Štefan Korečko a Štefan Hudák

Formálne metódy pre diskrétne systémy:

Petriho siete a B-metóda

2015

Recenzenti:
Prof. Ing. Dušan Krokavec, PhD.
Doc. RNDr. Petr Šaloun, PhD.

Poďakovanie
Táto monografia vznikla s podporou grantového projektu KEGA č. 050TUKE-4/2012: „Aplikácia technológií virtuálnej reality ako inovačného prostriedku pri výučbe formálnych metód“.

Obsah

Kapitola 1

Úvod

V informatike pojmom formálne metódy označujeme na matematickom základe postavené techniky pre špecifikáciu, vývoj, analýzu a verifikáciu počítačových systémov. Formálna metóda (FM) pozostáva [7] z formálneho jazyka s jednoznačne definovanou syntaxou a sémantikou a súboru procedúr, ktoré umožňujú so špecifikáciami v danom jazyku pracovať. Sémantika jazyka je definovaná pomocou vhodného matematického aparátu, akým je napríklad formálna logika či teória množín. Tým sa jazyky používané vo formálnych metódach líšia od bežných programovacích či špecifikačných jazykov, ktorých sémantika je opísaná v prirodzenom jazyku, čo necháva priestor na nejednoznačnú interpretáciu ich významu. Jednoznačne definovaná sémantika pre programovacie jazyky síce existuje, ale má podobu strojového kódu, ktorý je pre používateľa nečitateľný. Procedúry sú to, čo robí formálnu metódu metódou; umožňujú nám čosi robiť so zápisom vytvoreným v jazyku danej metódy. V praxi niektoré formálne metódy bežne používame bez toho, aby sme si ich príslušnosť do tejto skupiny vôbec uvedomovali. Napríklad, z oblasti vývoja hardvéru tu patria konečné automaty s výstupom (Mealy a Moore): syntax a sémantika zápisu automatu je daná formálnou definíciou jeho štruktúry a prechodovej a výstupnej funkcie. Procedúrami potom sú redukcia automatu či syntéza sekvenčného logického obvodu z neho. Ďalším príkladom sú regulárne výrazy [7]: syntax ich jazyka je možné definovať bezkontextovou gramatikou, sémantiku pomocou teórie množín a procedúrou môže byť vyhľadanie všetkých slov zodpovedajúcich danému výrazu v texte.

V súčasnej dobe existuje množstvo formálnych metód určených pre rôzne typy systémov (sekvenčné – súbežné, deterministické – stochastické) a podľa dostupných procedúr poskytujúcich rôznu úroveň funkcionality. Na základe toho J.P. Bowen [6, 4] delí formálne metódy, resp. ich použitie, na 3 úrovne:

Formálna špecifikácia (úroveň 0). K dispozícii je formálna notácia (jazyk), ktorá umožňuje opísať systém. Procedúry väčšinou nejdú za hranicu kontroly syntaxe či jednoduchej simulácie (animácie). Použitie takejto metódy je „najlacnejšie“, keďže je potrebné iba si osvojiť daný jazyk a softvérový nástroj pre ňu je v podstate na úrovni sofistikovanejšieho textového editora. Napriek obmedzenej funkcionalite vie takáto metóda prispieť ku kvalite vyvíjaného systému, keďže umožňuje vytvoriť jeho jednoznačný opis už v ranných fázach vývoja, kde je zvyčajne používaný prirodzený jazyk. Takto je priestor včas odstrániť nejednoznačnosti, ktoré by inak mohli viesť k závažným chybám v implementácii.

Formálna verifikácia alebo vývoj (úroveň 1). Metóda poskytuje techniky na overenie, či popísaný systém má želané vlastnosti, prípadne aj techniky pre prechod od abstraktnej špecifikácie ku konkrétnej, implementovateľnej špecifikácii. Tieto techniky nemusia nutne zahŕňať formálny dôkaz, no musia byť rigorózne.

Počítačom vykonaný dôkaz (úroveň 2). Metódy pre ktoré existujú techniky ako na úrovni 1, no navyše sú k dispozícii softvérové nástroje, ktoré realizujú formálne dôkazy. Tieto nástroje sú buď tzv. *overovače modelov* (model checkers) alebo *dokazovače teorém* (theorem provers). Overovače modelov sú plne automatické nástroje založené na prehľadávaní stavového priestoru, kde platnosť príslušnej vlastnosti preverujú pre každý stav, resp. triedu stavov. Ich výhodou je už spomínaná automatickosť, nevýhodou značné pamäťové a časové nároky a to aj v prípade relatívne jednoduchých systémov. Na druhej strane, dokazovače teorém vykonávajú dôkazy postupným aplikovaním sady pravidiel, tak ako sa to vo formálnych systémoch štandardne robí „na papieri“. Výhodou je, že nie je nutné budovať stavový priestor. Nevýhodou je fakt, že ak sa dokazovaču nepodarí platnosť teorémy dokázať neznamená to, že neplatí. Taktiež je často potrebný zásah používateľa, a teda dôkaz nie je úplne automatický.

V praxi je, samozrejme, možné použiť metódu vyššej úrovne na nižšej úrovni. Ďalej sa FM delia na *ľahké* (lightweight) a *ťažké* (heavyweight), a to podľa náročnosti ich použitia. Pri ľahkých sa očakáva, že pre vykonanie nejakej činnosti, napr. verifikácie, postačí stlačiť príslušné tlačidlo v nástroji danej metódy, pri ťažkej je nutná spolupráca používateľa. Preto overovače modelov zahrňujeme medzi ľahké a dokazovače teorém medzi ťažké FM.

V súvislosti s formálnymi metódami existuje viacero mýtov, napríklad, že ich nasadenie vo vývoji určite zaručí korektnosť príslušného systému či predraží vývoj [16]. Prvé z týchto tvrdení je nepravdivé, pretože aj keď daná FM poskytuje overenie platnosti nejakej vlastnosti v každom stave systému, nevie zaručiť, že táto vlastnosť naozaj správne vyjadruje požiadavky na korektnosť systému. Druhé tvrdenie vychádza zo skutočnosti, že nasadenie formálnych metód znamená pre vývojárov nutnosť osvojiť si nový jazyk a prácu s ním, a tiež, že je vhodné k vývoju prizvať experta na FM. Zanedbáva však fakt, že odstránenie chýb a nejednoznačností na úrovni špecifikácie (tzn. v ranných fázach vývoja) je vo všeobecnosti oveľa lacnejšie ako počas jeho implementácie. Formulácii a vyvráteniu týchto a podobných mýtov, a celkovo odporúčaniam pre použitie formálnych metód, sa venuje viacero prác, napríklad [4, 5, 6, 16]. Aktuálny prehľad praktického nasadenia formálnych metód je možné nájsť v [56], resp. jeho aktualizovanej verzii [13].

V tejto monografii sa zameriame na dve formálne metódy, a to Petriho siete a B-metódu. Náš výber má dva dôvody. Po prvé, tieto metódy reprezentujú odlišné rodiny FM: Petriho siete sa zameriavajú na opis správania systému, vrátane súbežného. Majú ľahko pochopiteľnú grafickú formu, podobnú konečným automatom, a sú použiteľné v rôznych oblastiach. B-metóda je orientovaná najmä na vývoj softvéru a sekvenčné systémy. Má dobre definovaný verifikačný a vývojový proces, ktorý umožňuje formálne dokázať, že systém pomocou nej špecifikovaný má požadované vlastnosti, previesť (zjemniť) tento systém do implementovateľnej podoby a dokázať, že táto tiež zachováva požadované vlastnosti. Druhým dôvodom je existencia pôvodných výsledkov výskumu, ktoré pre tieto metódy autori dosiahli. Z nich sú v tejto práci prezentované vybrané výsledky v riešení problému dosiahnuteľnosti v Petriho sieťach a vo vývoji podporných prostriedkov pre výučbu B-metódy a príbuzných formálnych metód.

Ďalšie časti publikácie sú organizované nasledovne: Kapitola 2 je ve-

novaná Petriho sieťam (PS), konkrétne základnému typu PS nazývanému P/T siete, farbeným PS a hodnotiacim PS. Jej časti 2.1.6, 2.1.7 predstavujú pôvodné výsledky v riešení problému dosiahnuteľnosti a dekompozičnom prístupe k nemu. Kapitola 3 podáva základný opis B-metódy a jej vývojového procesu. Na ňu nadväzuje kapitola 4, približujúca pod vedením autorov vyvinutý podporný softvér pre výučbu formálnych metód, ktorý využíva simulátor riadenia vlakovej dopravy ako virtuálnu reprezentáciu prostredia typického pre ich nasadenie. Kapitola prináša aj dva príklady riadiacich systémov v jazyku B-metódy, ktoré boli vyvinuté s použitím tohto softvéru. Záver práce podáva stručný prehľad ďalších výsledkov dosiahnutých autormi v súvislosti s Petriho sieťami a B-metódou a načrtáva výhľad do budúcnosti. Pre lepšiu orientáciu je publikácia doplnená o zoznam skratiek a typografické konvencie.

Kapitola 2

Petriho siete

Petriho siete (PS) patria medzi formálne metódy orientované na opis správania sa systémov a autorom ich pôvodnej verzie je Carl Adam Petri. Tú predstavil v svojej dizertačnej práci Kommunikation mit Automaten [50]. Táto verzia bola neskôr mnohými výskumníkmi rozšírená a obohatená o rôzne techniky analýzy a syntézy. V dnešnej dobe je k dispozícii množstvo typov Petriho sietí s rôznou vyjadrovacou[1] a modelovacou[2] silou. V tejto kapitole sa zameriame na tri typy PS, ku ktorým sa viažu pôvodné výsledky autorov. Ide o P/T siete, predstavujúce základný typ PS, farbené PS (CPN), veľmi rozšírený typ patriaci medzi tzv. vysokoúrovňové PS a hodnotiace PS (EvPN), ktoré sú rozšírením P/T sietí s vyjadrovacou silou Turingovho stroja. Koncepčne Petriho siete vychádzajú z konečných automatov, majú podobnú grafickú reprezentáciu, no navyše umožňujú prirodzeným spôsobom popísať súbežné a paralelné systémy, a to aj s nekonečným počtom stavov. Cenný je aj analytický aparát niektorých typov Petriho sietí, pomocou ktorého vieme napríklad získať invariantné (nemenné) vlastnosti modelovaného systému. Prehľad existujúcich typov PS, dostupných nástrojov, ako aj prípadov ich praktického nasadenia je možné nájsť na stránke Petri Nets World [64].

[1] Pod pojmom vyjadrovacia sila rozumieme schopnosť formálneho jazyka opísať určitý typ systému; bežne sa táto sila porovnáva voči Turingovmu stroju.

[2] Modelovacia sila hovorí o tom, do akej miery je v danom jazyku možné jednoducho vytvoriť prehľadný, zrozumiteľný a kompaktný opis systému. Napríklad programovací jazyk Java a jazyk symbolických inštrukcií (asembler) majú rovnakú vyjadrovaciu silu, ale Java má modelovaciu silu vyššiu ako asembler.

2.1 P/T siete

P/T siete (*Place/Transition nets*) predstavujú základný typ Petriho sietí, spájajúci ľahko pochopiteľnú grafickú notáciu s technikami analýzy, umožňujúcimi odvodiť invariantné vlastnosti zo štruktúry siete. P/T sieť má podobu orientovaného bipartitného (párneho) grafu, kde jedným typom vrcholov sú tzv. *miesta* (*places*) a druhým tzv. *prechody* (*transitions*). Miesta majú podobu kruhov, alebo elíps a vyjadrujú distribuovaný stav siete. Konkrétna hodnota stavu siete je daná počtom značiek (tzv. tokenov) v miestach. Prechody sú v tvare obdĺžnikov, štvorcov, alebo hrubých čiar a reprezentujú udalosti, ktoré v sieti môžu nastať. Prechody je možné vykonať, čím dôjde k zmene stavu siete, tzn. počtu tokenov v jej miestach. Stav siete sa nazýva *značenie* (*marking*). Prechody sa tiež zvyknú nazývať udalosťami a miesta podmienkami. Kedy je prechody možné vykonať, a ako ich vykonanie ovplyvní stav siete, je určené hranami a ním priradenými váhami. P/T siete sú tiež známe pod názvom *zovšeobecnené Petriho siete* (*GPN*, *Generalized Petri Nets*).

Väčšinu pojmov a vlastností, ktoré tu uvádzame pre P/T siete, nájdeme aj u iných typov PS. Aj tie sú zložené z miest, prechodov a orientovaných hrán a ich správanie je o realizácii prechodov a zmene značenia. Líšia sa však mechanizmom realizácie či povahou tokenov.

2.1.1 Použitá notácia

Predtým, ako presne definujeme P/T siete a ich správanie, zavedieme potrebnú notáciu, ktorú tu budeme používať.

Množinu prirodzených čísel $\{0, 1, 2, \ldots\}$ označíme ako $\mathbb{N}$, *množinu celých čísel* ako $\mathbb{Z}$. Symbol $\mathbb{Z}^k$ bude predstavovať množinu *k-rozmerných celočíselných vektorov*. Podmnožinu $\mathbb{Z}^k$ obsahujúcu nezáporné vektory pomenujeme $\mathbb{N}^k$. Pre vektory zo $\mathbb{Z}^k$ budeme používať relácie $=$, $\leq$ a $<$, definované ako

$$q \,\mathfrak{op}\, r \Leftrightarrow \forall i (1 \leq i \leq k) : q_i \,\mathfrak{op}\, r_i, \;\; \mathfrak{op} \in \{<, =, \leq\}$$

kde $q = (q_1, \ldots, q_k)$ a $r = (r_1, \ldots, r_k)$. Symbol ω znamená hodnotu väčšiu ako ktorékoľvek prirodzené číslo. Z toho vyplýva, že

$$\forall a (a \in \mathbb{Z}) : \omega + a = \omega - a = \omega.$$

Ak $p \leq q$ hovoríme, že q *pokrýva* p, ak $p = q$, že q *je totožný s* p. *Kardinalita* množiny A je označovaná ako $|A|$.

Ak má nejaký objekt x „obyčajnú“ aj vektorovú formu, označíme tú obyčajnú ako x a vektorovú ako $\vec{x}$. Písmeno „T“ v hornom indexe vždy znamená transpozíciu vektora alebo matice. Pre matice použijeme štandardný zápis, tzn. matica A s m riadkami a n stĺpcami bude definovaná ako $A = (A_{i,j})_{m \times n}$ a $A_{i,j}$ je jej prvok na i-tom riadku a j-tom stĺpci. $0_{m,n}$ je *nulová matica* rozmeru $m \times n$ a 0^k je k-rozmerný *nulový vektor*. Ak nebude uvedené inak, považujeme vektory za riadkové.

Ak σ je sekvencia (reťazec) symbolov z V (zapisujeme $\sigma \in V^*$) a $a \in V$, potom pod $\sigma(a)$ rozumieme počet výskytov a v σ.

2.1.2 Štruktúra a správanie P/T siete

P/T sieť je možné formálne definovať ako usporiadanú štvoricu [19].

Definícia 2.1.1 (P/T sieť)**.** *P/T sieť N je usporiadaná štvorica*

$$N = (P, T, pre, post)$$

kde *P je konečná množina miest,*
T je konečná množina prechodov,
$pre : P \times T \to \mathbb{N}$ je pre-podmienka a
$post : P \times T \to \mathbb{N}$ je post-podmienka.

□

Množiny P a T sú množiny vrcholov grafu P/T siete a funkcie *pre* a *post* definujú hrany medzi nimi:

- Ak pre nejaké $p \in P$ a $t \in T$ je $pre(p, t) > 0$, existuje v grafe siete orientovaná hrana z miesta p do prechodu t s váhou rovnou hodnote $pre(p, t)$.
- Ak pre nejaké $p \in P$ a $t \in T$ je $post(p, t) > 0$, existuje v grafe siete orientovaná hrana z prechodu t do miesta p s váhou $post(p, t)$.

Hodnoty váh väčšie ako 1 zapisujeme na resp. vedľa príslušnej hrany.

Značenie P/T siete $N = (P, T, pre, post)$ je definované ako funkcia z množiny miest P do množiny prirodzených čísel:

$$m : P \to \mathbb{N}$$

Hodnota $m(p)$ vyjadruje počet značiek (tokenov) v mieste p. Značenie celej siete je zložené zo značení jej miest a za predpokladu, že $P = \{p_1, \ldots, p_k\}$, možno ho zapísať vo vektorovej podobe ako

$$\vec{m} = (m(p_1), m(p_2), \ldots, m(p_k))$$

Na označenie rôznych značení P/T siete používame písmeno m s rôznymi dolnými, resp. hornými indexmi. Napríklad pre *počiatočné značenie*, tzn. značenie v ktorom sa sieť nachádza po inicializácii, štandardne používame označenie m_0. P/T sieť, ktorá obsahuje m_0 sa nazýva *inicializovaná* a zapisujeme ju ako $N_0 = (P, T, pre, post, m_0)$.

Vzťah medzi definíciou a grafom P/T siete ilustruje nasledujúci príklad.

Príklad 2.1.1 (Definícia a graf P/T siete MutEx)**.** P/T sieť $N_0 = (P, T, pre, post, m_0)$ s množinami miest a prechodov

$$P = \{p1, p2, p3, p4, p5\},$$
$$T = \{t1, t2, t3, t4\},$$

počiatočným značením

$$m_0 = (1, 0, 1, 0, 1), \text{ tzn. } m_0(p1) = 1, m_0(p2) = 0 \text{ až } m_0(p5) = 1$$

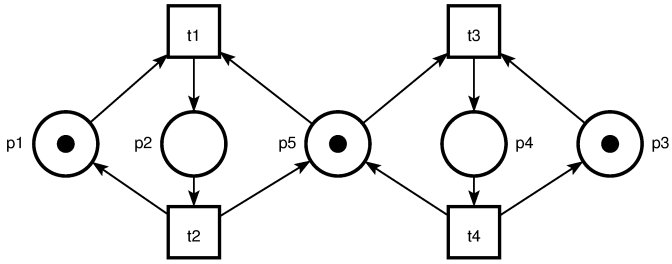
a nasledovne definovanými funkciami *pre* a *post*

p	**t**	**pre(p,t)**
p1	t1	1
p5	t1	1
p2	t2	1
p5	t3	1
p3	t3	1
p4	t4	1
inak		0

p	**t**	**post(p,t)**
p1	t2	1
p2	t1	1
p5	t2	1
p3	t4	1
p4	t3	1
p5	t4	1
inak		0

bude mať grafickú podobu ako na obrázku 2.1. Tento obrázok, podobne ako ostatné siete v časti 2.1, bol vytvorený pomocou nástroja TIme petri Net Analyzer [63].

□



Obr. 2.1: Graf P/T siete

Definícia P/T siete, ktorú sme uviedli, nie je jediná. Napríklad v [11] sú hrany definované reláciou F, $F \subseteq (P \cup T) \times (P \cup T)$.

Správanie, resp. *výpočet* P/T siete je definovaný ako postupné *vykonávanie* (odpaľovanie, realizácia) *vykonateľných* (realizovateľných, prípustných) prechodov. Vykonateľnosť a vykonanie prechodov sú definované nasledovne:

Definícia 2.1.2 (vykonateľnosť a vykonanie prechodu)**.** *Prechod $t \in T$ v PS $N = (P, T, pre, post)$ je* vykonateľný *v značení $m \in \mathbb{N}^{|P|}$ (zapisujeme $m \overset{t}{\vdash}$), ak je splnená podmienka:*

$$\forall p(p \in P) : m(p) \geq pre(p, t)$$

Výsledkom vykonania prechodu *je nové značenie $m' \in \mathbb{N}^{|P|}$ (zapisujeme $m \overset{t}{\vdash} m'$):*

$$m'(p) = m(p) - pre(p, t) + post(p, t)$$

□

Vykonanie prechodu $t \in T$ je teda možné len ak je v jeho *premiestach*, tzn. v miestach z ktorých vedie do t hrana, dostatok značiek. Samotné vykonanie t spôsobí odstránenie značiek z jeho pre-miest a pridanie značiek do jeho *post-miest*, tzn. do miestach do ktorých vedie z t hrana. Ak značenie PS umožňuje realizáciu viacerých prechodov naraz, vykoná sa iba jeden - ľubovoľný. To je dané vlastnosťou

nedeterminizmu PS. Prechody, ktoré je v danom značení možné naraz odpáliť nazývame *súbežne vykonateľné*. Či ich súbežné vykonanie bude naozaj možné, závisí od sémantiky daného typu PS. Ako vidno z definície 2.1.2, P/T siete takéto vykonanie nepovoľujú. Na základe tejto definície môžme definovať aj dlhšie výpočty v podobe sekvencií odpálení.

Definícia 2.1.3 (prípustná sekvencia prechodov, dosiahnuteľné značenie)**.** *Sekvencia* $\sigma \in T^*$, $\sigma = t_1, t_2, \ldots, t_{r-1}$ *je* prípustná sekvencia prechodov P/T siete N v značení m_1 *(zapisujeme* $m_1 \overset{\sigma}{\vdash}$*), ak existujú značenia* $m_1, m_2, \ldots, m_r$ *také, že* $m_i \overset{t_i}{\vdash} m_{i+1}, i = 1, 2, \ldots, r-1$.
Značenie m_r *sa nazýva* dosiahnuteľné značenie z m_1 cez σ *(zapisujeme* $m_1 \overset{\sigma}{\vdash} m_r$*).*

□

Pre P/T sieť $N_0 = (P, T, pre, post, m_0)$ je definovaná *množina dosiahnuteľných značení*:

$$\mathcal{R}(N_0) = \{m | m_0 \overset{*}{\vdash} m\}$$

Symbol „$*$“ predstavuje ľubovoľnú sekvenciu prechodov. *Jazyk* siete N_0 je množina

$$\mathcal{L}(N_0) = \{\sigma \in T^* | m_0 \overset{\sigma}{\vdash}\}$$

Vzťah medzi jazykom PS a formálnymi jazykmi Chomského hierarchie je popísaný v [19]. Výpočet P/T siete si ukážeme na sieti z príkladu 2.1.1.

