recs$, jež připadá na jeden blok.

Provedení vstup/výstupních operací spočívá v nalezení příslušného záznamu na disku a v přenosu mezi diskem a pamětí.Předpokládáme, že doby hledání na disku jsou rovnoměrně rozloženy v intervalu $0..T_{SEARCH}$, kde T_{SEARCH} je maximální doba hledání. Dále předpokládáme, že délka přenosu má konstantní hodnotu rovnu 1/10 doby otáčky disku (T_TR).

A.2 Síťový model systému

Pro specifikaci systému použijeme CPN (*obr. A.2*). Datové typy jsou definovány jako množiny hodnot, zbytek deklarací je podobný tradičním programovacím jazykům. Poznáme-

nejme, že zvolený jazyk popisu sítě nelze brát jako dogma; je možné zvolit i jiný druh popisu.

Časovaný přechod *generator* generuje prostřednictvím funkce *init* jednotlivé práce. Místo *gen_s* slouží jako paměť stavu generátoru prací. Práce jsou reprezentovány n-ticemi atributů $(t0, tcpu, mem, recs, tio)$, kde $t0$je čas vygenerování práce, $tcpu$je požadovaná doba zpracování procesorem, memje požadavek na paměť, $recs$je počet zpracovávaných bloků, tioje střední doba zpracování bloku procesorem. Práce se postupně objevují v místech $q1$, $q2$, $q3$, $q4$a znovu $q2$. Přechod $enter$obsadí část paměti podle požadavků práce (mem). Paměť o kapacitě 128 jednotek je modelována místem *memory*. V případě, že čas zpracování práce procesorem není vyčerpán ($tcpu > 0$), časovaný přechod $incpu$obsadí jeden z volných procesorů z místa *cpus*, zpožděním modeluje zpracování jednoho bloku, dekrementuje $tcpu$a vygeneruje náhodně jednu z diskových jednotek, kam se bude blok přenášet. n-tice reprezentující práci je zde doplněna identifikací disku. Přechod $search$obsadí požadovaný disk z místa $disks$a zpožděním modeluje hledání na disku. Přechod $transfer$obsadí jeden z volných kanálů, zpožděním modeluje přenos dat a posléze uvolní disk obsazený prací. Práce se pak dostává do místa $q2$, kde se podle $tcpu$ rozhoduje, zda bude dále zpracována procesorem ($incpu$), nebo zda ukončí svůj pobyt v systému. V druhém případě ($tcpu \leq 0$) přechod $leave$uvolní obsazenou část paměti (mem) a zaznamená dobu pobytu práce v systému do histogramu funkcí $table(time - t0)$. Globální proměnná *time*, přístupná pouze pro čtení, reprezentuje modelový čas.

A.3 Model systému založený na procesech

Nyní budeme specifikovat systém v C++/SIMLIB pomocí procesů soupeřících o zdroje.

```
// model PROCESOR.CPP

#include "simlib.h"

// doba simulace
#define TSIM 30  // min

// počty objektů
#define n 3
#define m 2
#define k 1

const double ARRTM = 9;      // s
const double PROCTM = 6;     // s
const double T_SEARCH = 80;  // ms
const double T_TR = 30;      // ms

// deklarace globálních objektů
Facility  cpus[n];
Queue     q2("q2");
Facility  disks[m];
Facility  chnnls[k];
Queue     q4("q4");
```

```
Store     memory("Paměť",128);
Histogram table("tabulka",0,2e3,25);

class JOB : public Process {  // třída požadavků
  double t0;
  double tcpu;
  double tio;
  double deltaT;
  int mem;
  int recs;
  int disk;
  void Behavior() {
    t0 = Time;
    Enter(memory,mem);
    while(tcpu>0)
    {
       int i,j;
       for(i=0; i<n-1; i++) if(!cpus[i].Busy()) break;
       Seize(cpus[i]);
       deltaT = Exponential(tio);
       tcpu -= deltaT;
       Wait(deltaT);  // in cpu
       for(i=0; i<n; i++)
         if(cpus[i].in==this) Release(cpus[i]);
       disk = Uniform(0,m);
       Seize(disks[disk]);
       Wait(Uniform(0,T_SEARCH));  // search
       for(j=0; j<k-1; j++) if(!chnnls[j].Busy()) break;
       Seize(chnnls[j]);
       Wait(T_TR/10+Uniform(0,T_TR));  // transfer
       for(j=0; j<k; j++)
         if(chnnls[j].in==this) Release(chnnls[j]);
       Release(disks[disk]);
    }
    Leave(memory,mem);
    table(Time-t0);
  }
public:
  JOB() {
    tcpu = Exponential(PROCTM*1e3);
    mem = Uniform(20,61);
    recs = (tcpu/1000*Uniform(2,10))+1;
    tio = tcpu/recs;
  }
};

class Generator : public Event {  // generátor požadavků
  void Behavior() {
    new JOB->Activate();
    Activate(Time+Exponential(ARRTM*1e3));
```

```
  }
};

int main()
{
  int i;
  for(i=0; i<n; i++)
    cpus[i].SetQueue(q2);
  for(i=0; i<k; i++)
    chnnls[i].SetQueue(q4);
  Init(0,TSIM*60*1e3);
  new Generator->Activate();
  Run();
  table.Output();
}
```

Za definicí parametrů modelovaného systému následuje deklarace globálních objektů modelu. Třída *JOB*popisuje zpracování úloh v systému procesem. Obsazení procesoru je realizováno cyklem, který vyhledá první volný procesor. Pokud není volný žádný procesor, provede se příkaz k obsazení posledního procesoru, který zařadí požadavek do vstupní fronty. Protože však procesory mají společnou vstupní frontu $q2$, bude ve skutečnosti požadavek vybrán k obsluze na prvním procesoru, který se uvolní. Proto nelze použít číslo obsazovaného procesoru pro jeho uvolnění. Pro uvolnění procesoru skutečně obsazeného požadavkem slouží opět cyklus, který vyhledá příslušný procesor podle jeho atributu *in*. Podobné strategie využíváme u kanálů, které mají opět společnou frontu $q4$. Třída *Generator* popisuje událost, která generuje požadavky tím, že se periodicky aktivuje.

V popisu experimentu je důležité nastavení společné fronty pro všechny procesory metodou *SetQueue*. Procesory mají společnou frontu $q2$, kanály mají společnou frontu $q4$. Všechny časy v modelu jsou vyjádřeny v milisekundách. Experiment trvá 60 minut modelového času. Po skončení experimentu se tiskne histogram doby zpracování úlohy v systému.