Príklad 2.1.2 (Výpočet P/T siete MutEx)**.** V počiatočnom značení $m_0 = (1, 0, 1, 0, 1)$ P/T siete z príkladu 2.1.1 sú vykonateľné prechody $t1$ a $t3$, no nie súbežne; tomu bráni skutočnosť, že ich spoločné premiesto $p5$ obsahuje iba jednu značku. Vykonaním $t1$ v m_0 prejdeme do nového značenia m_1:

$$m_0 \overset{t1}{\vdash} m_1,\ m_1 = (0, 1, 1, 0, 0),$$

vykonaním $t3$ by sme prešli do m_2:

$$m_0 \overset{t3}{\vdash} m_2,\ m_2 = (1, 0, 0, 1, 0).$$

V m_1 je vykonateľný iba $t2$ a privedie nás späť do m_0 ($m_1 \overset{t2}{\vdash} m_0$), podobne $t4$ v m_2. Žiadne iné značenie sa už v sieti dosiahnuť nedá a teda

$$\mathcal{R}(N_0) = \{m_0, m_1, m_2\}.$$

Jazyk tejto siete reprezentuje nasledovný regulárny výraz:

$$\mathcal{L}(N_0) = (t1.t2 + t3.t4)^*.(t1 + t3 + \varepsilon)$$

kde „+“ je alternatíva, „.“ zreťazenie, „*“ 0 alebo viac opakovaní a „ε“ prázdny reťazec.

Táto P/T sieť sa nazýva *MutEx* a vyjadruje vzájomné vylúčenie dvoch procesov. Význam jej miest a prechodov je nasledovný: ľavá časť siete zodpovedá prvému a pravá druhému procesu. Ak je značka v $p1$ znamená to, že prvý proces pracuje mimo kritickú oblasť, ak v $p2$ že pracuje v kritickej oblasti. Analogický význam majú $p3$ a $p4$ pre druhý proces. Kritická oblasť môže byť pamäť alebo iný zdroj ku ktorému je povolené naraz pristúpiť iba jednému procesu. Miesto $p5$ slúži ako semafor. Ak v ňom značka je, možno do kritickej oblasti vstúpiť ak nie je, nemožno. Odpálenie $t1$ je vstupom prvého procesu do kritickej oblasti, odpálenie $t2$ jej opustenie. Analogicky sú $t3$ a $t4$ vstupom a opustením pre druhý proces. Sieť MutEx nevyužíva naplno potenciál P/T sietí, keďže značenie akéhokoľvek miesta je maximálne 1 a váha hrán je vždy 1.

□

2.1.3 Algebraické vyjadrenie P/T siete

Štruktúra P/T siete je vyjadriteľná pomocou matíc a vektorov (definície 2.1.4, 2.1.5) a jej správanie maticovými a vektorovými operáciami (definícia 2.1.6). Toto algebraické vyjadrenie s výhodou použijeme pri formulovaní analytických vlastností (časť 2.1.4) a riešení problému dosiahnuteľnosti (časti 2.1.6, 2.1.7) P/T sietí. V tu uvedených definíciách, a aj nasledujúcich častiach, budeme predpokladať, že P/T sieť N_0 je

$$\begin{aligned} N_0 &= (P, T, pre, post, m_0) \\ P &= \{p_1, \ldots, p_k\} \\ T &= \{t_1, \ldots, t_n\}. \end{aligned} \tag{2.1}$$

Definícia 2.1.4 (maticové vyjadrenie P/T siete)**.** *P/T sieť N_0 (2.1) je reprezentovaná maticami*

- Pre-matica Pre, $Pre = (Pre_{i,j})_{k\times n}$, $Pre_{i,j} = pre(p_i, t_j)$,
- Post-matica $Post$, $Post = (Post_{i,j})_{k\times n}$, $Post_{i,j} = post(p_i, t_j)$ *a*
- Incidenčná matica C, $C = (C_{i,j})_{k\times n}$, $C = Post - Pre$.

Definícia 2.1.5 (vektorové vyjadrenie prechodu)**.** *Nech $t_j \in T$ je prechod P/T siete N_0* (2.1)*. Potom jeho* vektorovú forma *je*

$$\vec{t_j} = (C_{1,j}, \ldots, C_{k,j})$$

Pre každý prechod $t_j \in T$ navyše môžme definovať vektory

$$\begin{aligned} t_j^{\vec{pre}} &= (Pre_{1,j}, \ldots, Pre_{k,j}), \\ t_j^{\vec{post}} &= (Post_{1,j}, \ldots, Post_{k,j}), \end{aligned}$$

ktoré tiež definujú $\vec{t_j}$:

$$\vec{t_j} = t_j^{\vec{post}} - t_j^{\vec{pre}}.$$

□

Ak N_0 spĺňa podmienku, že neobsahuje tzv. *vlastné slučky*, formálne

$$\neg(\exists(t \in T \wedge p \in P) : pre(p,t) > 0 \wedge post(p,t) > 0), \tag{2.2}$$

tak je jej reprezentácia incidenčnou maticou jednoznačná a takúto sieť nazývame *čistá* (*pure*). Inak jednoznačná nie je, pretože ak napríklad $C_{i,j} = 0$, tak je síce zrejmé, že $pre(p_i, t_j) = post(p_i, t_j) = r$, ale z incidenčnej matice nevieme určiť či $r = 0$ (žiadna hrana medzi p_i a t_j) alebo $r > 0$ (hrany s váhou r oboma smermi medzi p_i a t_j). Realizovateľnosť a odpálenie prechodu budú v termínoch definícii 2.1.4, 2.1.5 určené nasledovne:

Definícia 2.1.6 (vykonateľnosť a vykonanie prechodu)**.** *Prechod t_j P/T siete N_0* (2.1) *je vykonateľný v značení m práve vtedy ak*

$$\vec{m} - t_j^{\vec{pre}} \geq 0^{\vec{k}}. \tag{2.3}$$

Vykonaním prechodu t_j v m vznikne nové značenie m':

$$\vec{m'} = \vec{m} + \vec{t_j}. \tag{2.4}$$

□

Pre čistú P/T sieť namiesto podmienky (2.3) postačuje (2.5), preto je pre ňu incidenčná matica jednoznačnou reprezentáciou.

$$\vec{m} + \vec{t_j} \geq 0^{\vec{k}}. \tag{2.5}$$

Pomocou algebraického vyjadrenia sa dá efektívne vypočítať aj značenie dosiahnuté odpálením sekvencie prechodov (veta 2.1.1). Tento výpočet využíva incidenčnú maticu C a vektor zvaný *Parikhov obraz* (definícia 2.1.7), ktorý určuje početnosť prechodov v sekvencii.

Definícia 2.1.7 (Parikhov obraz)**.** *Majme P/T sieť* N_0 (2.1) *a prípustnú sekvenciu prechodov* $\sigma \in T^*$. Parikhov obraz σ *vzhľadom na* $T = \{t_1, \ldots, t_n\}$ *je vektor*

$$\Psi(\sigma) = (\sigma(t_1), \ldots, \sigma(t_n)),$$

kde $\sigma(t_i)$ *je početnosť výskytu prechodu* t_i *v* σ.

Definícia 2.1.8 (vektorové vyjadrenie sekvencie prechodov)**.** *Majme P/T sieť* N_0 (2.1). *Potom* vektorové vyjadrenie sekvencie prechodov $\sigma \in T^*$ *je stĺpcový vektor* $[\sigma]$, $[\sigma] = C.(\Psi(\sigma))^T$.

Veta 2.1.1. *Majme P/T sieť* N_0 (2.1), *jej incidenčnú maticu* C *a reťazec* $\sigma \in T^*$. *Ak* $m_0 \overset{\sigma}{\vdash} m$ *tak* $m = m_0 + [\sigma]^T$.

Algebraické vyjadrenie ilustruje príklad 2.1.3.

Príklad 2.1.3 (Algebraické vyjadrenie siete MutEx)**.** Incidenčná matica P/T siete MutEx z príkladov 2.1.1 a 2.1.2 má tvar (2.6).

	$t1$	$t2$	$t3$	$t4$
$p1$	-1	1	0	0
$p2$	1	-1	0	0
$p3$	0	0	-1	1
$p4$	0	0	1	-1
$p5$	-1	1	-1	1

(2.6)

Napríklad pre prechod $t1$ majú vektory podľa definície 2.1.5 tvar

$$\begin{aligned} t\vec{1} &= (-1, 1, 0, 0, -1), \\ t1^{\vec{pre}} &= (1, 0, 0, 0, 1), \\ t1^{\vec{post}} &= (0, 1, 0, 0, 0) \end{aligned}$$

a odpálenie $m_0 \overset{t1}{\vdash} m_1$ je vypočítané ako

$$(0,1,1,0,0) = (1,0,1,0,1) + (-1,1,0,0,-1).$$

Parikhov obraz sekvencie $\sigma = t1\,t2\,t1\,t2\,t3\,t4$ je $\Psi(\sigma) = (2,2,1,1)$.

□

2.1.4 Analytické vlastnosti P/T sietí

Ako sme už spomenuli, P/T siete disponujú jedinečnými analytickými vlastnosťami, umožňujúcimi automatické odvodenie *invariantných vlastností* siete. Invariantné vlastnosti systému sú také, ktoré platia vždy, v každom jeho stave. Pre P/T sieť vieme odvodiť dva typy takýchto vlastností, a to:

S-invarianty (invarianty miest), ktoré definujú nemenný pomer medzi značeniami miest. Ak je známy význam miest, dajú sa prepísať na tvrdenia o danom systéme, ktoré platia v každom jeho stave.

T-invarianty (invarianty prechodov), charakterizujúce sekvencie prechodov, ktorých vykonanie vedie z daného značenia späť do toho istého značenia.

Oba invarianty majú podobu stĺpcových vektorov, S-invarianty majú rozmer množiny P a T-invarianty množiny T. Definície 2.1.9 a 2.1.10 definujú samotné invarianty a vety 2.1.2 a 2.1.3 formalizujú ich význam. Dôkaz oboch viet je jednoducho vykonateľný s využitím definícií a vety z časti 2.1.3. V oboch definíciách a vetách je N_0 P/T sieť v zmysle (2.1) (strana 11), C jej incidenčná matica a operátor „$\cdot$“ je násobenie vektorov a matíc.

Definícia 2.1.9. *Majme N_0 a k-zložkový stĺpcový vektor X, $k = |P|$. Ak platí, že $C^T \cdot X = (0^k)^T$ tak X je* S-invariant *siete N_0.*

Veta 2.1.2. *Majme stĺpcový vektor X, ktorý je S-invariant siete N_0. Potom pre $\forall m\,(m \in \mathcal{R}(N_0)) :\ \vec{m} \cdot X = \vec{m_0} \cdot X$.*

Definícia 2.1.10. *Majme N_0 a n-zložkový stĺpcový vektor Y, $n = |T|$. Ak platí, že $C \cdot Y = (0^n)^T$ tak Y je* T-invariant *siete N_0.*

Veta 2.1.3. *Majme stĺpcový vektor Y, ktorý je T-invariant siete N_0. Ďalej majme reťazec $\sigma \in T^*$ taký, že $m_0 \overset{\sigma}{\vdash} m$ a Parikhov obraz σ je Y ($\Psi(\sigma) = Y$). Potom $m_0 = m$.*

Rovnice $\vec{m} \cdot X = \vec{m_0} \cdot X$, ktoré získame aplikáciou vety 2.1.2, sa zvyknú označovať ako *axiómy siete*. Konkrétny výpočet a význam invariantov znova ukážeme na sieti MutEx.

Príklad 2.1.4 (Invarianty siete MutEx)**.** Vektor X pre rovnicu podľa definície 2.1.9 má pre sieť MutEx (príklady 2.1.1–2.1.3) podobu $X = (x_1, x_2, x_3, x_4, x_5)^T$ a násobíme ho maticou C^T, získanou transpozíciou incidenčnej matice (2.6). Dostaneme tak sústavu rovníc (2.7) s nekonečným počtom riešení, keďže 1. a 2. rovnica sú lineárne závislé, podobne 3. a 4. rovnica.

$$\begin{array}{rcl} -x_1 + x_2 - x_5 & = & 0 \\ x_1 - x_2 + x_5 & = & 0 \\ -x_3 + x_4 - x_5 & = & 0 \\ x_3 - x_4 + x_5 & = & 0 \end{array} \tag{2.7}$$

Ak ako závislé premenné zvolíme x_2 a x_4, dostávame (2.8).

$$\begin{array}{rcl} x_2 & = & x_1 + x_5 \\ x_4 & = & x_3 + x_5 \end{array} \tag{2.8}$$

Keďže lineárna kombinácia invariantov je tiež invariant, je postačujúce uviesť „základné“ invarianty, kde kombinácie hodnôt dosadené za nezávislé premenné zodpovedajú jednotkovým vektorom v danom priestore (tu v 3-rozmernom), aj keď zaujímavé vlastnosti môže ukrývať aj niektorý odvodený invariant. Po dosadení jednotkových vektorov do (2.8) základné S-invarianty budú tri:

$$\begin{array}{rcl} X_1 & = & (1,1,0,0,0)^T \\ X_2 & = & (0,0,1,1,0)^T \\ X_3 & = & (0,1,0,1,1)^T \end{array} \tag{2.9}$$

Z (2.9) použitím vety 2.1.2 získame tri axiómy:

$$m(p_1) + m(p_2) = 1 \tag{2.10}$$

$$m(p_3) + m(p_4) = 1 \tag{2.11}$$

$$m(p_2) + m(p_4) + m(p_5) = 1 \tag{2.12}$$

Význam axióm, vzhľadom na systém ktorý sieť opisuje, je nasledujúci:

- Prvý proces pracuje buď mimo kritickú oblasť alebo v nej (podľa (2.10)). To isté platí pre druhý proces (podľa (2.11)).

- Oba procesy nemôžu pracovať naraz v kritickej oblasti. Vstup do kritickej oblasti je povolený iba ak v nej nepracuje žiaden proces (podľa (2.12)).

Axióma (2.12) je dôkazom, že systém má požadovanú vlastnosť, že nedovolí vstúpiť viacerým procesom naraz do kritickej oblasti.

Pri výpočte T-invariantov použijeme vektor $Y = (y_1, y_2, y_3, y_4)^T$ a maticu C (2.6). Sústava podľa definície 2.1.10 bude mať 4 rovnice, ktoré sa znova redukujú na 2 nezávislé (2.13).

$$\begin{array}{rcl} y_1 & = & y_2 \\ y_3 & = & y_4 \end{array} \tag{2.13}$$

Základné T-invarianty budú dva (2.14)

$$\begin{array}{rcl} Y_1 & = & (1,1,0,0)^T \\ Y_2 & = & (0,0,1,1)^T \end{array} \tag{2.14}$$

a vyjadrujú, že akákoľvek sekvencia prechodov σ prípustná v značení m a taká, že $\sigma(t1) = \sigma(t2)$ a $\sigma(t3) = \sigma(t4)$ dovedie sieť po vykonaní v m späť do m.

□

2.1.5 Algoritmické problémy a vlastnosti P/T sietí

Pre Petriho siete bolo sformulovaných viacero problémov, viažúcich sa na podstatné vlastnosti systémov, ktoré opisujú. Definícia najdôležitejších problémov a súvisiacich vlastností pre P/T sieť N_0 (2.1) (strana 11) je nasledovná:

Problém dosiahnuteľnosti (*RP, Reachability Problem*)**.**
Majme PS N_0 a vektor $\vec{q} \in \mathbb{N}^k$. Problém určiť či $\vec{q} \in \mathcal{R}(N_0)$ je nazývaný *problémom dosiahnuteľnosti* (*inštanciou problému dosiahnuteľnosti*) siete N_0 (pre značenie q).

Problém ohraničenosti (*Boundedness Problem*)**.**
Problém rozhodnúť či PS N_0 je ohraničená (bounded) sa nazýva *problém ohraničenosti* siete N_0.

PS N_0 je *ohraničená*, ak $\exists r \in \mathbb{N}$ také, že pre $\forall m \in \mathcal{R}(N_0)$ a pre $\forall p \in P$ platí, že $m(p) \leq r$

Problém pokrytia (*Coverability Problem*).
Majme PS N_0 a $q \in \mathbb{N}^k$. *Problém pokrytia*, $Cow(N_0, q)$, je zistiť či $\exists m \in \mathcal{R}(N_0) : q \leq m$.

Problém reverzibility (*Reversibility Problem*).
Problém rozhodnúť či PS N_0 je reverzibilná (reversible) sa nazýva *problém reverzibility* siete N_0.

PS N_0 je *reverzibilná*, ak $\forall m \in \mathcal{R}(N_0) : \exists \sigma \in T^* : (m \overset{\sigma}{\vdash} m_0)$.

Problém domovského stavu (*Home State Problem*).
Problém rozhodnúť či PS N_0 má domovský stav (home state) sa nazýva *problém domovského stavu* siete N_0.

Značenie $m \in \mathcal{R}(N_0)$ je *domovský stav* (*home state marking*) ak pre $\forall m' \in \mathcal{R}(N_0) : \exists \sigma \in T^* : (m' \overset{\sigma}{\vdash} m)$. Ide o zovšeobecnenie problému reverzibility.

Problém ekvivalencie a obsiahnuteľnosti (*Equality and Containment Problem*).
Majme 2 Petriho siete $N_{0i} = (P_i, T_i, pre_i, post_i, m_{0i}),\ i = 1, 2$, také, že $|P_1| = |P_2|$.

Problém rozhodnúť či sú tieto siete ekvivalentné ($N_{01} \approx N_{02}$), tzn. či $\mathcal{R}(N_{01}) = \mathcal{R}(N_{02})$, sa nazýva *problém ekvivalencie*.

Problém rozhodnúť či N_{01} je obsiahnutá v N_{02} ($N_{01} \leq N_{02}$), tzn. či $\mathcal{R}(N_{01}) \subseteq \mathcal{R}(N_{02})$, sa nazýva *problém obsiahnuteľnosti*.

Deadlock problém (*Deadlock Problem*).
Problém Deadlock(N_0) je určiť či existuje $m \in \mathcal{R}(N_0)$, ktoré je deadlockové.

Značenie $m \in \mathcal{R}(N_0$, je *deadlockové*, ak neexistuje $t \in T$, ktorý je realizovateľný v m.

Problém živosti (*Liveness Problem*).
Problém rozhodnúť či PS N_0 je i-živá, $i = 0, 1, 2, 3, 4$ sa nazýva *problém i-živosti* siete N_0.

Problém rozhodnúť či PS N_0 je živá sa nazýva *problém živosti* siete N_0.

Pojmy živosť a i-živosť vymedzujú nižšie uvedené definície 2.1.11 a 2.1.12.

Živosť

Pre mnohé systémy, napríklad komunikačné protokoly, je veľmi dôležitou vlastnosťou *živosť*, ktorá hovorí o tom, že v žiadnom stave nedôjde k situácii keď výpočet nebude môcť pokračovať. Pre P/T siete existujú 4 úrovne živosti, založené na živosti jednotlivých prechodov. Tie sú definované nasledovne:

Definícia 2.1.11 (živosť prechodu)**.** *Majme P/T sieť*

$$N_0 = (P, T, pre, post, m_0)$$

a prechod $t \in T$*. Prechod* t *sa nazýva*

- 0- živý $(L_0(t)) \Leftrightarrow_{def} \neg(\exists \sigma \in T^* : m_0 \overset{\sigma}{\vdash} \wedge\sigma(t) \geq 1)$
- 1- živý $(L_1(t)) \Leftrightarrow_{def} \exists\, \sigma \in T^* : m_0 \overset{\sigma}{\vdash} \wedge\sigma(t) \geq 1$
- 2- živý $(L_2(t)) \Leftrightarrow_{def} pre\ \forall r \in \mathbb{N}\ \exists \sigma \in T^* : m_0 \overset{\sigma}{\vdash} \wedge\sigma(t) \geq r$
- 3- živý $(L_3(t)) \Leftrightarrow_{def} \exists\, \sigma \in T^* : m_0 \overset{\sigma}{\vdash} \wedge\ \sigma(t) = \omega$
- 4- živý $(L_4(t)) \Leftrightarrow_{def} \forall m \in \mathcal{R}(N_0) : \exists\, \sigma \in T^* : m \overset{\sigma}{\vdash} \wedge\sigma(t) \geq 1$

□

Zápis $\sigma(t)$ je, v súlade s časťou 2.1.1, početnosť výskytu prechodu t v sekvencii σ. Pri 3-živosti je počet výskytov t v σ neobmedzený a 4-živý prechod je vlastne 1-živý v každom značení siete. Jednotlivé typy živosti prechodov a najmä rozdiel medzi 2- a 3-živosťou ilustruje sieť na obr. 2.2, prevzatá z [49].

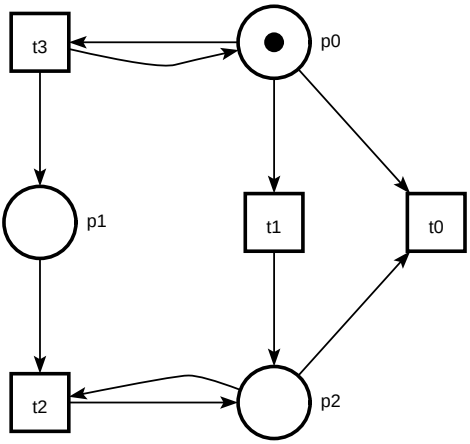
Živosť Petriho siete definujeme na základe živosti jej prechodov:

Definícia 2.1.12 (živosť siete)**.** *Sieť* N_0 *je* i-živá *práve vtedy, ak každý jej prechod je i-živý:*

$$L_i(N_0) \Leftrightarrow_{def} \forall\, t \in T : L_i(t)$$

Petriho sieť N_0 *sa nazýva* živá *práve vtedy, ak je 4-živá:*

$$Live(N_0) \Leftrightarrow_{def} L_4(N_0).$$



Obr. 2.2: Petriho sieť, v ktorej je prechod $t0$ 0-živý, $t1$ 1-živý, $t2$ 2-živý a $t3$ 3-živý.

2.1.6 Problém dosiahnuteľnosti a jeho riešenie

Spomedzi predstavených problémov má výsadné postavenie problém dosiahnuteľnosti (RP), keďže väčšina ostatných sa dá redukovať práve na tento problém [19]. V tejto časti stručne opíšeme pôvodnú metódu analýzy a riešenia problému dosiahnuteľnosti, ktorá je dielom druhého autora a jej vyčerpávajúci opis je možné nájsť v [19], a v nasledujúcej predstavíme autormi navrhnutú [31] formalizáciu jedného z prístupov k de/kompozičnému riešeniu tohto problému, pomenovaného T-Junc.

Metódu, resp. algoritmus z [19] budeme označovať RP_{M_w} a formalizáciu algoritmu pre T-Junc ako $\mathrm{RP/Tjunc}_{M_w}$.

Kľúčovým prvkom metódy RP_{M_w} je štruktúra nazvaná *konečný automat typu* M_w, alebo jednoducho M_w *automat*, či M_w. Jej analýza je založená na výsledkoch teórie automatov a konvexnej analýze stavového priestoru reprezentovaného M_w automatom. Podoba M_w automatu je do veľkej miery podobná tzv. *grafu pokrytia* (coverability graph) [49]. Metóda redukuje RP na tzv. *modifikovaný problém celočíselného lineárneho programovania* (*MILP*, *modified integer linear programming problem*). MILP k štandardnému problému celočíselného lineárneho prog-

ramovania (ILP) pridáva nutnosť preveriť platnosť podmienky *con*, ktorá platí vtedy a len vtedy keď existuje cesta v stavovom priestore z bodu reprezentujúceho riešenie ILP X do počiatočného bodu hyperkocky $C(X)$. Nedávno bolo objavené [20], že na M_w automate je možné založiť nový prístup k odhaľovaniu deadlockových značení v P/T sieťach.

Napriek svojmu veku si metóda RP_{M_w} zachováva svoju jedinečnosť aj v porovnaní s najnovšími prístupmi k riešeniu RP ako sú [41, 45, 46]: metóda [19] je založená na štúdiu jazykových vlastností výpočtov P/T sietí kým ostatné prístupy sú stavovo–orientované. Existuje však silná väzba medzi touto metódou a výsledkami prezentovanými v [45, 46], napríklad grafy dosiahnuteľnosti z [45] a produkčné grafy z [46] majú vlastnosti podobné M_w a množiny (ko-)dosiahnuteľných konfigurácií z [45] sú blízke prefixovým a postfixovým jazykom z [19].

Výhodami RP_{M_w} sú najmä systematický spôsob konštrukcie M_w a použiteľnosť jediného M_w pre všetky inštancie RP danej siete. Navyše bola v [19] odvodená časová zložitosť riešenia RP pre najhorší prípad a odhadnutá dolná hranica jeho priestorovej zložitosti. Horná hranica časovej zložitosti bola stanovená na $O(2^{b^{2^{k+1}}k^2})$, kde $k = |P|$. Je tiež potrebné poznamenať, že v prípade neohraničenej P/T siete M_w automat zodpovedá Kosarajuovej-Lambertovej-Mayrovej-Sacerdoteho-Tenneyho (KLMST) dekompozícii[3]. To je celkom prirodzené, keďže základy tejto metódy a práce Kosaraju [40] a Mayra [48] pochádzajú z rovnakého obdobia, čoho svedkom je práca [28].

Vektorový adičný systém

Aby sa zjednodušila s ním súvisiaca teória nebol algoritmus RP_{M_w} navrhnutý priamo pre P/T siete ale pre *vektorové adičné systémy* (*VAS*), ktoré sú ekvivalentné čistým[4] P/T sieťam. VAS je v [19] definovaný nasledovne:

Definícia 2.1.13 (vektorový adičný systém a jeho výpočet)**.** *Nech* $k \in \mathbb{N}$. *Potom k-rozmerný VAS je dvojica* $W_k = (q_0, W)$, *kde* $q_0 \in \mathbb{N}^k$ *je počiatočný stav* W_k *a* W *konečná množina k-rozmerných celočíselných vektorov.*

[3] Skratka KLSMT je prevzatá z [46].

[4] Pozri (2.2) na strane 12

Nech $q, q' \in \mathbb{N}^k$ a $w \in W$. Potom vykonateľnosť ($q \xrightarrow{w}$) a vykonanie ($q \xrightarrow{w} q'$) w sú definované nasledovne:

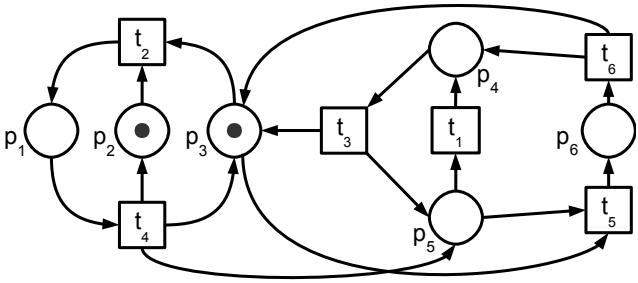
$$\begin{aligned} q \xrightarrow{w} \quad &\Leftrightarrow_{def} \quad q + w \geq 0^k \\ q \xrightarrow{w} q' \quad &\Leftrightarrow_{def} \quad q' = q + w \end{aligned}$$

□

Vykonateľnosť a vykonanie vo VAS možno spôsobom totožným s definíciou 2.1.3 rozšíriť na sekvencie vektorov z W ($\xrightarrow{\sigma}$, $\xrightarrow{*}$). *VAS ekvivalent P/T siete* N_0 (2.1) (strana 11) má tvar

$$W_k(N_0) = (\vec{m_0}, \{\vec{t_1}, \ldots, \vec{t_n}\}).$$

Z uvedeného vyplýva, že RP_{M_w} je použiteľný iba pre čisté siete. Toto obmedzenie je však len technické: každú „nečistú“ P/T sieť vieme vložením dodatočných miest a prechodov upraviť tak, aby vyhovovala podmienke (2.2). Značenia a výpočet sa síce rozšíria o nové prvky, no význam vzhľadom na opisovaný systém sa nezmení. Z tohto dôvodu, a kvôli jednoznačnému vzťahu medzi VAS a algebraickým vyjadrením čistej P/T siete, budeme v tejto práci hovoriť o RP_{M_w} ako o algoritme pre všetky P/T siete a v jeho popise budeme používať termíny P/T sietí a nie VAS. Vzťah VAS a P/T siete ilustruje príklad 2.1.5.



Obr. 2.3: Neohraničená P/T sieť

Príklad 2.1.5 (P/T sieť a VAS)**.** VAS zodpovedajúci neohraničenej

P/T sieti z obrázka 2.3 má tvar

$$\begin{aligned} W_6 &= ((0,1,1,0,0,0), \{\vec{t_1}, \ldots, \vec{t_6}\}), \\ \vec{t_1} &= (0,0,0,1,-1,0), \\ \vec{t_2} &= (1,-1,-1,0,0,0), \\ \vec{t_3} &= (0,0,1,-1,1,0), \\ \vec{t_4} &= (-1,1,1,0,1,0), \\ \vec{t_5} &= (0,0,-1,0,-1,1), \\ \vec{t_6} &= (0,0,1,1,0,-1). \end{aligned}$$

□

Algoritmus $\mathbf{RP}_{M_w}$ riešenia problému dosiahnuteľnosti

Samotný RP_{M_w} algoritmus pre inštanciu $RP(q, N_0)$ možno zhrnúť do nasledujúcich šiestich krokov, ktoré následne bližšie vysvetlíme:

Krok 1 Pre N_0 zostav konečný automat $M_w(N_0)$. Ak stavy $M_w(N_0)$ sú ω-vektory (tzn. vektory obsahujúce aspoň jednu ω), choď na krok 3. Inak choď na krok 2.

Krok 2 V M_w hľadaj stav totožný s q. Ak taký nájdeš, choď na krok 5, inak choď na krok 6.

Krok 3 V M_w hľadaj stav pokrývajúci q. Ak taký nájdeš, choď na krok 4, inak choď na Krok 6.

Krok 4 Zostav a vyrieš modifikovaný problém celočíselného lineárneho programovania $MILP_{N_0}(A, X, B(q))$. Ak výsledok riešenia je *true* choď na krok 5, inak choď na krok 6.

Krok 5 Odpovedz $q \in \mathcal{R}(N_0)$ a skonči.

Krok 6 Odpovedz $q \notin \mathcal{R}(N_0)$ a skonči.

Automat $M_w(N_0)$, vytvorený v prvom kroku, reprezentuje stavový priestor N_0. Je definovaný ako

$$M_w(N_0) = (Q, W, \delta, \rho_0),$$

kde Q je množina stavov, W je vstupná abeceda, $W = T$, $\delta : Q \times W \longrightarrow Q$ je prechodová funkcia a $\rho_0 \in Q$ je počiatočný stav. Každý $q \in Q$ má vektorovú formu

$$\vec{q} \in (\mathbb{N} \cup \{\omega\})^k$$

Podľa toho či N_0 je ohraničená alebo nie, $M_w(N_0)$ patrí do jednej z nasledujúcich kategórií:

RG-podobný. Ak je N_0 ohraničená, potom $M_w(N_0)$ je identický s grafom dosiahnuteľnosti, $Q = \mathcal{R}(N_0)$ a $\vec{m_0} = \vec{\rho_0}$.

CG-podobný. Ak N_0 nie je ohraničená, potom $M_w(N_0)$ je podobný grafu pokrytia. Najväčším rozdielom medzi M_w a grafom pokrytia je, že M_w nemá špeciálny stav pre m_0. V CG-podobnom M_w každý stav $\rho \in Q$ má podobu ω-vektora a reprezentuje nekonečnú podmnožinu $\mathcal{R}(N_0)$, konkrétne všetky $m \in \mathcal{R}(N_0)$ spĺňajúce podmienku $\vec{m} \leq \vec{\rho}$. Prirodzene, $\vec{m_0} \leq \vec{\rho_0}$. Stavy $M_w(N_0)$ sa tiež nazývajú *makrostavmi* a sú navzájom neporovnateľné.

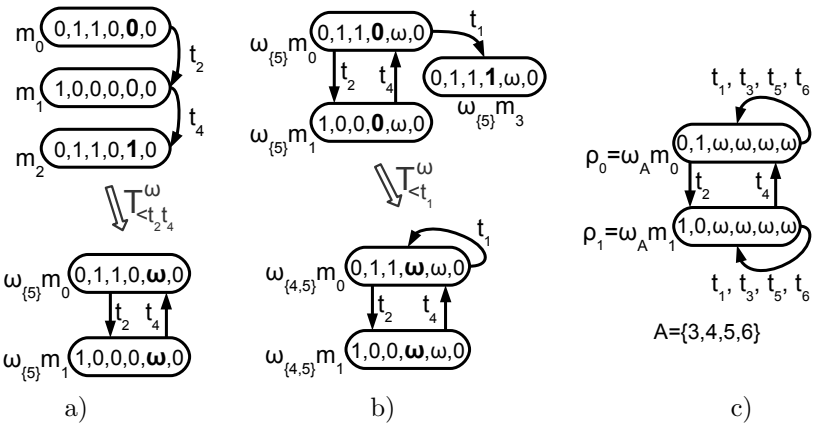
Proces konštrukcie $M_w(N_0)$ je v [19] definovaný ako postupnosť transformácií $T^{\omega}_{<x}$ takzvaného *vektorového stavového stromu* (*VST*) VAS-u $W_k(N_0)$. Tento strom je totožný so stromom dosiahnuteľnosti N_0. Každému stavu q z VST je priradený jazyk

$$Y_q = \{u \in T^* | \exists q' \in \mathbb{N}^k : q \xrightarrow{u} q'\}$$

nazývaný *postfixný jazyk q*. Transformácia $T^{\omega}_{<x}$ nahradí nekonečnú cestu pozostávajúcu z pod-ciest $q \xrightarrow{\pi} q'$, $\pi \in T^*$, kde $q \leq q'$, slučkou $\rho \xrightarrow{\pi} \rho$, $\rho = T^{\omega}_{<x}(q) = T^{\omega}_{<x}(q')$, $\rho = \omega_A q$. Ako $\omega_A q$ označujeme vektor získaný z q nahradením prvkov na súradniciach z A symbolom ω a A, $A \subseteq \{1 \dots k\}$, je množina súradníc, na ktorých platí ostrá nerovnosť medzi q a q'. $T^{\omega}_{<x}$ má invariantnú vlastnosť $\forall q'', q \xrightarrow{*} q'' : Y_{q''} \subseteq Y_{T^{\omega}_{<x}(q'')}$. Postupnosť transformácií $T^{\omega}_{<x}$ je pre každú P/T sieť konečná. Konštrukciu automatu siete z obr. 2.3 približuje nasledujúci príklad.

Príklad 2.1.6 (konštrukcia M_w)**.** V praxi tvorba M_w prebieha tak, že aplikujeme transformácie hneď ako je to možné. To znamená, že pre našu sieť z obr. 2.3 aplikujeme $T^{\omega}_{<t_2 t_4}$ hneď ako zistíme, že $m_0 \leq m_2$. Táto transformácia „omegalizuje“ piatu súradnicu vektorov na príslušnej ceste a vytvára slučku t_2, t_4 (obr. 2.4 a)). V druhom kroku sa ďalšia ω objaví na štvrtej súradnici $\omega_{\{5\}} m_0$ a $\omega_{\{5\}} m_3$. Táto ω sa tiež rozšíri na $\omega_{\{5\}} m_1$ (obr 2.4 b)). Zvyšné transformácie pozostávajú z krokov podobných druhému, a to pre t_3, t_5 a t_6 pre oba makrostavy a t_1 pre druhý makrostav. Nakoniec dostávame M_w automat z obr. 2.4 c).

□



Obr. 2.4: Konštrukcia M_w siete z obr. 2.3: prvý krok (a), druhý krok (b) a výsledný automat (c)

Ak je $M_w(N_0)$ CG-podobný tak jeho stavový diagram pozostáva z $n \geq 1$ *silne súvislých komponentov* a príslušná inštancia RP, $RP(q, N_0)$, sa redukuje na inštanciu[5] (2.15) MILP,

$$MILP_{N_0}(A, X, B(q)) \;=\; ILP_{N_0}(A, X, B(q)) \wedge con_{N_0}(A, X, B(q)). \tag{2.15}$$

Prvá časť MILP je ILP inštancia získaná nasledovne: Predpokladajme, že sme našli makrostav ρ automatu M_w taký, že $q \leq \rho$ a ρ a ρ_0 sú v tom istom scc, v scc_{ρ_0}. Tiež predpokladajme, že v, $v \in T^*$, je jediná cesta z ρ_0 do ρ. Teraz je potrebné nájsť všetky rôzne *jednoduché slučky* v scc_{ρ_0}. Jednoduchá slučka je cyklus (tzn. uzavretá postupnosť prepojených vrcholov) v grafe M_w, ktorý neobsahuje iné cykly. Nech jednoduché slučky scc_{ρ_0} tvoria množinu $W_L = \{\ell_1, \ell_2, \ldots, \ell_{d_0}\}$. Potom môžeme zostaviť sústavu lineárnych rovníc (2.16).

$$\begin{array}{rcl} A.X & = & B(q), \quad B(q) = q^T - \vec{m_0}^T - [v] \\ A & = & ([\ell_1], [\ell_2] \cdots [\ell_{d_0}]) \end{array} \tag{2.16}$$

[5] V [19] je inštancia MILP definovaná ako $MILP_{W_k}(A, X, B(q)) = ILP_{W_k}(A, X_0, B(q)) \wedge con_{W_k}(A, X_0, B_0)$. Jej zápis (2.15) je upravený tak, aby neobsahoval symboly v tejto práci nepoužívané. V podstate W_k je $W_k(N_0)$, X_0 má súvis s minimálnymi riešeniami (2.16) a $B_0 = B(q) - q$.

Vyriešiť ILP_{N_0} znamená vypočítať sústavu rovníc (2.16). Ak nájdeme také jej riešenie X', ktoré je nezáporným celočíselným vektorom, potom $ILP_{N_0}(A, X, B(q)) = true$.

Riešenie (2.16) (vektor X') udáva počet slučiek – sekvencií prechodov, ktoré je potrebné vykonať na ceste z m_0 do q, ale nehovorí o tom či a v akom poradí sú vykonateľné. Preto je potrebný ďalší test v podobe predikátu $con_{N_0}(A, X, B(q))$. Test je o zistení či v N_0 existuje prípustná sekvencia prechodov, v ktorej početnosť jednotlivých prechodov súhlasí s X' a štruktúrou jednoduchých slučiek v scc_{ρ_0}. Ak je stavový priestor N_0 semilineárny tak con_{N_0} nie je potrebné počítať (automaticky platí). Výsledky predstavené v [19] navyše ukazujú, že trieda P/T sietí pre ktoré je postačujúce riešiť ILP_{N_0} môže byť ešte väčšia.

Ak má automat $M_w(N_0)$ viac ako jeden scc a q je pokrytý niektorým stavom ρ, ktorý nie je v počiatočnom scc_{ρ_0} tak množina W_L a matica A v (2.16) musia obsahovať slučky všetkých scc na ceste z ρ_0 do ρ. Ak existuje viac ciest z ρ_0 do ρ tak je potrebné riešiť inštanciu ILP pre každú z nich (resp. až kým sa nenájde nejaká s vyhovujúcim X'). Riešenie $MILP_{N_0}$ ukazuje príklad 2.1.7.

Príklad 2.1.7 (výpočet $MILP_{N_0}$)**.** Nech je našou úlohou zistiť či v P/T sieti z príkladu 2.1.5 je $q = (0, 1, 1, 1, 0, 1)$ dosiahnuteľným značením. Po zostrojení príslušného M_w (obr. 2.1.6 c)) zistíme, že $q \leq \rho_0$. $M_w(N_0)$ je tvorený jediným scc s $W_L = \{\ell_1, \ell_2, \ell_3, \ell_5, \ell_6\}$, kde $\ell_2 = t_2, t_4$ a pre $i = 1, 3, 5, 6$ je $\ell_i = t_i$. Príslušná sústava rovníc podľa (2.16) bude

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 1 & 0 & -1 & 0 & 1 \\ -1 & 1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_5 \\ X_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Jej riešenie je $\vec{X'} = (2, 2, 1, 1, 0)^T$, takže $ILP_{N_0} = true$. Aj $con_{N_0} = true$ pretože existuje $\sigma = t_2t_4t_1t_2t_3t_1t_4t_5$ taká, že $m_0 \stackrel{\sigma}{\vdash} q$. Môžeme teda konštatovať, že $q \in \mathcal{R}(N_0)$.

□

2.1.7 De/kompozičný prístup k dosiahnuteľnosti

Ako už bolo spomenuté, zložitosť RP je obrovská a zdá sa, že jedinou možnosťou ako sa s ňou vysporiadať je použiť pri jeho riešení de/kompozičný prístup. V [18, 19] sú navrhnuté 3 spôsoby de/kompozície P/T siete $N_0 = (P, T, pre, post, m_0)$ na podsiete N_1, N_2,
$N_i = (P_i, T_i, pre_i, post_i, m_{0_i})$, $i \in \{1, 2\}$:

P-Junc dekompozícia je založená na spoločných miestach, ktoré sa vyskytujú v oboch podsieťach: $P_1 \cap P_2 = P_c$, $T_1 \cap T_2 = \varnothing$. Množinu spoločných miest P_c nazývame *množina konformných miest.*

T-Junc dekompozícia je založená na spoločných prechodoch: $T_1 \cap T_2 = T_s$, $P_1 \cap P_2 = \varnothing$. Množinu T_s nazývame *množina synchrónnych prechodov.*

PT-Junc dekompozícia je založená na spoločných miestach aj prechodoch: $P_1 \cap P_2 = P_c$, $T_1 \cap T_2 = T_s$.

Z týchto spôsobov sa ako najsľubnejší javí T-Junc, keďže redukuje zložitosť najefektívnejším spôsobom, môže byť aplikovaný na každú P/T sieť a, ako uvidíme ďalej, má dobré vlastnosti vzhľadom na možnú paralelnú implementáciu. V [31] bol spôsob T-Junc precizovaný a formalizovaný do podoby algoritmu RP/Tjunc$_{M_w}$, ktorý predstavíme v tejto časti.

Znova predpokladajme, že P/T sieť N_0 je definovaná ako v (2.1) (strana 11). Potom jej podsiete N_1, N_2, získané T-junc dekompozíciou budú (2.17)

$$\begin{array}{l} N_z = (P_z, T_z, pre_z, post_z, m_{0_z}),\ z \in \{1, 2\} \\ P_1 \cup P_2 = P,\ P_1 \cap P_2 = \varnothing,\ |P_1| = k_1,\ |P_2| = k_2, \\ T_1 \cup T_2 = T,\ T_1 \cap T_2 = T_s \neq \varnothing, \\ pre_z = (P_z \times T_z) \lhd pre,\ post_z = (P_z \times T_z) \lhd post, \\ m_{0_z} = P_z \lhd m_0 \end{array} \tag{2.17}$$

a tento ich vzájomný vzťah označíme ako $N_0 = Tjunc[T_s(N_1, N_2)]$. Symbol „$\lhd$“ je doménová reštrikcia alebo zúženie (pozri tabuľku 3.7 na strane 59).

Súvislosť riešení RP pre N_1, N_2 a N_0 vyjadruje veta 2.1.4, prevzatá z [18]. V nej $\sigma|_{T_z}$ je projekcia σ na T_z, tzn. sekvenciu $\sigma|_{T_z}$ získame zo σ odstránením všetkých prechodov, ktoré nepatria do T_z.

Veta 2.1.4. *Nech N_0, N_1, N_2 sú P/T siete a $N_0 = Tjunc[T_s(N_1, N_2)]$. Potom pre každý $q \in \mathbb{N}^k$ platí, že $q \in \mathcal{R}(N_0)$ práve vtedy ak platí*

$$\begin{aligned}
&\exists(q_1, q_2) : q_1 \in \mathbb{N}^{k_1} \wedge q_2 \in \mathbb{N}^{k_2} \wedge q = (q_1, q_2) \wedge \\
&\qquad q_1 \in \mathcal{R}(N_1) \wedge q_2 \in \mathcal{R}(N_2) \wedge \\
&\exists(\sigma \in T^*, \sigma_1 \in T_1^*, \sigma_2 \in T_2^*) : \sigma_1 = \sigma|_{T_1} \wedge \sigma_2 = \sigma|_{T_2} \wedge \\
&\qquad m_{0_1} \overset{\sigma_1}{\vdash} q_1 \; v\, N_1 \;\wedge\; m_{0_2} \overset{\sigma_2}{\vdash} q_2 \; v\, N_2 \;\wedge \sigma_1|_{T_s} = \sigma_2|_{T_s}
\end{aligned}$$

Algoritmus RP/Tjunc$_{M_w}$ je založený na vete 2.1.5, ktorá je vlastne preformulovaním vety 2.1.4 do podoby bližšej implementácii T-Junc prístupu. Pred uvedením vety 2.1.5 a jej dôkazu zavedieme precíznejšiu špecifikáciu množín z (2.17)):

$$\begin{aligned}
&P = \{p_1, \ldots, p_a, p_{a+1}, \ldots, p_k\},\; T = \{t_1, \ldots, t_b, \ldots, t_c, \ldots, t_n\} \\
&P_1 = \{p_1, \ldots, p_a\},\; P_2 = \{p_{a+1}, \ldots, p_k\} \\
&T_1 = \{t_1, \ldots, t_c\},\; T_2 = \{t_b, \ldots, t_n\},\; T_s = \{t_b, \ldots, t_c\}
\end{aligned}$$

Pre jednoduchosť predpokladajme, že riešenie RP v N_1 aj N_2 vedie len k jednej ILP inštancii v každej z nich.

Veta 2.1.5. *Nech N_0, N_1, N_2 sú ako vo vete 2.1.4 a $q \in \mathbb{N}^k$, $q_1 \in \mathbb{N}^{k_1}$ a $q_2 \in \mathbb{N}^{k_2}$ sú také, že $q = (q_1, q_2)$. Ďalej nech*

$$\begin{aligned}
&A_z X_z = B(q_z),\; B(q_z) = q_z^T - m_{0_z}^T - [v_z], \\
&W_{L_z} = \{\ell_{z.1}, \ldots, \ell_{z.d_z}\},\; A_z = ([\ell_{z.1}], \ldots [\ell_{z.d_z}]) \\
&X_z = (x_{z.1}, \ldots, x_{z,d_z})^T
\end{aligned} \tag{2.18}$$

sú IPL inštancie pre $RP(q_z, N_z)$ a

$$A'_z X'_z = B'(q_z) \tag{2.19}$$

sú ich explicitné riešenia ($z = 1, 2$). Potom platí, že systém lineárnych rovníc (2.20)

$$\begin{pmatrix} A'_1 & 0_{a,d_2} \\ 0_{k-a,d_1} & A'_2 \\ L_1 & -L_2 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} B'(q_1) \\ B'(q_2) \\ B_{1.2} \end{pmatrix} \tag{2.20}$$

kde

$$\begin{aligned}
&L_z = (L_{z_{i,j}})_{(c-b+1)\times d_z},\; L_{z_{i,j}} = \ell_{z.j}(t_{b+i-1}),\; z = 1, 2 \\
&B_{1.2} = (B_{1,2_1}, \ldots, B_{1,2_{c-b+1}})^T,\; B_{1,2_i} = v_2(t_{b+i-1}) - v_1(t_{b+i-1})
\end{aligned}$$

má nezáporné celočíselné riešenie vtedy a len vtedy ak

$$\begin{aligned}\exists(\sigma_1 \in T_1^*, \sigma_2 \in T_2^*) : & ILP_{N_1}(A_1, X_1, B(q_1)) = true \wedge \\ & ILP_{N_2}(A_2, X_2, B(q_2)) = true \wedge \\ & \forall(t \in T_s) : \sigma_1(t) = \sigma_2(t).\end{aligned} \tag{2.21}$$

□

Dôkaz. Dôkaz vety 2.1.5. Systémy rovníc (2.19) sú explicitnými riešeniami systémov (2.18), takže sú v tvare získanom z (2.18) Gaussovou eliminačnou metódou. Sú ekvivalentné (2.18) a tých istých rozmerov. Je teda postačujúce vetu dokázať pre (2.22) namiesto (2.20).

$$\begin{pmatrix} A_1 & 0_{a,d_2} \\ 0_{k-a,d_1} & A_2 \\ L_1 & -L_2 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} B(q_1) \\ B(q_2) \\ B_{1.2} \end{pmatrix} \tag{2.22}$$

Sústava (2.22) pozostáva z troch (skupín) rovníc:

$$A_1 X_1 = B(q_1) \tag{2.23}$$

$$A_2 X_2 = B(q_2) \tag{2.24}$$

$$\begin{pmatrix} L_1 & -L_2 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = B_{1.2} \tag{2.25}$$

Rovnice (2.23) a (2.24) sú ILP inštancie $ILP_{N_1}(A_1, X_1, B(q_1))$ a $ILP_{N_2}(A_2, X_2, B(q_2))$, ostáva nám teda ukázať, že (2.25) zodpovedá podmienke $\forall(t \in T_s) : \sigma_1(t) = \sigma_2(t)$.

Uvažujme t_{b+i-1}, i-teho člena $T_s = \{t_b, \ldots, t_c\}$. V σ_z môže prechod t_{b+i-1} byť súčasťou úvodnej cesty v_z alebo niektorej zo slučiek z W_{L_z}. Kým v_z sa v σ_z vyskytuje iba raz, slučky môžu byť realizované opakovane. A presný počet ich opakovaní je daný X_z. Môžeme teda napísať

$$\sigma_z(t_{b+i-1}) = v_z(t_{b+i-1}) + \sum_{j=1}^{d_z} \ell_{z.j}(t_{b+i-1}) \cdot x_{z.j} \tag{2.26}$$

Po aplikovaní (2.26) na rovnosť $\sigma_1(t_{b+i-1}) = \sigma_2(t_{b+i-1})$ a nasledovných úpravách dostávame

$$\sum_{j=1}^{d_1} \ell_{1.j}(t_{b+i-1}) \cdot x_{1.j} - \sum_{j=1}^{d_2} \ell_{2.j}(t_{b+i-1}) \cdot x_{2.j} = v_2(t_{b+i-1}) - v_1(t_{b+i-1})$$

čo je presne tvar i-tej rovnice z (2.25), čím vetu možno považovať za dokázanú.

□

Algoritmus RP/Tjunc$_{M_w}$

Teraz sme pripravení sformulovať algoritmus RP/Tjunc$_{M_w}$ na riešenie $RP(q, N_0)$ pomocou T-Junc de/kompozície $N_0 = Tjunc[T_s(N_1, N_2)]$:

Krok 1 Vyrieš ILP inštancie (2.18) pre $RP(q_1, N_1)$ a $RP(q_2, N_2)$ kde $q = (q_1, q_2)$, $q_1 \in \mathbb{N}^{k_1}$, $q_2 \in \mathbb{N}^{k_2}$. Ak obe majú nezáporné celočíselné (nzc) riešenia, choď na krok 2, inak choď na krok 7.

Krok 2 Vyrieš ILP inštanciu (2.20). Jej riešenie vedie k jednému z nasledujúcich troch prípadov:

1. Neexistuje nzc riešenie (2.20). Potom choď na krok 7.
2. Existuje práve jedno nzc riešenie. Potom priraď toto riešenie do nového vektora X, $X \in \mathbb{N}^{d_1+d_2}$ a choď na krok 4.
3. Existuje nekonečne veľa nzc riešení, ktoré tvoria nekonečnú čiastočne usporiadanú množinu Xi_nf. Potom vlož všetky minimálne prvky Xi_nf do novej množiny $X0$ a choď na krok 3.

Krok 3 Ak množina $X0$ neexistuje alebo je prázdna, choď na krok 7. Inak vyber prvý prvok z $X0$, vlož ho do nového vektora X, $X \in \mathbb{N}^{d_1+d_2}$, a choď na krok 4.

Krok 4 Nech X má tvar $X = (x_{1.1}, \ldots, x_{1.d_1}, x_{2.1}, \ldots, x_{2.d_2})$. Podľa (2.27) vytvor množiny Φ_1, Φ_2. Zahoď X a choď na krok 5.

$$\begin{array}{rcl} \Phi_1 & = & (\varphi_1(t_1), \ldots, \varphi_1(t_b), \ldots, \varphi_1(t_c)) \\ \Phi_2 & = & (\varphi_2(t_b), \ldots, \varphi_2(t_c), \ldots, \varphi_2(t_n)) \\ \varphi_z(t_i) & = & v_z(t_i) + \sum_{j=1}^{d_z} \ell_{z.j}(t_i) \cdot x_{z.j} \\ & & i = 1 \ldots n,\ z = 1, 2 \end{array} \tag{2.27}$$

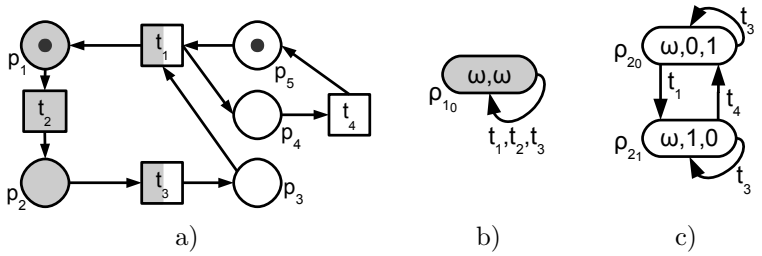
Krok 5 Pre každú dvojicu postupností (σ_1, σ_2) takých, že $\sigma_z \in T_z^*$, $\Psi(\sigma_z) = \Phi_z$, $z = 1, 2$, $\sigma_1|_{T_s} = \sigma_2|_{T_s}$ over ich vykonateľnosť v N_1, resp. N_2. Ak sa nájde pár (σ_1, σ_2) taký, že $m_{0_1} \xrightarrow{\sigma_1} q_1$ v N_1 a $m_{0_2} \xrightarrow{\sigma_2} q_2$ v N_2, skonči overovanie a choď na krok 6. Ak sa po overení všetkých možných (σ_1, σ_2) takýto pár nenájde, choď na krok 3.

Krok 6 Odpovedz $q \in \mathcal{R}(N_0)$ a skonči.

Krok 7 Odpovedz $q \notin \mathcal{R}(N_0)$ a skonči.

Kroky 1 a 2 sú priamou aplikáciou vety 2.1.5. Kroky 3 až 5 sú o systematickom hľadaní páru prípustných sekvencií, ktoré sa synchronizujú na prechodoch z T_s. To znamená sekvencií, kde je rovnaká nie len početnosť ale aj poradie prechodov z T_s. Krok 3 ukazuje, že toto hľadanie je vždy konečné, dokonca aj v prípadoch keď sústava (2.20) má nekonečne veľa riešení. Je to tak preto, lebo táto nekonečnosť je spôsobená skupinami slučiek, ktoré umožňujú znova navštíviť to isté značenie. Je teda postačujúce skúmať iba sekvencie ktoré vedú k len jednej „návšteve“ daného značenia, tzn. sekvencie zostavené podľa minimálnych riešení (2.20). Táto situácia je analogická overovaniu podmienky con_{N_0} (con_{W_k}) v RP$_{M_w}$ algoritme. Konkrétne použitie algoritmu ukážeme na príklade 2.1.8.

Príklad 2.1.8 (aplikácia RP/Tjunc$_{M_w}$)**.** Uvažujme sieť N_0 z obr. 2.5 a). Povedzme, že chceme vedieť či vektor $q = (0, 1, 0, 1, 0)$ je do-



Obr. 2.5: P/T sieť N_0 (a) a M_w automaty jej podsietí N_1 (b) a N_2 (c). Sivé miesta a prechody patria N_1, biele N_2 a sivo–biele obom

siahnuteľným značením N_0. Rozdelíme N_0 na N_1, N_2 tak, že

$$P_1 = \{p_1, p_2\},\ T_1 = \{t_1, t_2, t_3\},\ \vec{m_{0_1}} = (1, 0),\ q_1 = (0, 1)$$
$$P_2 = \{p_3, p_4, p_5\},\ T_2 = \{t_1, t_3, t_4\},\ \vec{m_{0_2}} = (0, 0, 1),\ q_2 = (0, 1, 0)$$

a $T_s = \{t_1, t_3\}$. M_w automaty podsietí sú na obr. 2.5 b) a c) a ich jednoduché slučky sú ako na obr. 2.6. Máme $v_1 = \varepsilon$ (prázdny reťazec)

d	$\ell_{1.d}$	$[\ell_{1.d}]^T$	$\ell_{1.d}(t_1)$	$\ell_{1.d}(t_3)$
1	t_1	$(1,0)$	1	0
2	t_2	$(-1,1)$	0	0
3	t_3	$(0,-1)$	0	1

d	$\ell_{2.d}$	$[\ell_{2.d}]^T$	$\ell_{2.d}(t_1)$	$\ell_{2.d}(t_3)$
1	t_3	$(1,0,0)$	0	1
2	t_1t_4	$(-1,0,0)$	1	0

Obr. 2.6: Jednoduché slučky z príkladu 2.1.8.

a $v_2 = t_1$. Systémy rovníc pre $RP(q_1, N_1)$ a $RP(q_2, N_2)$ sú už explicitnými riešeniami, takže môžeme zostaviť sústavu (2.28) typu (2.20).

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{1.1} \\ x_{1.2} \\ x_{1.3} \\ x_{2.1} \\ x_{2.2} \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \tag{2.28}$$

Sústava má nekonečne veľa *nzc* riešení

$$Xi_nf = \{(r, r+1, r, r, r-1) | r \in \mathbb{N} - \{0\}\}$$

a

$$X0 = \{(1, 2, 1, 1, 0)\}.$$

Po overení dvojíc prípustných sekvencií nájdeme $\sigma_1 = t_2t_3t_1t_2$, $\sigma_2 = t_3t_1$, $\sigma_1|_{T_s} = \sigma_2|_{T_s} = t_3t_1$ a konštatujeme, že $q \in \mathcal{R}(N_0)$.

□

Príklad ukazuje, že algoritmus RP/Tjunc$_{M_w}$ môže byť použitý aj pre ohraničené P/T siete, no je potrebné preskúmať za akých okolností je to efektívnejšie ako prehľadávanie veľkých stavových priestorov. Je tiež zaujímave, že pri sieti z obr. 2.5 a) sa nepodarilo nájsť prípad kedy by sústava (2.20) mala nejaké nzc riešenie a nebolo by možné nájsť σ_1, σ_2 také, že $\sigma_1|_{T_s} \neq \sigma_2|_{T_s}$. Túto vlastnosť sme pozorovali aj pri iných P/T sieťach, čo vzbudzuje nádej, že pre istú podtriedu P/T sietí bude možné kroky 3 až 5 z algoritmu vylúčiť.

Pred sformulovaním vety 2.1.5 sme predpokladali, že riešenie ILP pre N_1 a N_2 vedie iba k jednej inštancii v každej z nich. Ak máme viac inštancií, je potrebné RP/Tjunc$_{M_w}$ vykonať pre každú dvojicu, kde prvá inštancia je z N_1 a druhá z N_2, kým nedostaneme odpoveď $q \in \mathcal{R}(N_0)$ alebo všetky dvojice nevyskúšame.

Možnosti rozšírenia a paralelizácie

Algoritmus RP/Tjunc$_{M_w}$ možno jednoducho rozšíriť pre T-Junc dekompozíciu s viac ako dvoma podsieťami za predpokladu, že ich množiny synchrónnych prechodov sú navzájom disjunktné. Napríklad, dekompozícia na tri podsiete $N_z = (P_z, T_z, pre_z, post_z, m_{0_z})$, $z = 1.2.3$, ktorú označíme ako $N_0 = Tjunc[T_{s_{1.2}}(N_1, N_2), T_{s_{2.3}}(N_2, N_3)]$, je definovaná analogicky s (2.17) s výnimkou množiny synchrónnych prechodov, kde $T_1 \cap T_2 = T_{s_{1.2}}$, $T_2 \cap T_3 = T_{s_{2.3}}$, $T_1 \cap T_3 = \varnothing$, $T_{s_{1.2}} \cap T_{s_{2.3}} = \varnothing$. Potom namiesto sústavy (2.20) máme

$$\begin{pmatrix} A'_1 & 0_{|P_1|,d_2} & 0_{|P_1|,d_3} \\ 0_{|P_2|,d_1} & A'_2 & 0_{|P_2|,d_3} \\ 0_{|P_3|,d_1} & 0_{|P_3|,d_2} & A'_3 \\ L_1 & -L_2 & 0_{|T_{s_{1.2}}|,d_3} \\ 0_{|T_{s_{2.3}}|,d_3} & L_2 & -L_3 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} B'(q_1) \\ B'(q_2) \\ B'(q_3) \\ B_{1.2} \\ B_{2.3} \end{pmatrix}$$

kde všetky prvky sú definované analogicky s (2.20). Kroky 3 až 5 algoritmu RP/Tjunc$_{M_w}$ musia byť modifikované tak, aby overovali sekvencie pre $T_{s_{1.2}}$ aj $T_{s_{2.3}}$.

RP/Tjunc$_{M_w}$ umožňuje vysoký stupeň paralelizmu s minimálnymi požiadavkami na synchronizáciu: V kroku 1 môže byť každá ILP inštancia riešená ako samostatná úloha. V kroku 2 je tiež možné paralelizovať: Pred riešením celej sústavy sa dá Gaussova metóda aplikovať samostatne na podsústavy rovníc pre každú $T_{s_{i,j}}$ (tzn. tie zahŕňajúce L_i, L_j a $B_{i.j}$). Kroky 3 a 4 sú paralelizovateľné na toľko úloh, koľko je množín synchrónnych prechodov v danej dekompozícii. V kroku 5 prakticky každú dvojicu (σ_i, σ_j) možno overiť v samostatnej úlohe, no prakticky kejšie bude v rámci jednej úlohy preverovať páry z množiny

$$\sigma_{ij}^{\delta} = \{(\sigma_a, \sigma_b) \mid \sigma_a \in T_i, \sigma_b \in T_j,\ \sigma_a|_{T_{s_{i,j}}} = \sigma_b|_{T_{s_{i,j}}} = \delta\}.$$

Iné prístupy

Výsledky súvisiace s RP/Tjunc$_{M_w}$ algoritmom sú najpríbuznejšie prácam G. Tonţa [54] a H. Yena [57, 58]. Práca [54] je tiež založená na výsledkoch z [18, 19] a prezentuje algoritmus pre PT-Junc dekompozíciu na dve podsiete. Dekompozičná stratégia je tu založená na štrukturálnych vlastnostiach danej siete, odvodených z jej incidenčnej matice. Algoritmus z [54] môže po modifikácii byť použitý na samotnú dekompozíciu pred aplikovaním RP/Tjunc$_{M_w}$. V [57], a nasledovnej práci [58], je navrhnutá dekompozičná technika pre podtriedu P/T sietí so semilineárnymi množinami dosiahnuteľných značení. Dekompozícia použitá v [57] môže byť považovaná za špeciálny prípad PT-Junc s vlastnosťou, že každá prípustná sekvencia môže byť rozdelená na podsekvencie, i-ta podsekvencia pozostáva len z prechodov i-tej podsiete. Táto technika nie je vždy použiteľná, no nevyžaduje hľadanie konkrétnych sekvencií odpálení a RP inštancie možno riešiť iba na základe ILP.

2.2 Farbené Petriho siete

Farbené Petriho siete (Coloured Petri Nets, CPN) predstavujú jednu z najúspešnejších modifikácií P/T sietí. Ide o tzv. technickú modifikáciu, čo znamená, že CPN majú vyjadrovaciu silu totožnú s P/T sieťami (a teda v princípe je pre ne použiteľný rovnako silný analytický aparát). Za úspechom CPN však stojí skutočnosť, že ponúkajú podstatne väčšiu modelovaciu silu. Obmedzená modelovacia sila P/T sietí, a iných typov PS so značkami, ktoré od seba nie je možné odlíšiť, totiž spôsobovala, že pri zložitejších systémoch sa siete stávali príliš rozsiahlymi. Tým sa strácala aj výhoda grafickej reprezentácie. S týmto problémom sa snažili vysporiadať mnohí výskumníci zavedením rôznych rozšírení P/T sietí, ktoré obsahovali nové koncepty ako časovanie, vplyv externých udalostí na činnosť siete, globálne premenné modifikované sieťou a podmieňujúce odpálenie prechodov či prioritu prechodov. Nevýhodou väčšiny týchto modelov bolo ich obmedzenie na použitie v špecifických oblastiach a chýbajúci analytický aparát [23], ktorý bol „obetovaný“ kvôli zvýšeniu vyjadrovacej sily. Významné [23] z tohto hľadiska bolo predstavenie *predikátovo-prechodových sietí* (*PrT-siete*, *Predicate–Transition Nets*) [15] v roku 1981.

PrT siete patria medzi tzv. *vysokoúrovňové* (high-level) PS, ktorých

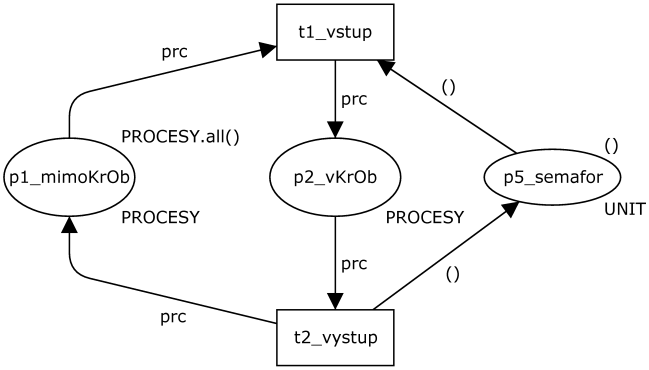
základnou črtou sú individualizované značky, tzn. značky, ktorým môže byť priradená hodnota. Obsahujú aj ďalšie koncepty, ako výrazy na hranách a prechodoch a premenné, potrebné na manipuláciu so značkami a ich hodnotami. Ich vyjadrovacia sila však ostala rovnaká ako u P/T sietí. PrT siete však trpeli istými problémami spojenými s analýzou, najmä pri interpretácii získaných invariantov. Za účelom riešenia týchto problémov bola v [21] predstavená prvá verzia Farbených PS, neskôr označovaná ako CP^{81}-siete [23]. CP^{81}-siete síce tento problém riešili, no výrazy na hranách boli horšie čitateľné a pochopiteľné ako tie v PrT sieťach. Keďže si boli PrT a CP^{81} siete veľmi podobné, vznikol kombinovaný model, nazvaný vysokoúrovňové Petriho siete (HL-nets), ktorý bol predstavený v [22]. Tento názov sa však začal používať aj pre PrT a CP^{81} siete, preto boli HL-nets premenované na farbené Petriho siete (CPN). Pre CPN bol vyvinutý modelovací, simulačný a analytický nástroj *CPN Tools*, voľne dostupný z [62]. V ňom boli vytvorené aj príklady uvedené v tejto časti.

Základným rozdielom nedzi CPN a P/T sieťami teda je, že v CPN sú značky (tokeny) individualizované - každá má svoj typ (*množinu farieb*) a hodnotu (*farbu*). Počet, typ a podmienky kladené na značky ktoré sa zúčastňujú realizácie prechodov sú vyjadrené pomocou *sieťových výrazov*, kde patria *hranové výrazy* (priradené hranám siete) a *stráže* (priradené prechodom siete). Neindividualizované značky majú v CPN priradený typ `UNIT` obsahujúci jedinú hodnotu „`()`“.

V tejto časti je stručne uvedená definícia nehierarchickej CPN a jej správania. Definície sú prevzaté z [23], kde možno nájsť aj ich podrobnejšie vysvetlenie, vrátane príkladu, a z [26], čo je druhá časť trojzväzkového diela [25], [26], [24], ktoré sa farbenými Petriho sieťami zaoberá podrobne. Aktuálnou knižnou publikáciou venujúcou sa CPN a ich aplikácii je [27]. Predtým ako prejdeme k formálnym definíciám uvedieme príklad ukazujúci čo sa za vyššou modelovacou silou CPN skrýva.

Príklad 2.2.1 (MutEx ako CPN)**.** P/T sieť MutEx z príkladov 2.1.1 až 2.1.4 síce jednoducho modeluje vzájomné vylúčenie dvoch procesov, no ak by sme chceli pridať ďalší proces, jej štruktúra by sa rozrástla o časť zodpovedajúca miestam $p1$, $p2$, prechodom $t1$, $t2$ a hranám spájajúcim ich navzájom a s $p5$. To je preto, že jednotlivé procesy nie je možné odlíšiť inak ako umiestnením príslušných značiek do rôznych miest. Naproti tomu v CPN vieme značkám priradiť hodnoty vyjad-

rujúce ich príslušnosť k procesu. Potom si vystačíme s dvoma miestami (`p1_mimoKrOb`, `p2_vKrOb`) pre všetky procesy, samotné procesy odlíšime hodnotou značky. Podobne bude stačiť jeden prechod pre vstup (`t1_vstup`) do a jeden pre opustenie (`t2_vystup`) kritickej oblasti. Graf siete je na obr. 2.7. Okrem grafu CPN obsahuje aj textové



Obr. 2.7: Graf CPN MutEx

deklarácie, definujúce v nej použité množiny farieb, funkcie, premenné a konštanty. Pre MutEx to budú množiny `UNIT` a `PROCESY`, konštanta `maxPr` a premenná `prc`:

```
colset UNIT = unit;
colset PROCESY = index pr with 1..maxPr;
val maxPr=5;
var prc: PROCESY
```

Množina farieb `PROCESY` je typom pre miesta `p1_mimoKrOb` a `p2_vKrOb` a obsahuje hodnoty `pr(1)` až `pr(5)`, keďže `maxPr`= 5. Miesto `p1_mimoKrOb` obsahuje v počiatočnom značení značky pre všetky hodnoty z `PROCESY` (dané inicializačným výrazom `PROCESY.all()`). Odpálením `t1_vstup` sa odoberie jedna zo značiek z `p1_mimoKrOb` a značka s tou istou hodnotou sa pridá do `p2_vKrOb`. (hodnota sa zachová vďaka priradeniu do premennej `prc`). Tiež sa odoberie značka z `p5_semafor`. Prechod `t2_vystup` spôsobí opačnú zmenu značenia. Na rozdiel od P/T siete tu na zvýšenie počtu procesov postačuje zvýšiť hodnotu konštanty `maxPr`.

□

Formálnu definíciu CPN začneme uvedením výrazov, ktoré budeme v ďalšom používať (tab. 2.1).

Výraz	**Význam**
$\mathbb{N}$	Množina prirodzených čísel vrátane nuly.
$\mathbb{B}$	Booleovský typ, $\mathbb{B} = \{true, false\}$.
$[S \to T]$	Množina všetkých funkcií z S do T. S a T sú množiny.
$[S \to T]_L$	Množina všetkých lineárnych funkcií z S do T.
$Type(e)$	Typ výrazu e.
$Var(e)$	Množina premenných vo výraze e.

Tabuľka 2.1: Výrazy používané v definíciách k CPN

Pred uvedením definície CPN je nutné uviesť definíciu *multi-množín*, ktoré sú používané na vyjadrenie značení a taktiež v sieťových výrazoch.

Definícia 2.2.1. Multi-množina m *nad neprázdnou množinou* S *je funkcia* $m \in [S \to \mathbb{N}]$*, ktorú reprezentujeme ako formálny súčet:*

$$\sum_{s \in S} m(s)'s$$

Ako S_{MS} *označujeme množinu všetkých multi-množín nad* S*. Čísla z* $\{m(s) |\ s \in S\}$ *sú* koeficienty *multi-množiny. Platí, že* $s \in m \Leftrightarrow m(s) \neq 0$*.*

Číslo $m(s)$ teda udáva počet výskytov prvku s v multi-množine m. Napríklad multi-množina $2'a + 3'j + 1'k$ je vlastne $\{a, a, j, j, j, k\}$. V angličtine sa pre multimnožinu používa aj výraz „bag“. Pre multi-množiny existuje niekoľko štandardných operátorov.

Definícia 2.2.2. Sčítanie (2.29)*,* skalárne násobenie (2.30)*,* porovnanie (2.31)*,* (2.32)*,* veľkosť (2.33) *a* rozdiel *(*(2.34) *sú definované pre*

$m, m_1, m_2 \in S_{MS}$ a $n \in \mathbb{N}$ takto:

$$m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))'s \tag{2.29}$$

$$n * m = \sum_{s \in S} (n * m(s))'s \tag{2.30}$$

$$m_1 \neq m_2 = \exists s\,(s \in S) : m_1(s) \neq m_2(s) \tag{2.31}$$

$$m_1 \leq m_2 = \forall s\,(s \in S) : m_1(s) \leq m_2(s) \tag{2.32}$$

$$|m| = \sum_{s \in S} m(s) \tag{2.33}$$

$$m_1 - m_2 = \sum_{s \in S} (m_1(s) - m_2(s))'s \tag{2.34}$$

Rozdiel (2.34) *je definovaný len ak* $m_2 \leq m_1$.

Operátory porovnania $=, \geq, >, <$ sú definované rovnakým spôsobom, ako $\neq$ a $\leq$ v (2.31) a (2.32).

2.2.1 Definícia farbenej Petriho siete a jej výpočtu

Definícia 2.2.3. *Farbená Petriho sieť je 9-tica*

$PN_C = (\Sigma, P, T, A, N, C, G, E, I)$, *kde:*

Σ *je konečná množina neprázdnych* typov *(*množín farieb*),*
P *je konečná množina* miest,
T *je konečná množina* prechodov,
A *je konečná množina* hrán, *takých že:*
$P \cap T = P \cap A = T \cap A = \varnothing$,
N *je* funkcia uzlov. *Je definovaná z* A *do* $P \times T \cup T \times P$,
C *je* funkcia farieb. *Je definovaná z* P *do* Σ,
G *je* funkcia stráží. *Je definovaná z* T *do výrazov takých, že:*
$\forall t\,(t \in T) : (Type(G(t)) = \mathbb{B} \wedge Type(Var(G(t))) \subseteq \Sigma)$,
E *je* funkcia hranových výrazov. *Je definovaná z* A *do výrazov takých, že:*
$\forall a\,(a \in A) : (Type(E(a)) = C(p)_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma)$,
kde $N(a) = (p, t) \vee N(a) = (t, p)$,
I *je* inicializačná funkcia. *Je definovaná z* P *do uzavretých výrazov takých, že:* $\forall p\,(p \in P) : Type(I(p)) = C(p)_{MS}$.

Množina $\mathbb{B}$ a funkcie $Type$ a Var sú definované v tabuľke 2.1. Uzavretý výraz je taký, v ktorom sa nevyskytujú (voľné) premenné.

Aby bola definícia CPN úplná, je nutné definovať jazyk, v ktorom sa špecifikujú sieťové výrazy. Pre CPN, ako sú používané v CPN Tools sa používa funkcionálny programovací jazyk *SML*, presnejšie jeho mierne modifikovaný variant nazvaný *CPN ML*. Môže však ísť aj o iný jazyk, no musí spĺňať požiadavky uvedené v [23].

V ďalších definíciách, týkajúcich sa *výpočtu CPN*, bude použitá nasledujúca notácia pre všetky $t \in T$ a $(x_1, x_2) \in P \times T \cup T \times P$:

- $A(t) = \{a \in A |\ N(a) \in (P \times \{t\}) \cup (\{t\} \times P)\}$,
- $Var(t) = \{v |\ v \in Var(G(t)) \vee \exists a.(a \in A(t) \wedge v \in Var(E(a)))\}$,
- $A(x_1, x_2) = \{a \in A |\ N(a) = (x_1, x_2)\}$,
- $E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$,
- $Expr < b >$ je hodnota získaná vyhodnotením výrazu $Expr$ pri priradení hodnôt premenným určeným väzbou b.

Definícia 2.2.4. Väzba *(binding) prechodu t je funkcia b definovaná na $Var(t)$, taká že*

- $\forall v\,(v \in Var(t)) : b(v) \in Type(v)$ *a*
- $G(t) < b >= true$.

$B(t)$ je množina všetkých väzieb prechodu t.

Definícia 2.2.5. Tokenový element *(token element) je dvojica (p, c), kde $p \in P$ a $c \in C(p)$.* Väzobný element *(binding element) je dvojica (t, b), kde $t \in T$ a $b \in B(t)$.*

TE je množina všetkých tokenových elementov a BE je množina všetkých väzobných elementov.

Definícia 2.2.6. Značenie *(marking) je multi-množina nad TE a* krok *je neprázdna konečná multi-množina nad BE. $\mathbb{M}$ je množina všetkých značení a $\mathbb{Y}$ je množina všetkých krokov.*

Počiatočné značenie M_0 *je značenie, ktoré vznikne ohodnotením inicializačných výrazov:* $\forall (p, c)\,((p, c) \in TE) : M_0(p, c) = (I(p))(c)$.

Každé značenie $M \in TE_{MS}$ určuje takú jedinečnú funkciu M^* definovanú na P, že $M^*(p) \in C(p)_{MS}$:

$$\forall p \forall c\,(p \in P \wedge c \in C(p)) : (M^*(p))(c) = M(p,c)$$

Na druhej strane každá funkcia M^*, definovaná na P, taká že $M^*(p) \in C(p)_{MS}$ pre všetky $p \in P$, určuje jedinečné značenie M:

$$\forall (p,c)\,((p,c) \in TE) : M(p,c) = (M^*(p))(c)$$

Preto sa značenia často reprezentujú ako funkcie definované na P a používa sa rovnaké meno (M) pre funkčnú (M^*) aj multi-množinovú (M) reprezentáciu značenia.

Definícia 2.2.7. *Krok Y je* realizovateľný *(enabled) v značení M ak platí:*

$$\forall p\,(p \in P) : \sum_{(t,b) \in Y} E(p,t)<b> \quad \leq M(p)$$

Potom hovoríme, že (t,b) je realizovateľný a tiež, že t je realizovateľný. Elementy kroku Y sú súbežne realizovateľné *(concurrently enabled), ak $|Y| \geq 1$.*

Ak Y je realizovateľný v značení M_1, môže sa realizovať; tým sa značenie zmení z M_1 na M_2, ktoré je definované takto:

$$\forall p\,(p \in P) : M_2(p) = \Big(M_1(p) - \sum_{(t,b) \in Y} E(p,t)<b>\Big) + \sum_{(t,b) \in Y} E(t,p)<b>$$

Hovoríme, že M_2 je priamo dosiahnuteľný *(directly reachable) z M_1, čo zapisujeme ako $M_1[Y> M_2$ alebo ako $M_1 \overset{Y}{\vdash} M_2$.*

□

Podobne ako v P/T sieťach namiesto výrazu „realizácia“ (kroku alebo prechodu) tiež používame výrazy „výskyt“ alebo „odpálenie“.

Definícia 2.2.8. Konečná postupnosť výskytov *(finite occurrence sequence) je postupnosť značení a krokov:*

$$M_1[Y_1> M_2[Y_2> M_3 \ldots M_n[Y_n> M_{n+1}$$

taká, že $n \in \mathbb{N}$, a $M_i[Y_i > M_{i+1}$ pre každé $i \in \{1,2,\ldots,n\}$. M_1 je štartovacie značenie, M_{n+1} koncové a n je dĺžka postupnosti.

Značenie M'' je dosiahnuteľné *zo značenia M' práve vtedy ak existuje konečná postupnosť výskytov začínajúca v M' a končiaca v M''.*

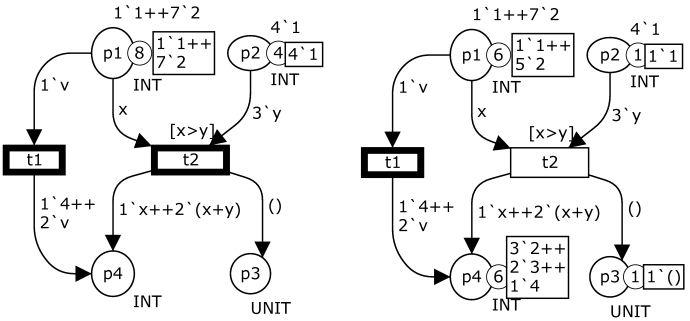
Množinu značení dosiahnuteľných z M označujeme $[M>$. Značenie je dosiahnuteľné práve vtedy ak patrí do $[M_0>$.

Tu uvedené definície približuje príklad 2.2.2.

Príklad 2.2.2 (CPN a jej definícia)**.** CPN na obr. 2.8, na rozdiel od MutEx, nemá nonkrétny význam, no ilustruje zložitejšie manipulácie so značkami pri odpaľovaní prechodov. Miesta `p1`, `p2`, `p4` obsahujú značky typu celé číslo (`INT`), `p3` neindividualizované značky (`UNIT`). Premenné siete (`v`, `x`, `y`) sú všetky typu `INT`. Inicializačné a hranové výrazy majú podobu multi-množín. Oproti definícii 2.2.1 sa pri ich zápise používa dvojité plus, takže napríklad `p1` má v počiatočnom značení (obr. 2.8 a)) osem značiek, jednu hodnoty `1` a sedem hodnoty `2`, čo je dané inicializačným výrazom

```
1‘1++7‘2.
```

Na obr. 2.8 sú znázornené aj aktuálne značenia miest: v krúžku, pri každom mieste ktoré obsahuje aspoň jednu značku, je ich počet a následne v obdĺžniku presný zápis značenia ako multi-množiny. Na obr. 2.8 a) sú tieto výrazy zhodné s inicializačnými výrazmi, ktoré sú uvedené v pravom hornom rohu každého miesta s neprázdnym počiatočným značením. Hranové výrazy definujú multi-množiny značiek, ktoré sú pri odpálení prechodu odobrané, resp. pridané. Napríklad pri odpálení prechodu `t1` sa z `p1` odoberie jedna značka hodnoty zhodnej s hodnotou priradenou premennej `v` (výraz `1‘v`) a do `p4` pridajú dve značky hodnoty premennej `v` a jedna hodnoty `4` (výraz `1‘4++2‘v`). V počiatočnom značení sú vykonateľné oba prechody, čo je znázornené hrubým orámovaním na obr. 2.8 a). Prechod `t1` je možné odpáliť pri väzbe `v`= 1 aj `v`= 2, no prechod `t2` iba pri väzbe `x`= 2 a `y`= 1, keďže premenné musia spĺňať stráž prechodu `t1` – výraz `x>y`. Ak sa manipuluje s jediným tokenom nemusíme počet uvádzať (`x` je to isté ako `1‘x`). Na výraze hrany z `t2` do `p4` môžeme vidieť, že hodnoty nových značiek sa dajú vypočítať z odobraných pomocou operátorov a funkcií (tu operátor sčítania). Na obr. 2.8 b) je sieť po odpálení prechodu `t1` aj `t2` pri väzbe `v`= 2, `x`= 2 a `y`= 1. Prechod `t1` je stále možné odpáliť, `t2` už nie, keďže v `p2` je nedostatočný počet tokenov.



Obr. 2.8: Farbená PS v počiatočnom značení (a) a po vykonaní prechodov t1 a t2 (b)

Formálne, v zmysle definície 2.2.3, túto sieť zapíšeme nasledovne:

$$
\begin{aligned}
CPN_{frg} = \;& (\Sigma, P, T, A, N, C, G, E, I) \\
P = \;& \{\mathtt{p1}, \mathtt{p2}, \mathtt{p3}, \mathtt{p4}\}, \\
T = \;& \{\mathtt{t1}, \mathtt{t2}\} \\
A = \;& \{a_{11}, a_{12}, a_{22}, a_{14}, a_{24}, a_{23}\} \\
N = \;& \{(a_{11}, (\mathtt{p1}, \mathtt{t1})), (a_{12}, (\mathtt{p1}, \mathtt{t2})), (a_{22}, (\mathtt{p2}, \mathtt{t2})), \\
& (a_{14}, (\mathtt{t1}, \mathtt{p4})), (a_{24}, (\mathtt{t2}, \mathtt{p4})), (a_{23}, (\mathtt{t2}, \mathtt{p3}))\} \\
C = \;& \{(\mathtt{p1}, (\mathtt{INT}), (\mathtt{p2}, (\mathtt{INT}), (\mathtt{p3}, (\mathtt{UNIT}), (\mathtt{p4}, (\mathtt{INT})\} \\
G = \;& \{(\mathtt{t1}, \mathtt{true}), (\mathtt{t2}, \mathtt{x>y})\} \\
E = \;& \{(a_{11}, \mathtt{1`v}), (a_{12}, \mathtt{x}), (a_{22}, \mathtt{3`y}), (a_{14}, \mathtt{1`4++2`v}), \\
& (a_{24}, \mathtt{1`x++2`(x+y)}), (a_{23}, \mathtt{()})\} \\
I = \;& \{(\mathtt{p1}, (\mathtt{1`1++7`2}), (\mathtt{p2}, (\mathtt{4`1}), (\mathtt{p3}, \varnothing), (\mathtt{p4}, \varnothing)\}
\end{aligned}
$$

Počiatočné značenie siete bude

$$M_0(\mathtt{p1}) = 1'1 + 7'2, M_0(\mathtt{p2}) = 4'1, M_0(\mathtt{p3}) = M_0(\mathtt{p4}) = \varnothing$$

a vykonaním kroku Y,

$$Y = 1'(\mathtt{t1}, (\mathtt{v}, 2)) + 1'(\mathtt{t2}, \{(\mathtt{x}, 2), (\mathtt{y}, 1)\})$$

v M_0 dosiahneme značenie M_1 ($M_0[Y> M_1$), definované ako

$$M_1(\mathtt{p1}) = 1'1 + 5'2,\ M_1(\mathtt{p2}) = 1'1,\ M_1(\mathtt{p3}) = 1'() \ a$$
$$M_1(\mathtt{p4}) = 3'2 + 2'3 + 1'4.$$

□

2.2.2 Analytické vlastnosti farbenej Petriho siete

Farbené Petriho siete majú rovnakú vyjadrovaciu silu, ako P/T siete. Každú CPN, ktorej Σ obsahuje len konečné množiny, môžeme previesť na výpočtovo ekvivalentnú P/T sieť. Ak by bola niektorá množina farieb nekonečná (tzn. Σ by obsahovala aj nekonečnú množinu), potom by sme pri pokuse o prevod dospeli k P/T sieti s nekonečnou štruktúrou.

Keďže majú CPN aj P/T siete rovnakú vyjadrovaciu silu, mali by byť aj rovnako analyzovateľné. Naozaj, aj pre CPN existujú metódy pre automatický výpočet invariantov miest (S-invarianty) a prechodov (T-invarianty), ktoré sú stručne popísané v tejto časti. Tie vychádzajú z metód pre P/T siete, no ich výpočet je oveľa komplikovanejší. Najprv uvedieme reprezentáciu CPN incidenčnou maticou.

Definícia 2.2.9. Incidenčná matica *CPN je matica* IM *typu* $m \times n$, *kde* $m = |P|$ *a* $n = |T|$, *s prvkami, ktoré sú funkciami:*

$$\begin{aligned} IM(p,t) &\in [B(t)_{WS} \to C(p)_{WS}]_L \\ IM(p,t)(b) &= E(t,p)<b> - E(p,t)<b>,\ b \in B(t). \end{aligned}$$

Výraz $E(t,p) < b > -E(p,t) < b >$ je tzv. *váženou množinou* (weighted set), ktorá je definovaná ako multi-množina kde sú povolené aj záporné koeficienty. Operácia odčítania je tu vždy realizovateľná a môžeme násobiť aj záporným číslom. Množinu všetkých vážených množín nad množinou A označíme A_{WS}.

Definícia 2.2.10. S-invariant *nehierarchickej CPN je množina funkcií*

$$Ws = \{W_p | p \in P\},\ kde\ W_p \in [C(p) \to A_{WS}]_L,\ A \in \Sigma$$

taká, že

$$\forall M\,(M \in [M_0>) : \sum_{p \in P} W_p(M(p)) = \sum_{p \in P} W_p(M_0(p)).$$

Funkcie W_p sú tzv. *váhy miest* (place weights). Sú definované ako $W_p \in [C(p) \to A_{WS}]_L$, kde $A \in \Sigma$ je typ zdieľaný všetkými váhami. Funkciu W_p možno rozšíriť na $\underline{W_p} \in [C(p)_{WS} \to A_{WS}]_L$, definovanú ako:

$$\underline{W_p}(m) = \sum_{c \in C(p)} m(c) * W_p(c),\ m \in C(p)_{WS}$$

Keďže nie je nutné rozlišovať medzi W_p a jej lineárnym rozšírením $\underline{W_p}$, budeme obe označovať W_p. Samotný *S-invariant* získame ako riešenie rovnice

$$WS * IM = \vec{0}$$

kde

$\vec{0}$ je stĺpcový vektor nulových funkcií, ktoré každý argument zobrazia do $\varnothing$ ($\varnothing$ je prázdna vážená množina),

IM je icidenčná matica,

$*$ je kompozícia funkcií W_p a $IM(p,t)$,

WS je riadkový vektor, ktorého prvkami sú funkcie z množiny Ws (je to vlastne vektorová podoba Ws).

Vektor WS je riešením rovnice a určuje S-invariant.

T-invariant je definovaný analogicky k S-invariantu, aj keď v tomto prípade nie je invariantom priamo množina váhových funkcií.

Definícia 2.2.11. *Pre nehierarchickú CPN* množina váh prechodov *(transition weights) s doménou $A \in \Sigma$ je množina funkcií*

$$Wtr = \{W_t | t \in T\},\ kde\ W_t \in [A_{WS} \to B(t)_{WS}]_L.$$

Wtr je prechodový tok *(transition flow) práve vtedy ak:*

$$\forall a \forall p\,(a \in A \wedge p \in P) : \sum_{(t,b) \in Wtr(a)} E(p,t) < b > = \sum_{(t,b) \in Wtr(a)} E(t,p) < b >$$

Konečná multi-množina $Y \in BE_{MS}$ je T-invariant *práve vtedy ak:*

$$\forall p\,(p \in P) : \sum_{(t,b) \in Y} E(p,t) < b > = \sum_{(t,b) \in Y} E(t,p) < b >$$

□

$Wtr(a)$ je množina $Wtr(a) = \{Wtr(a) |\ t \in T \wedge a \in A_{WS}\}$. Vzťah prechodového toku a T-invariantu vyjadruje nasledujúca veta:

Veta 2.2.1. *Wtr je prechodový tok práve vtedy, ak pre každé $a \in A$ je $Wtr(a)$ T-invariant.*

Prechodové toky, z ktorých určíme T-invarianty, dostaneme ako riešenie rovnice

$$IM * WT = \vec{0}$$

kde

$\vec{0}$ je stĺpcový vektor nulových funkcií, ktoré každý argument zobrazia do $\varnothing$,

IM je icidenčná matica,

$*$ je kompozícia funkcií $IM(p,t)$ a W_t,

WT je riadkový vektor, ktorého prvkami sú funkcie z množiny Wtr (je to vlastne vektorová podoba Wtr).

Riešením rovnice je vektor WT, z ktorého určíme T-invarianty.

Metóda výpočtu invariantov CPN je omnoho náročnejšia ako tá pre P/T siete, keďže namiesto čísel manipuluje s funkciami. O náročnosti svedčí aj fakt, že napriek prísľubu danému v [26] dodnes neexistuje nástroj ju implementujúci. Alternatívne sa využíva prevod (tzv. unfolding) na ekvivalentnú P/T sieť a výpočet invariantov pre ňu.

2.3 Hodnotiace Petriho siete

Hodnotiace Petriho siete (Evaluative Petri Nets, EvPN) sú rozšírením P/T sietí, ktoré bolo predstavené v [17]. Ide však o odlišný typ rozšírenia ako sú CPN. Kým CPN si zachovali vyjadrovaciu silu P/T sietí a zvýšili modelovaciu silu, EvPN majú vyjadrovaciu silu väčšiu, konkrétne rovnú Turingovým strojom (TS), no ich modelovacia sila ostáva na úrovni P/T sietí, keďže používa navzájom neodlíšiteľné značky. Napriek vyššej vyjadrovacej sile disponujú EvPN analytickým aparátom, umožňujúcim odvodenie invariantných vlastností siete. Tento aparát je bližšie opísaný v [17], tu uvádzame iba základné definície[6] štruktúry a správania sa EvPN. Okrem EvPN existujú aj iné typy PS s vyjadrovacou silou TS, snáď najznámešie sú PS s inhibítormi [11, 49].

Definícia 2.3.1. Hodnotiaca Petriho sieť *(EvPN) je 5-tica*

[6] Definície uvedené v tejto časti boli oproti [17] prepracované.

$H = (P_H, T_H, pre_H, post_H, m_0)$, *kde*

$P_H = P \cup P_F \cup P_E$ *je konečná množina miest:*
$P = \{p_1, \ldots, p_k\}$, $k = |P|$,
$P_F = \{fn_1, \ldots, fn_v\}$, $v = |P_F|$,
$\forall ii (1 \leq ii \leq v) : fn_{ii} = f_{ii}(p_1, \ldots, p_k), f_{ii} \in F$,
$P_E = \{pr_1, \ldots, pr_w\}$, $w = |P_E|$,
$\forall jj (1 \leq jj \leq w) : pr_{jj} = e_{jj}(p_1, \ldots, p_k), e_{jj} \in E$;

$T_H = \{t_1, \ldots, t_{|T_H|}\}$ *je konečná množina prechodov;*

$pre : P_H \times \Omega \times T_H \rightarrow \mathbb{N} \times \mathbb{C}$
$post : P_H \times \Omega \times T_H \rightarrow \mathbb{N} \times \mathbb{C}$
sú funkcie definujúce prepojenie miest a prechodov;

m_0 *je počiatočné značenie.*

F je množina funkcií s oborom hodnôt $\mathbb{N}-\{0\}$, E je množina predikátov, $\Omega = P \cup P_F \cup \{1\}$, $\mathbb{C} = \mathbb{K} \cup \{\omega, d\}$, $\mathbb{K} \subseteq \mathbb{N}$. Symboly ω a d nepatria do $P_H \cup T_H$.

□

Symbol ω aj tu predstavuje číslo väčšie ako ktorékoľvek prirodzené číslo a d umožňuje značeniu ísť do záporných čísel: ak $pre(X, Y, t) = (n, d)$ alebo $post(X, Y, t) = (n, d)$ tak $m(X)$ môže nadobúdať záporné hodnoty.

Trieda hodnotiacich Petriho sietí v sebe zahŕňa triedu P/T sietí a ich niektorých modifikácií, napríklad sietí s obmedzenou kapacitou miest. Tieto siete sú také EvPN, ktoré majú $P_H = P$, $Y = \{1\}$ a kde platí

$$\forall p, t, \nu \, (p \in P \wedge t \in T_H \wedge \nu \in \mathbb{N}) : pre(p, 1, t) \neq (\nu, d) \wedge post(p, 1, t) \neq (\nu, d).$$

Stav EvPN je vyjadrený jej značením:

Definícia 2.3.2. Značenie *EvPN H je funkcia $m : P_H \rightarrow \mathbb{Z}$.*

Ak $pl \in P$, potom $m(pl)$ je počet tokenov v mieste pl.

Ak $pl \in P_F$, $pl = fn = f(p_1, \ldots, p_k)$, potom
$m(pl) = f(m(p_1), \ldots, m(p_k))$.

Ak $pl \in P_E$, $pl = pr = e(p_1, \ldots, p_k)$, *potom*
$m(pl) = 1 \Leftrightarrow e(m(p_1), \ldots, m(p_k)) = true$,
$m(pl) = 0 \Leftrightarrow e(m(p_1), \ldots, m(p_k)) = false$.

Vektorová forma m *je vektor* $\vec{m} \in \mathbb{Z}^{|P_H|}$,

$$\begin{aligned} \vec{m} &= (m(pl_1), m(pl_2), \ldots, m(pl_{|P_H|})) \\ &= (m(p_1), \ldots, m(p_k), m(fn_1), \ldots, m(fn_v), m(pr_1), \ldots, m(pr_w)) \end{aligned}$$

□

Značenie m je možné rozdeliť na dve časti. Prvá bude pozostávať zo značení miest z P a druhá zo značení miest z $P_F \cup P_E$. Prvá časť, ktorú nazveme *nezávislé značenie*, jednoznačne určuje celé značenie EvPN; druhá časť (*závislé značenie*) je iba funkciou prvej, definovanou *miestami-funkciami* z P_F a *miestami-predikátmi* z P_E. Miesta z P nazývame *individuálne premenné*.

Definícia 2.3.3. *Nech* H *je EvPN a* m *je jej značenie. Potom* nezávislé značenie H *je vektor* $\iota(\vec{m}) = (m(p_1), \ldots, m(p_k))$, $p_i \in P$. Nezávislé počiatočné značenie H *je vektor* $\iota(\vec{m_0}) = (m_0(p_1), \ldots, m_0(p_k))$, $p_i \in P$.

Príklad 2.3.1 (EvPN Divider)**.** Táto EvPN je prevzatá z [17] a simuluje celočíselné delenie dvoch čísel, $x1$ a $x2$. Po postupnom odpálení všetkých vykonateľných prechodov sa *EvPN Divider* dostane do mŕtveho (deadlockového) značenia m_f, kde:

$$\begin{aligned} m_f(p_8) &= m_0(p_6)\ div\ m_0(p_7) = x1\ div\ x2 \\ m_f(p_9) &= m_0(p_6)\ mod\ m_0(p_7) = x1\ mod\ x2 \end{aligned}$$

Definícia *EvPN Divider* je v tab. 2.2 a jej graf je na obr. 2.9.

□

2.3.1 Výpočet EvPN

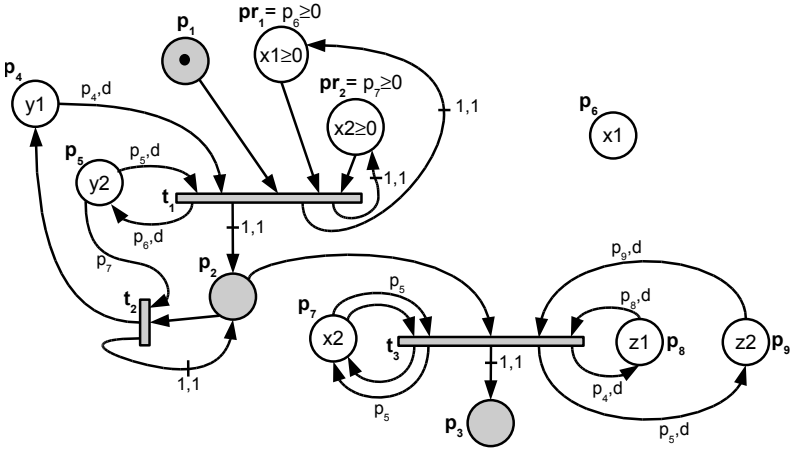
V definíciách týkajúcich sa výpočtu EvPN budú používané množiny ${}^\bullet t$, $t^\bullet$, funkcie $Pr_\Omega(X,t)$, $Pr^{\vec{d}}_\Omega(X,t)$, $Ps_\Omega(X,t)$, $Ps^k_\Omega(X,t)$, kde $X \in P_H$, $t \in T_H$, a funkcia $M(Y)$, kde $Y \in \Omega$. Tieto sú definované nasledovne:

$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$, $P_F = \varnothing$, $P_E = \{pr_1, pr_2\}$
$pr_1 = p_6 \geq 0$, $pr_2 = p_7 \geq 0$
$T_H = \{t_1, t_2, t_3\}$
$m_0 = (1, 0, 0, y1, y2, x1, x2, z1, z2, x1 \geq 0, x2 \geq 0)$

p	**o**	**t**	**pre(p,o,t)**
p_1	1	t_1	$(1, \omega)$
p_2	1	t_2	$(1, \omega)$
p_2	1	t_3	$(1, \omega)$
p_4	p_4	t_1	$(1, d)$
p_5	p_5	t_1	$(1, d)$
p_5	p_7	t_2	$(1, \omega)$
p_7	p_5	t_3	$(1, \omega)$
p_7	1	t_3	$(1, \omega)$
p_8	p_8	t_3	$(1, d)$
p_9	p_9	t_3	$(1, d)$
pr_1	1	t_1	$(1, \omega)$
pr_2	1	t_1	$(1, \omega)$
inak			$(0, 0)$

p	**o**	**t**	**post(p,o,t)**
p_2	1	t_1	$(1, 1)$
p_3	1	t_3	$(1, 1)$
p_4	1	t_2	$(1, \omega)$
p_5	p_6	t_1	$(1, \omega)$
p_7	p_5	t_3	$(1, \omega)$
p_7	1	t_3	$(1, \omega)$
p_8	p_4	t_3	$(1, d)$
p_9	p_5	t_3	$(1, d)$
pr_1	1	t_1	$(1, 1)$
pr_2	1	t_1	$(1, 1)$
p_2	1	t_2	$(1, 1)$
inak			$(0, 0)$

Tabuľka 2.2: Definícia *EvPN* Divider.



Obr. 2.9: Graf EvPN Divider

$$
\begin{aligned}
{}^\bullet t &= \{X \in P_H |\ \exists Y \in \Omega\ :\ pre(X,Y,t) \neq (0,0)\} \\
t^\bullet &= \{X' \in P_H |\ \exists Y' \in \Omega\ :\ post(X',Y',t) \neq (0,0)\} \\
Pr_\Omega(X,t) &= \{Y | Y \in \Omega \wedge pre(X,Y,t) = (\nu_Y, c_Y) \wedge \nu_Y > 0\} \\
Pr^{\bar{d}}_\Omega(X,t) &= \{Y | Y \in \Omega \wedge pre(X,Y,t) = (\nu_Y, c_Y) \wedge \nu_Y > 0 \wedge\ c_Y \neq d\} \\
Ps_\Omega(X,t) &= \{Y' | Y' \in \Omega \wedge post(X,Y',t) = (\nu_{Y'}, c_{Y'}) \wedge \nu_{Y'} > 0\} \\
Ps^k_\Omega(X,t) &= \{Y' | Y' \in \Omega \wedge post(X,Y',t) = (\nu_{Y'}, c_{Y'}) \wedge \nu_{Y'} > 0 \wedge \\
&\quad c_{Y'} \in \mathbb{K}\} \\
M(Y) &= \begin{cases} 1 & if\ Y = 1 \\ m(Y) & if\ Y \in P \cup P_F \end{cases}
\end{aligned}
$$

Definícia 2.3.4. *Prechod $t \in T_H$* je vykonateľný v značení m *(čo označujeme $m \overset{t}{\vdash}$) hodnotiacej Petriho siete H práve vtedy, ak platia nasledujúce podmienky:*

$$
\forall X \in {}^\bullet t : Pr^{\bar{d}}_\Omega(X,t) \neq \varnothing \Rightarrow M(X) \geq \sum_{\forall Y \in Pr^{\bar{d}}_\Omega(X,t)} \nu_Y . M(Y) \qquad pre(X,Y,t) = (\nu_Y, c_Y) \tag{2.35}
$$

$$
\forall X', Y' (X' \in\ t^\bullet \wedge Y' \in Ps^k_\Omega(X',t) \wedge post(X',Y',t) = (\nu_{Y'}, c_{Y'})) : \\
M(X') + \nu_{Y'} . M(Y') \leq c_{Y'} \tag{2.36}
$$

Ak t je vykonateľný v m, môže byť vykonaný (odpálený). Vykonanie t v m vedie k novému značeniu m' (čo označujeme $m \overset{t}{\vdash} m'$):

$$\begin{array}{l} \forall X \in P_H : m'(X) = \; m(X) \quad - \quad \sum_{\forall Y \in Pr_\Omega(X,t)} \nu_Y . m(Y) \\ \qquad\qquad\qquad\qquad\quad + \quad \sum_{\forall Y' \in Ps_\Omega(X,t)} \nu_{Y'} . m(Y') \\ pre(X,Y,t) = (\nu_Y, c_Y), \\ post(X,Y',t) = (\nu_{Y'}, c_{Y'}). \end{array} \tag{2.37}$$

□

Definícia 2.3.5. *Sekvencia $\sigma \in T_H^*$, $\sigma = t_{i_1}, t_{i_2}, \ldots, t_{i_r}$ je* prípustná sekvencia prechodov *(psp)* EvPN H v m_0, *práve vtedy, ak existuje sekvencia značení $m_0, m_1, \ldots, m_r$ taká, že $\forall j (1 \leq j \leq r) : m_{j-1} \overset{t_{i_j}}{\vdash} m_j$.*

□

Kapitola 3

B-metóda

B-metóda (*B*, *B-method*) a jej špecifikačný a návrhový jazyk *B-jazyk* (*B-language*) boli vytvorené v rokoch 1985 až 1988 J.R. Abrialom, výskumnou skupinou „Programming Research Group“ na Oxfordskej univerzite a výskumným oddelením firmy British Petroleum International, ktorá projekt iniciovala a sponzorovala.

B-jazyk vychádza z formálneho špecifikačného jazyka Z (Z notation) [55], na ktorého vývoji sa J.R. Abrial taktiež podieľal. B-metóda podporuje nielen formálnu špecifikáciu systému, ale aj celý vývojový proces od formálnej špecifikácie cez sériu zjemnení až po spustiteľnú implementáciu vo zvolenom programovacom jazyku. Pomocou tzv. *povinných dôkazov* (*POb*, *Proof Obligations*) je možné overiť vnútornú konzistenciu špecifikácie a taktiež verifikovať zjemnenie voči abstraktnejšej špecifikácii. Za účelom validácie formálnej špecifikácie voči požiadavkám zákazníka (ktoré sú opísané abstraktne, ale neformálne) je možné použiť animačné (simulačné) techniky.

Pre návrh a vývoj v B boli vyvinuté dva robustné komerčne dostupné nástroje, *B-Toolkit* a *Atelier B*. B-Toolkit bol vyvíjaný firmou B-Core, tá však už dnes na trhu nepôsobí. Atelier B [61] bol vytvorený francúzskou firmou Digilog v spolupráci s J.R. Abrialom. Po Digilog-u vývoj a podporu nástroja Atelier B prebrala firma ClearSy [69]. ClearSy s Atelier B je v súčasnosti (2014) zrejme jediným väčším hráčom na poli komerčného využitia B-metódy a podieľa sa aj na jej ďalšom vývoji. B-metóda je v poslednej dobe využívaná najmä na vývoj riadiaceho softvéru pre automatizované linky metra v spolupráci so spoločnosťami

Siemens (Mobility) a Alstom. Nástroj Atelier B je dostupný zdarma z [61].

Stručný opis B-metódy uvedený v tejto kapitole vychádza najmä z publikácie [43], ktorá sa hlbšie venuje problematike jej použitia pri návrhu softvéru, a je v nej uvedených množstvo príkladov, a z publikácie [1], ktorú možno považovať za základnú literatúru o B-metóde a jej jazyku. Dobrými zdrojmi informácií o B sú aj knižná publikácia [52] a web stránky nástroja Atelier B [61].

V posledných rokoch sa vyvíja modifikovaná verzia B-metódy, nazvaná Event-B [2], s pozmenenou filozofiou, vývojovým procesom a špecifikačným jazykom a aplikačnou doménou rozšírenou všeobecne na návrh diskrétnych systémov, a nie primárne softvéru.

3.1 Vývojový proces v B

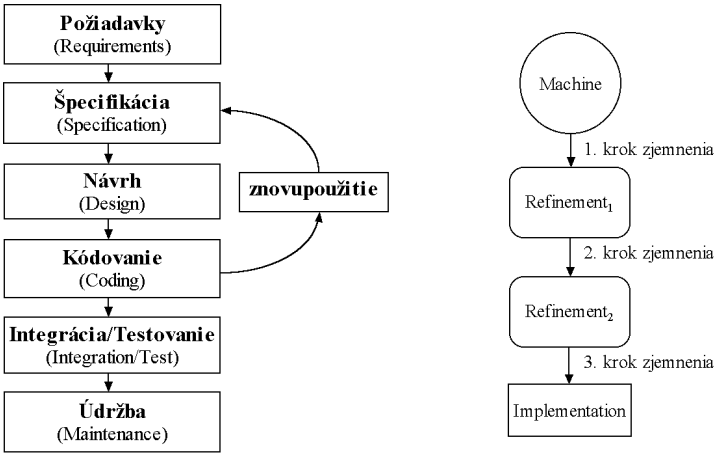
Proces, resp. životný cyklus, vývoja softvérového systému v B-metóde (obr. 3.1 a)) je podobný vývoju bežného softvéru. Líši sa však dôrazom na formálny matematický dôkaz podstatných vlastností špecifikácie a ich zachovania pri prechode k implementácii. Ďalším výrazným rozdielom je, že prítomnosť abstraktnej formálnej špecifikácie systému umožňuje chyby návrhu odstrániť v počiatočných fázach vývoja: Táto špecifikácia, s jednoznačne definovanou sémantikou, je k dispozícii v rannom štádiu vývoja, kedy sa pri klasickom vývoji systému zvyčajne používajú iba semiformálne diagramatické modely (napr. UML). Proces vývoja pozostáva z nasledujúcich fáz [43]:

1. **Analýza požiadaviek.**
   Vytvorenie neformálneho alebo štruktúrovaného modelu problémovej domény a požiadaviek na vytváraný systém. Výsledkom je *množina analytických modelov* (napríklad UML diagramy).

2. **Vývoj špecifikácie.**

   (a) Formalizácia prvkov analytických modelov do podoby *abstraktných strojov*. Analytické modely sa pritom používajú aj na dekompozíciu špecifikácie na koncepčne zmysluplné komponenty. Abstraktné stroje sú špecifikačné komponenty podobné triedam v objektovom programovaní.

   (b) Animácia na overenie (validáciu) špecifikácie voči vybraným požiadavkám na systém a testovacím scenárom.



Obr. 3.1: Vývoj v B: životný cyklus vývoja v B (a) a príklad sekvencie špecifikačných komponentov (b)

(c) Generovanie *povinných dôkazov* (POb) a ich dôkaz. Pojmom povinné dôkazy označujeme tvrdenia (predikáty), ktorých dokázaním potvrdíme platnosť požadovaných invariantných vlastností špecifikácie (tzn. abstraktných strojov). Niekedy pod povinnými dôkazmi myslíme aj samotný proces dokazovania ich platnosti.

Výsledkom je formálna *abstraktná špecifikácia systému*. Tá v B pozostáva z kolekcie *abstraktných strojov* (MACHINE), navzájom poprepájaných (sprístupnených) kompozičnými mechanizmami, ktoré B-metóda poskytuje. Tejto fáze sa venujú časti 3.3 a 3.6.

3. **Návrh.**

(a) Identifikácia dekompozície implementácie systému, zahŕňajúcej znovu použiteľné komponenty z predošlého vývoja alebo knižníc.

(b) Zjemnenie vybraných komponentov formálnej špecifikácie.

Postupným zjemňovaním komponentu vzniká sekvencia špecifikačných komponentov typu *zjemnenie* (REFINEMENT), jej konečným výsledkom je špecifikačný komponent *implementácia* (IMPLEMENTATION). Proces zjemňovania ilustruje obr. 3.1 b), kde kruh je abstraktný stroj, zaoblené obdĺžniky sú zjemnenia a obdĺžniky implementácie.

(c) Generovanie povinných dôkazov vytvorených zjemnení a ich dôkaz. Povinné dôkazy zjemnenia overujú, či zjemnenie konkretizuje (zjemňuje) príslušný abstraktnejší komponent tak, ako je to dané v invariante zjemnenia.

Produktom tejto fázy je konkrétny *formálny návrh*, pozostávajúci z jedného alebo viacerých špecifikačných komponentov IMPLEMENTATION. Zaoberajú sa ňou časti 3.4 a 3.5.

4. **Kódovanie, integrácia, testovanie.**

   (a) Použitie generátora kódu na návrhy najnižšej úrovne, na implementácie.

   (b) Testovanie vygenerovaného kódu s použitím modelových prípadov, vytvorených podľa požiadaviek na systém.

   Táto fáza produkuje *vykonateľnú implementáciu* v konkrétnom programovacom jazyku (napr. ADA, C či Java).

Z uvedených krokov sú samotnou B-metódou, a nástrojmi pre ňu vytvorenými, podporované kroky 2 až 4a.

3.2 B-jazyk

B-jazyk, jazyk v ktorom sa zapisujú špecifikačné komponenty od MACHINE po IMPLEMENTATION, je možné rozdeliť na dve časti:

Jazyk pre zápis výrazov a predikátov. Umožňuje zapísať výrazy a predikáty definujúce parametre, konštanty, množiny, premenné a invariantné vlastnosti špecifikačného komponentu. Predikáty a výrazy sú použité aj v rámci konštruktov GSL. Jazyk je vybudovaný na základoch klasickej logiky a teórie množín (Zermelo-Fraenkel).

Jazyk zovšeobecnenej substitúcie (GSL). Obsahuje príkazy, pomocou ktorých sú definované operácie špecifikačného komponentu. Vychádza z jazyka strážených príkazov, definovaného E.W Dijkstrom [12].

V častiach 3.2.1 až 3.2.5 je uvedený popis predikátov a väčšiny[1] výrazov, pričom sú rozdelené podľa toho s akými typmi matematických objektov manipulujú. Sú uvedené v tabuľkách, rozdelených na 3 stĺpce. V prvom („Výraz“, resp. „Predikát“) je uvedený matematický zápis výrazu, resp. predikátu, v druhom stĺpci („ASCII“) je jeho zápis používaný pri písaní špecifikácie (zdrojového kódu) daného komponentu[2] a v treťom („Význam“) jeho význam. Pod symbolom

$$\mathcal{P}_0(x)$$

budeme v tabuľkách 3.1 až 3.8 rozumieť obmedzujúci predikát v tvare

$$x \in S,\ x = E,\ x \subset S \text{ alebo } x \subseteq S,$$

kde $z\backslash S$ a $z\backslash E$. Zápis

$$z\backslash S, \text{ resp. } z\backslash E$$

znamená, že vo výraze S, resp. E nie je žiaden voľný výskyt premenných zo zoznamu z, ktorý obsahuje x.

Syntaxi a sémantike jazyka GSL je venovaná časť 3.2.6. Kompletný popis B-jazyka je možné nájsť v referenčnom manuáli [8] a ďalšej dokumentácii pre Atelier B.

3.2.1 Všeobecné a množinové predikáty a výrazy

Všeobecné predikáty a (jediný) všeobecný výraz sú uvedené v tabuľkách 3.1 a 3.2. V tabuľkách 3.3 a 3.4 sú predikáty a vybrané výrazy pre množiny. Symboly P, Q sú predikáty, E, F výrazy, S, T, R množiny, z je zoznam premenných a x je premenná.

Namiesto slova množina(y) budeme niekedy v týchto a ďalších tabuľkách používať skratku „mn.“.

[1] Sú vynechané niektoré redundantné výrazy pre množiny, relácie a sekvencie, ktoré môžu byť skonštruované pomocou iných, tu uvedených, výrazov. Tiež sú vynechané výrazy a predikáty pre záznamy a stromy.

[2] Tak ako pri štandardnom programovaní sú špecifikačné komponenty písané ako textové súbory, tzn. nie je v nich možné priamo zapisovať matematické symboly.

Predikát	**ASCII**	**Význam**
$P \wedge Q$	`P & Q`	Konjunkcia: „P a Q“.
$P \vee Q$	`P or Q`	Disjunkcia: „P alebo Q“.
$P \Rightarrow Q$	`P => Q`	Implikácia: „ak P potom Q“.
$P \Leftrightarrow Q$	`P <=> Q`	Ekvivalencia: „P práve vtedy ak Q“.
$\neg P$	`not(P)`	Negácia: „nie P“.
true	`true`	Vždy pravdivý predikát.
$\forall z.(Q \Rightarrow P)$	`!(z).(Q => P)`	Všeobecná kvantifikácia: „pre každé z, ak platí Q, potom platí P“. Predikát Q musí pre každú premennú x zo zoznamu z obsahovať $\mathcal{P}_0(x)$.
$\exists z.P$	`#(z).P`	Existenčná kvantifikácia: „pre niektoré z platí P“. P musí pre každú x zo z obsahovať $\mathcal{P}_0(x)$.
$E = F$	`E = F`	Rovnosť: „E je rovný F“.
$E \neq F$	`E /= F`	Nerovnosť: „E je rôzny od F“.

Tabuľka 3.1: Všeobecné predikáty

Výraz	**ASCII**	**Význam**
$E \mapsto F$	`E |-> F`	Usporiadaná dvojica (E, F).

Tabuľka 3.2: Všeobecný výraz

Predikát	**ASCII**	**Význam**
$E \in S$	`E: S`	Príslušnosť k množine : „E patrí do S“.
$E \notin S$	`E /: S`	„E nepatrí do S.“
$S \subseteq T$	`S <: T`	„S je podmnožina T.“ $S \subseteq T \equiv \forall x.(x \in S \Rightarrow x \in T)$
$S \not\subseteq T$	`S /<: T`	„S nie je podmnožina T.“ $S \not\subseteq T \equiv \neg(S \subseteq T)$
$S \subset T$	`S <<: T`	„S je vlastná podmnožina T.“ $S \subset T \equiv (S \subseteq T) \wedge \neg(T \subseteq S)$
$(S \not\subset T)$	`S /<<: T`	„S nie je vlastná podmnožina T.“ $S \not\subset T \equiv \neg(S \subset T)$

Tabuľka 3.3: Množinové predikáty

Výraz	**ASCII**	**Význam**
$\varnothing$	`{}`	prázdna množina (mn.)
$\{z\|P\}$	`{z|P}`	Definovanie mn. pomocou vlastností jej prvkov (set comprehension). Predikát P musí pre každú x zo z obsahovať $\mathcal{P}_0(x)$.
$\{z\|z \in R \wedge P\}$	`{z|z:R&P}`	Iný zápis set comprehension.
$\{E\}$	`{E}`	Mn. s jediným prvkom, definovaným E.
$\{E, \ldots, F\}$	`{E,...,F}`	Mn. s prvkami, definovanými E až F.
$S \times T$	`S*T`	Kartézsky súčin množín S a T. $S \times T = \{(x, y)\|\ x \in S \wedge y \in T\}$
$\mathbb{P}(S)$	`POW(S)`	Mn. všetkých podmnožín (power set) S. $(x \in \mathbb{P}(S)) \Leftrightarrow (x \subseteq S)$
$\mathbb{P}_1(S)$	`POW1(S)`	Mn. všetkých neprázdnych podmnožín S. $\mathbb{P}_1(S) = \mathbb{P}(S) - \varnothing$
$\mathbb{F}(S)$	`FIN(S)`	Mn. všetkých konečných podmnožín S.
$\mathbb{F}_1(S)$	`FIN1(S)`	Mn. všetkých neprázdnych konečných podmnožín S.
$S \cup T$	`S \/ T`	Zjednotenie množín S a T. $S \cup T = \{x\|\ x \in S \vee x \in T\}$
$S \cap T$	`S /\ T`	Prienik množín S a T. $S \cap T = \{x\|\ x \in S \wedge x \in T\}$
$S - T$	`S - T`	Rozdiel množín S a T. $S - T = \{x\|\ x \in S \wedge x \notin T\}$

Tabuľka 3.4: Niektoré množinové výrazy

3.2.2 Predikáty a výrazy pre prirodzené a celé čísla

Prirodzené a celé čísla sú tiež výrazmi. Predikáty a výrazy pre ne sú uvedené v tabuľkách 3.5 a 3.6. Sú tu použité rovnaké symboly ako v časti 3.2.1 a tiež symboly n a m zastupujúce čísla, resp. číselné výrazy.

Predikát	**ASCII**	**Význam**
$m > n$	`m > n`	„ m je väčšie ako n.“
$m < n$	`m < n`	„ m je menšie ako n.“
$m \geq n$	`m >= n`	„ m je väčšie ako alebo rovné n.“
$m \leq n$	`m <= n`	„ m je menšie ako alebo rovné n.“

Tabuľka 3.5: Predikáty pre prirodzené a celé čísla

Výraz	**ASCII**	**Význam**
$\mathbb{Z}$	`INT`, `INTEGER`	Množina celých čísel.
$\mathbb{N}$	`NAT`, `NATURAL`	Množina prirodzených čísel.
$\mathbb{N}_1$	`NAT1`, `NATURAL1`	Množina nenulových prirodzených čísel.
$min(S)$	`min(S)`	Minimum z množiny S, $S \in \mathbb{P}_1(\mathbb{Z})$.
$max(S)$	`max(S)`	Maximum z množiny S, $S \in \mathbb{F}_1(\mathbb{Z})$.
$m + n$	`m + n`	Súčet m a n.
$m - n$	`m - n`	Rozdiel m a n.
$m \times n$	`m * n`	Súčin m a n.
m/n	`m / n`	Celočíselné delenie m číslom n.
$m \; mod \; n$	`m mod n`	Zvyšok po celočíselnom delení m/n.
$m..n$	`m..n`	Množina prirodzených čísel od m po n.
$\|S\|$	`card(S)`	Počet prvkov konečnej množiny S.
$\Sigma z.(P\|F)$	`SIGMA(z).(P|F)`	Súčet hodnôt výrazu F nad prirodzenými číslami pre z také, že P platí. P musí pre každú x zo z obsahovať $\mathcal{P}_0(x)$.
$\Pi z.(P\|F)$	`PI(z).(P|F)`	Ako $\Sigma z.(P\|F)$, ale súčin.

Tabuľka 3.6: Výrazy pre prirodzené a celé čísla

Pre prirodzené čísla je výraz $m - n$ definovaný iba pre $m \geq n$. V tabuľke 3.6 nájdeme dvojaké označenie pre množiny $\mathbb{Z}$, $\mathbb{N}$ a $\mathbb{N}_1$. To dlhšie (napr. `INTEGER`) označuje abstraktnú, nekonečnú množinu a kratšie (napr. `INT`) implementovateľnú, tzn. zhora aj zdola ohraničenú.

3.2.3 Výrazy pre relácie

Reláciou je každá podmnožina nejakého kartézskeho súčinu. Relácia je teda množina usporiadaných dvojíc, a je tiež výrazom. Niektoré výrazy pre relácie sú uvedené v tabuľke 3.7, kde S, T, U, V sú množiny, r, p, q sú relácie, $s \subseteq S$ a $t \subseteq T$.

Výraz	**ASCII**	**Význam**
$S \leftrightarrow T$	`S <-> T`	Množina relácií z S do T. $S \leftrightarrow T = \mathbb{P}(S \times T)$
$dom(r)$	`dom(r)`	Definičný obor relácie r, $r \in S \leftrightarrow T$. $dom(r) = \{x \mid x \in S \wedge \exists y.(y \in T \wedge (x,y) \in r)\}$
$ran(r)$	`ran(r)`	Obor hodnôt relácie r, $r \in S \leftrightarrow T$. $ran(r) = \{y \mid y \in T \wedge \exists x.(x \in S \wedge (x,y) \in r)\}$
$p;q$	`p;q`	Kompozícia relácií $p \in S \leftrightarrow T$ a $q \in T \leftrightarrow U$. $p;q = \{(x,z) \mid (x,z) \in S \times U \wedge$ $\exists y.(y \in T \wedge (x,y) \in p \wedge (y,z) \in q)\}$
$p \circ q$	`p circ q`	$p \circ q = q;p$
$id(S)$	`id(S)`	$id(S) = \{(x,y) \mid (x,y) \in S \times S \wedge x = y\}$
$s \lhd r$	`s <| r`	Zúženie r, $r \in S \leftrightarrow T$, podľa s. $s \lhd r = \{(x,y) \mid (x,y) \in r \wedge x \in s\}$
$r \rhd t$	`r |> t`	$r \rhd t = \{(x,y) \mid (x,y) \in r \wedge y \in t\}$
r^{-1}	`r~`	Inverzná relácia k r, $r \in S \leftrightarrow T$. $r^{-1} = \{(y,x) \mid (y,x) \in T \times S \wedge (x,y) \in r\}$
$p \otimes q$	`p >< q`	Priamy súčin $p \in S \leftrightarrow U$ a $q \in S \leftrightarrow V$. $p \otimes q = \{(x,(y,z)) \mid (x,(y,z)) \in S \times (U \times V)$ $\wedge (x,y) \in p \wedge (x,z) \in q\}$
$p \parallel q$	`p || q`	Paralelný súčin $p \in S \leftrightarrow T$ a $q \in V \leftrightarrow U$. $p \parallel q = \{((x,y),(m,n)) \mid ((x,y),(m,n)) \in$ $(S \times V) \times (T \times U) \wedge (x,m) \in p \wedge (y,n) \in q\}$
r^n	`iterate(r,n)`	n-tá iterácia r, $n \in \mathbb{N}$, $r \in S \leftrightarrow S$. $r^0 = id(S),\ r^{n+1} = r;r^n$

Tabuľka 3.7: Niektoré výrazy pre relácie

3.2.4 Výrazy pre funkcie

Funkcie, alebo zobrazenia, sú relácie pre ktoré platí, že nemôžu obsahovať dve rôzne dvojice s rovnakým prvým prvkom. Funkcie sú takisto výrazmi. Výrazy pre funkcie sú uvedené v tabuľke 3.8, kde P je predikát, E výraz, S, T sú množiny a z je zoznam premenných.

Výraz	**ASCII**	**Význam**
$S \nrightarrow T$	`S +-> T`	Množina parciálnych funkcií z S do T. $S \nrightarrow T = \{r \mid r \in S \leftrightarrow T \wedge (r^{-1}; r) \subseteq id(T)\}$
$S \rightarrow T$	`S --> T`	Množina úplných funkcií z S do T. $S \rightarrow T = \{f \mid f \in S \nrightarrow T \wedge dom(f) = S\}$
$S \rightarrowtail T$	`S >+> T`	Množina parciálnych injektívnych funkcií z S do T. $S \rightarrowtail T = \{f \mid f \in S \nrightarrow T \wedge f^{-1} \in T \nrightarrow S\}$
$S \rightarrowtail T$	`S >-> T`	Množina úplných injektívnych funkcií z S do T. $S \rightarrowtail T = (S \rightarrowtail T) \cap (S \rightarrow T)$
$S \twoheadrightarrow T$	`S +->> T`	Množina parciálnych surjektívnych funkcií z S do T. $S \twoheadrightarrow T = \{f \mid f \in S \nrightarrow T \wedge ran(f) = T\}$
$S \twoheadrightarrow T$	`S -->> T`	Množina úplných surjektívnych funkcií z S do T. $S \twoheadrightarrow T = (S \twoheadrightarrow T) \cap (S \rightarrow T)$
$S \rightarrowtail\!\!\!\!\twoheadrightarrow T$	`S >->> T`	Množina bijektívnych funkcií z S do T. $S \rightarrowtail\!\!\!\!\twoheadrightarrow T = (S \twoheadrightarrow T) \cap (S \rightarrowtail T)$
$\lambda z.(z \in S \wedge P \mid E)$	`%(z).(z: S & P | E)`	Konštrukcia funkcie. Je to funkcia $\{(z, y) \mid z \in S \wedge y \in E \wedge P\}$, s doménou $\{z \mid z \in S \wedge P\}$, pričom $y \backslash E$ a $y \backslash P$.
$\lambda z.(P \mid E)$	`%(z).(P | E)`	Konštrukcia funkcie. Predikát P musí pre každú x zo z obsahovať $\mathcal{P}_0(x)$.
$f(x)$	`f(x)`	Hodnota funkcie f v x, $x \in dom(f)$.

Tabuľka 3.8: Výrazy pre funkcie

Tvar predikátu $\mathcal{P}_0(x)$ je definovaný v úvode časti 3.2.

3.2.5 Výrazy pre sekvencie

Sekvencia nad množinou S je parciálna funkcia z $\mathbb{N}_1$ do S, ktorej definičný obor je interval $1..n$, $n \in \mathbb{N}$. Sekvencie sú takisto výrazmi. Niektoré výrazy pre sekvencie sú uvedené v tabuľke 3.9, kde E, F sú výrazy, S je množina, s, t sú sekvencie prvkov z S a e je prvok z S.

Výraz	**ASCII**	**Význam**
$[\,]$	`<>`	Prázdna sekvencia.
$[E]$	`[E]`	Sekvencia s jediným prvkom, definovaným E. $[E] = \{1 \to E\}$
$[E, \ldots, F]$	`[E,...,F]`	Sekvencia s prvkami, definovanými E až F.
$size(s)$	`size(s)`	Dĺžka konečnej sekvencie s.
$seq(S)$	`seq(S)`	Množina konečných sekvencií prvkov z S.
$seq_1(S)$	`seq1(S)`	Množina konečných neprázdnych sekvencií prvkov z S.
$iseq(S)$	`iseq(S)`	Množina injektívnych sekvencií prvkov z S. $iseq(S) = seq(S) \cap (\mathbb{N}_1 \rightarrowtail S)$.
$perm(S)$	`perm(S)`	Množina permutácií S. $perm(S) = (1..card(S)) \twoheadrightarrow S$.
$s^\frown t$	`s^t`	Zreťazenie sekvencií s a t.
$e \to e$	`e -> s`	Sekvencia vytvorená pridaním e na začiatok s.
$s \leftarrow e$	`s <- e`	Sekvencia vytvorená pridaním e na koniec s.
$rev(s)$	`rev(s)`	Sekvencia s opačným poradím prvkov, ako s.
$s \uparrow n$	`s /|\ n`	Sekvencia vytvorená z s ponechaním iba jej prvých n prvkov, $n \leq size(s)$.
$s \downarrow n$	`s |\/ n`	Sekvencia vytvorená z s odstránením jej prvých n prvkov, $n \leq size(s)$.
$first(s)$	`first(s)`	Prvý prvok neprázdnej sekvencie s.
$last(s)$	`last(s)`	Posledný prvok neprázdnej sekvencie s.
$tail(s)$	`tail(s)`	Sekvencia vytvorená odstránením prvého prvku z neprázdnej sekvencie s.
$front(s)$	`front(s)`	Sekvencia vytvorená odstránením posledného prvku z neprázdnej sekvencie s.

Tabuľka 3.9: Niektoré výrazy pre sekvencie

3.2.6 Zovšeobecnená substitúcia

Operácie špecifikačných komponentov v B-metóde sú zapísané v *jazyku zovšeobecnenej substitúcie* (Generalized Substitution Language, GSL), ktorý je súčasťou B-jazyka. Konštrukty GSL sa nazývajú *zovšeobecnené substitúcie* (Generalized Substitution, GS). Jazyk GSL bol navrhnutý tak, aby

- zjednodušoval požiadavky na dôkazy,
- poskytol jednotný zápis od abstraktných strojov až k procedurálnemu kódu v implementácii a
- podporoval dekompozíciu operácií v zjemneniach abstraktného stroja.

Syntax zovšeobecnenej substitúcie

Prehľad konštruktov (zovšeobecnených substitúcií) jazyka GSL a ich intuitívny význam je uvedený v tabuľke 3.10. Symboly použité v nej a ďalších tabuľkách (po 3.13) majú nasledujúci význam:

- S, S_1, S_2, T, U sú zovšeobecnené substitúcie,
- I, E, P, Q sú predikáty,
- x, v sú premenné alebo zoznamy premenných,
- e, ve sú zoznamy výrazov nad premennými; e má rovnaký rozmer ako x a
- xx, vv sú premenné.

Rozdiel medzi $P|S$ a $P \Longrightarrow S$ je možné intuitívne pochopiť nasledovne: v $P|S$ by sa vykonávanie S nemalo vyvolať ak P nie je splnené a v $P \Longrightarrow S$ sa vykonávanie S nemôže vyvolať ak P nie je splnené. Ak platí P, tak je vykonanie $P|S$ aj $P \Longrightarrow S$ rovnaké: vykoná sa GS S. Ak P neplatí, tak vykonanie $P|S$ môže viesť do ľubovoľného stavu a nemusí ani terminovať[3]. Na druhej strane GS $P \Longrightarrow S$ je pri neplatnosti P nevykonateľná.

[3]V prípade neplatnosti P je $P|S$ definovaná rovnako ako príkaz *Abort* v jazyku strážených príkazov E.W. Dijkstru [12].

Syntax	**Názov a intuitívny význam**
$x := e$	jednoduché priradenie (základná substitúcia)
SKIP	prázdna GS: nerob nič
$S_1 [\!] S_2$	viazaná voľba : rob S_1 alebo S_2
$P \mid S$	pre-podmienka : ak P platí, správaj sa ako S
$P \Longrightarrow S$	strážený príkaz : iba ak P platí, vykonaj S
$@v.S$	neviazaný nedeterminizmus: pre nejaké v vykonaj S
$S_1; S_2$	sekvenčná kompozícia : vykonaj S_1, potom S_2
$S_1 \parallel S_2$	viacnásobná GS : vykonaj S_1 a S_2
WHILE E DO S INVARIANT I VARIANT ve END	cyklus: rob S kým platí E

Tabuľka 3.10: Zovšeobecnené substitúcie

Niektoré GS, alebo ich kombinácie, sa v operáciách špecifikačných komponentov zapisujú v pozmenenom tvare, ako to ukazuje tab. 3.11. Konštrukty ANY a VAR zavádzajú novú lokálnu premennú v, ktorá

GS	**Zápis GS v operáciách**
$S_1 [\!] S_2$	CHOICE S_1 OR S_2 END
$P \mid S$	PRE P THEN S END
$P \Longrightarrow S$	SELECT P THEN S END
$@v.S$	VAR v IN S END
$@v.(P \Longrightarrow S)$	ANY v WHERE P THEN S END
$(E \Longrightarrow S_1) [\!] (\neg E \Longrightarrow S_2)$	IF E THEN S_1 ELSE S_2 END
$@vv.(vv \in ss \Longrightarrow xx := vv)$	$xx :\in ss$

Tabuľka 3.11: Zápis niektorých GS v operáciách komponentov

v prípade ANY je obmedzená splnením predikátu P. Konštrukt ANY nazývame *zovšeobecnený strážený príkaz*. V každej fáze vývoja systému je možné používať iba podmnožinu týchto substitúcií:

- Vo fáze abstraktnej špecifikácie systému, tzn. v abstraktných strojoch, nie je povolené používať sekvenčnú kompozíciu (;), WHILE cykly a konštrukt VAR.

- V zjemneniach je povolené používať všetky substitúcie okrem konštruktov WHILE a VAR.

- V záverečnej, implementačnej, fáze (tzn. v implementáciách) je možné používať iba procedurálne konštrukty, analogické s tými v bežných programovacích jazykoch. Tu patria priradenie (v:=e), cyklus WHILE, príkaz vetvenia IF - THEN - ELSE a príkaz VAR.

Pre uľahčenie písania špecifikácií poskytuje jazyk GSL aj ďalšie konštrukty. Tie sú však len alternatívnym zápisom niektorých kombinácií tu uvedených zovšeobecnených substitúcií. Možno ich nájsť v [8].

Sémantika zovšeobecnenej substitúcie

Sémantika GS je definovaná v zmysle *predikátových transformérov* a *kalkulu najslabšej pre-podmienky* (Weakest precondition calculus) E.W. Dijkstru [12].

Najslabšiu pre-podmienku zapisujeme $[S]P$, čo znamená, že ak výpočet S (S je nejaká GS) začne v stave spĺňajúcom predikát $[S]P$, určite skončí (terminuje) a to v stave (post-stave) spĺňajúcom predikát P.

Definícia predikátu $[S]P$ pre základnú substitúciu, jednoduché priradenie $x := e$, kde $x = (x_1, \ldots, x_n)$ je n-tica (zoznam) premenných a $e = (e_1, \ldots, e_n)$ je n-tica (zoznam) výrazov, rovnakej dĺžky ako x, je

$$[x := e]P \equiv P[x := e] \tag{3.1}$$

Predikát $P[x := e]$ znamená textuálnu substitúciu každého z výrazov $e_1, \ldots, e_n$ za zodpovedajúci výraz $x_1, \ldots, x_n$. Ak sa v niektorom e_i nachádza premenná, ktorá sa po vykonaní substitúcie stane viazanou vo výslednom predikáte, potom kvantifikovaná premenná, ktorá by viazala túto premennú je premenovaná, aby sa vyhlo premenným z e_i.

Definícia $[S]P$ pre ostatné GS je vytvorená štrukturálnou indukciou zo základnej substitúcie (tab. 3.12). V prípade $[@v.S]P$ premenná v nie je voľná v P a v poslednom prípade γ je nová premenná, ktorá nie je voľná v substitúcii $WHILE$ alebo v zahrnutých predikátoch a l je zoznam premenných modifikovaných v cykle.

Význam definícií zdá sa byť jasný, podrobnejšie vysvetlenie si žiada snáď len sémantika konštruktu WHILE, predstavujúceho klasický prefixný cyklus, doplnený o nezáporný celočíselný výraz ve a invariant

$[x := e]P$	$\Leftrightarrow$	P[x:=e]
$[SKIP]P$	$\Leftrightarrow$	P
$[S_1 [\!] S_2]P$	$\Leftrightarrow$	$[S_1]P \wedge [S_2]P$
$[E \mid S]P$	$\Leftrightarrow$	$E \wedge [S]P$
$[E \Longrightarrow S]P$	$\Leftrightarrow$	$E \Rightarrow [S]P$
$[@v.S]P$	$\Leftrightarrow$	$\forall v : [S]P$
$[S_1; S_2]P$	$\Leftrightarrow$	$[S_1][S_2]P$
[IF E THEN S_1 ELSE S_2 END]P	$\Leftrightarrow$	$(E \Rightarrow [S_1]P) \wedge (\neg E \Rightarrow [S_2]P)$
[WHILE E DO S INVARIANT I VARIANT ve END]P	$\Leftarrow$	1. $I \wedge$ 2. $\forall l : (I \wedge E \Rightarrow [S]I) \wedge$ 3. $\forall l : (I \wedge \neg E \Rightarrow P) \wedge$ 4. $\forall l : (I \wedge E \Rightarrow ve \in \mathbb{N}) \wedge$ 5. $\forall l : (I \wedge E \wedge (ve = \gamma) \Rightarrow [S](ve < \gamma))$

Tabuľka 3.12: Sémantika GS

cyklu I. Zavedenie týchto dvoch zložiek ako aj samotnej sémantiky cyklu vychádza z [12]. Význam jednotlivých častí jeho predikátu $[S]P$ je nasledovný:

1. Invariant cyklu I platí pred začatím vykonávania cyklu. V praxi často máme, že premenné používané v cykle sú pred vstupom doň inicializované nejakou GS T a teda potrebujeme vypočítať najslabšiu pre-podmienku pre

   $$T; \text{WHILE } E \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } ve \text{ END.}$$

   Potom namiesto I počítame $[T]I$.

2. Invariant cyklu I je zachovávaný substitúciou cyklu (telom cyklu) S za predpokladu, že sa vstúpilo do cyklu. I je vždy pravdivý na začiatku každého vykonávania tela cyklu a aj pri ukončení cyklu.

3. Invariant spolu s negáciou podmienky cyklu ($\neg E$) implikuje post-podmienku.

4. Počas vykonávania tela cyklu je variant ve vždy prirodzené číslo. Obyčajne je výrazom zahŕňajúcim premenné modifikované v tele cyklu.

5. Variant *ve* je vždy striktne zmenšený pri každom vykonaní tela cyklu.

Z bodov 4 a 5 vyplýva, že cyklus bude terminovať po konečnom počte iterácií. Táto podmienka je v skutočnosti silnejšia ako najslabšia pre-podmienka pre WHILE (preto je v tabuľke implikácia a nie ekvivalencia).

Najslabšia pre-podmienka je, celkom prirodzene, kritériom pre porovnávanie zovšeobecnených substitúcií (definícia 3.2.1).

Definícia 3.2.1. *Nech S_1 a S_2 sú GS. Potom S_1 je rovná S_2 práve vtedy, ak pre každý predikát P platí, že $[S_1]P$ je ekvivalentné $[S_2]P$:*

$$(S_1 = S_2) \Leftrightarrow_{def} \forall P.([S_1]P \Leftrightarrow [S_2]P)$$

Na základe predchádzajúcej definície je možné sformulovať niekoľko *zákonov ekvivalencie pre GS*. Je ich možné nájsť v časti 6.1 publikácie [1] a niektoré z nich sú uvedené v tabuľke 3.13.

$P \Longrightarrow (Q \Longrightarrow S)$	$=$	$(P \wedge Q) \Longrightarrow S$
$P \Longrightarrow (S [\!] T)$	$=$	$(P \Longrightarrow S) [\!] (P \Longrightarrow T)$
$P \Longrightarrow (@v.S)$	$=$	$@v.(P \Longrightarrow S)$
$@v.S [\!] @v.T$	$=$	$@v.(S [\!] T)$

Tabuľka 3.13: Niektoré zákony ekvivalencie pre GS

V tejto časti sme sa nezaoberali sémantikou viacnásobnej GS. Tá je definovaná odlišne od ostatných a pred jej uvedením je potrebné zaviesť niekoľko predikátov charakterizujúcich GS.

Predikáty charakterizujúce GS

Pre každú GS S možno definovať niekoľko dôležitých predikátov, ktoré ju charakterizujú.

Prvým predikátom je $fis(S)$, ktorý sa nazýva *podmienka uskutočniteľnosti* (feasibility condition). Charakterizuje množinu stavov[4] z ktorých je vykonávanie S prípustné, ale nemusí nevyhnutne terminovať, pretože

[4]Hovoríme, že predikát P charakterizuje stav s, ak P v stave s platí.

post-stav v tomto prípade môže reprezentovať nedefinovaný stav, ktorý nespĺňa žiaden predikát. Predikát $fis(S)$ je definovaný takto:

$$fis(S) = \neg([S]false) \tag{3.2}$$

Druhý predikát je $trm(S)$, *podmienka ukončenia* (termination condition). Charakterizuje množinu stavov z ktorých ak začne vykonávanie (výpočet) S, tak určite terminuje. Je definovaný nasledovne:

$$trm(S) = [S]true \tag{3.3}$$

Pre bližšiu ilustráciu sú tvary predikátov fis a trm niektorých GS uvedené v tabuľke 3.14.

S	**fis(S)**	**trm(S)**
$x := e$	$true$	$true$
SKIP	$true$	$true$
$S_1 [] S_2$	$fis(S_1) \vee fis(S_2)$	$trm(S_1) \wedge trm(S_2)$
$P \mid S_1$	$P \Rightarrow fis(S_1)$	$P \wedge trm(S_1)$
$P \Longrightarrow S_1$	$P \wedge fis(S_1)$	$P \Rightarrow trm(S_1)$
$@v.S_1$	$\exists v : fis(S_1)$	$\forall v : trm(S_1)$
$@v.(P \Longrightarrow S_1)$	$\exists v : (P \wedge fis(S_1))$	$\forall v : (P \Rightarrow trm(S_1))$

Tabuľka 3.14: Predikáty fis a trm niektorých GS

Tretím predikátom, mierne odlišnej povahy ako predchádzajúce dva, je $prd_x(S)$, Nazýva sa *pred-po predikát* (before-after predicate) a definuje ako vykonanie S zmení stav toho špecifikačného komponentu, v ktorom sa vyskytuje. Je definovaný rovnicou (3.4).

$$prd_x(S) = \neg([S](x \neq x')) \tag{3.4}$$

Symbol x v (3.4) predstavuje zoznam stavových premenných špecifikačného komponentu, v ktorom sa S vyskytuje a x' zoznam premenných, iných ako stavových, rovnakej dĺžky ako x. Každá premenná z x' má rovnaký typ ako príslušná premenná z x a jej meno je zvyčajne menom príslušnej premennej z x, doplneným o apostrof. V $prd_x(S)$ vlastne x reprezentuje stav špecifikačného komponentu pred vykonaním S a x' stav po vykonaní S. Tvar predikátu prd pre niektoré GS uvádza tab. 3.15.

S	$\mathbf{prd_x(S)}$
$x := e$	$x' = e$
SKIP	$x' = x$
$S_1 [] S_2$	$prd_x(S_1) \vee prd_x(S_2)$
$P \mid S_1$	$P \Rightarrow prd_x(S_1)$
$P \Longrightarrow S_1$	$P \wedge prd_x(S_1)$
$@v.S_1$	$\exists v : prd_x(S_1)$, v nie je voľné v x'
$@v.(P \Longrightarrow S_1)$	$\exists v : (P \wedge prd_x(S_1))$, v nie je voľné v x'

Tabuľka 3.15: Predikát prd_x niektorých GS

Na prvý pohľad sa môže zdať, že dvojitá negácia, ktorú predikát prd obsahuje, je zbytočná a že $\neg[S](x \neq x') \Leftrightarrow [S](x = x')$. To však platí iba v prípade GS, ktoré sú deterministické, vždy vykonateľné a vždy terminujúce (tab. 3.16). Pre ostatné GS sú $\neg[S](x \neq x')$ a $[S](x = x')$ rozdielne, príklady takýchto GS sú uvedené v tabuľke 3.17.

S	$\neg[\mathbf{S}](\mathbf{x} \neq \mathbf{x}')$, $[\mathbf{S}](\mathbf{x} = \mathbf{x}')$
$x := 3$	$x' = 3$
$(x > 0 \Longrightarrow x := 5) []$	$(x > 0 \wedge x' = 5) \vee$
$(\neg(x > 0) \Longrightarrow x := 3)$	$(\neg(x > 0) \wedge x' = 3)$

Tabuľka 3.16: Príklady GS s rovnakými prd_x a $[S](x = x')$

S	$\neg[\mathbf{S}](\mathbf{x} \neq \mathbf{x}')$	$[\mathbf{S}](\mathbf{x} = \mathbf{x}')$	**Dôvod rozdielu**
$x := 3 [] x := 5$	$x' = 3 \vee x' = 5$	$x' = 3 \wedge x' = 5$	nedeterminizmus S
$x > 0 \mid x := 5$	$x > 0 \Rightarrow x' = 5$	$x > 0 \wedge x' = 5$	$trm(S) \neq true$
$x > 0 \Longrightarrow x := 5$	$x > 0 \wedge x' = 5$	$x > 0 \Rightarrow x' = 5$	$fis(S) \neq true$
$@v.(v > 0$	$\exists v.(v > 0$	$\forall v.(v > 0$	nedeterminizmus S
$\Longrightarrow x := v)$	$\wedge x' = v)$	$\Rightarrow x' = v)$	a $fis(S) \neq true$

Tabuľka 3.17: Príklady GS s rôznymi prd_x a $[S](x = x')$

Sémantika viacnásobnej GS

Sémantika viacnásobnej GS, $S_1 \| S_2$, je pomocou najslabšej pre-podmienky definovaná iba pre prípad

$$x_1 := e_1 \| x_2 := e_2 \tag{3.5}$$

keďže

$$x_1 := e_1 \| x_2 := e_2 \quad = \quad x_1, x_2 := e_1, e_2 \tag{3.6}$$

Najslabšiu pre-podmienku pre GS (3.5) teda vypočítame podľa vzťahu (3.1), kde $x = (x_1, x_2)$ a $e = (e_1, e_2)$. GS (3.5) nazývame *viacnásobná základná substitúcia*.

V ostatných prípadoch je potrebné upraviť danú GS na tvar (3.6) použitím vlastností viacnásobnej GS (tab. 3.18). Vlastnosť (d) platí len za predpokladu, že $trm(S) = true$ a v prípade (e) v nie je voľné v S. Obmedzenie použiteľnosti vlastnosti (e) možno jednoducho odstrániť premenovaním premenných zo zoznamu v.

(a)	$S \| \text{SKIP}$	$=$	S
(b)	$S \| (P \| T)$	$=$	$P \| (S \| T)$
(c)	$S \| (T [\!] U)$	$=$	$(S \| T) [\!] (S \| U)$
(d)	$S \| (P \Longrightarrow T)$	$=$	$P \Longrightarrow (S \| T)$
(e)	$S \| @v.T$	$=$	$@v.(S \| T)$

Tabuľka 3.18: Vlastnosti viacnásobnej GS

Viacnásobnú GS, $S_1 \| S_2$, je možné tiež úplne definovať použitím predikátov trm a prd [1]:

$$trm(S_1 \| S_2) \quad = \quad trm(S_1) \wedge trm(S_2) \tag{3.7}$$

$$prd_{x,y}(S_1 \| S_2) \quad = \quad prd_x(S_1) \wedge prd_y(S_2) \tag{3.8}$$

Symbol x v (3.8) predstavuje zoznam premenných špecifikačného komponentu, v ktorom sa vyskytuje GS S_1 a y zoznam premenných komponentu, v ktorom sa vyskytuje S_2.

Zo vzťahov (3.7) a (3.8) je možné odvodiť nasledujúci závažný výsledok o povahe viacnásobnej GS [1]:

Veta 3.2.1. *Nech S, T sú GS, pracujúce s rôznymi premennými x a y, a P, Q sú predikáty také, že x nie je voľné v P a y nie je voľné v Q (tzn. $x \backslash P$ a $y \backslash Q$). Potom*

$$[S]P \wedge [T]Q \Rightarrow [S \| T](P \wedge Q).$$

Normálna forma GS

Podľa [1] je možné každú GS, ktorá sa vyskytuje v nejakom abstraktnom stroji, zapísať v normálnej forme uvedenej vo vete 3.2.2.

Veta 3.2.2. *Nech S je zovšeobecnená substitúcia, x je zoznam stavových premenných abstraktného stroja, v ktorom sa S vyskytuje, x' je zoznam premenných, iných ako stavových, rovnakej dĺžky ako x a P, Q sú predikáty, x' nie je voľné v P. Potom*

$$S = P | @x'.(Q \Longrightarrow x := x')$$

pre nejaké P a Q, kde x' nie je voľné v P.

Dôkaz vety 3.2.2 pre GS „:=“, SKIP, PRE, CHOICE, SELECT a ANY je uvedený v [1].

Taktiež v [43] sa uvázda, že každá GS môže byť vyjadrená v normálnej forme v tvare:

```
PRE P
THEN
    ANY w
    WHERE Q
    THEN
        S1
    END
end
```

Táto normálna forma korešponduje s vetou 3.2.2 (jediným rozdielom je, že GS S_1 nie je bližšie špecifikovaná) a jej dodržiavanie je dôležité pre správne automatické generovanie povinných dôkazov v nástrojoch pre B-metódu.

Použitím predikátov (3.3), (3.4) môžeme bližšie určiť normálnu formu z vety 3.2.2:

Veta 3.2.3. *Nech S, x a x' sú rovnaké ako v prípade vety 3.2.2. Potom*

$$S = trm(S)|@x'.(prd_x(S) \Longrightarrow x := x')$$

Dôkaz vety 3.2.3 je tiež uvedený v [1]. Ako dôsledok vety môžeme konštatovať [1], že predikáty $trm(S)$ a $prd_x(S)$ kompletne charakterizujú GS S. Takisto dobre môžeme povedať, že S charakterizujú predikáty $trm(S)$ a $trm(S) \Rightarrow prd_x(S)$.

3.3 Abstraktný stroj - Machine

Ako už bolo spomenuté vyššie, formálna špecifikácia systému je v B-jazyku opísaná kolekciou abstraktných strojov, prepojených kompozičnými mechanizmami (časť 3.6), vďaka ktorým možno opísať aj značne rozsiahle systémy.

Abstraktný stroj (MACHINE), skrátene *B-stroj*, je svojou koncepciou blízky triede v objektovo orientovanom programovaní, či skôr balíku (package) v jazyku ADA. Slúži na zjednotenie (zapuzdrenie) kolekcie matematických prvkov, konštánt, množín, stavových premenných a súboru operácií na týchto premenných do pomenovaného modulu, ktorý potom podľa potreby môže byť videný inými komponentmi, alebo do nich začlenený. Premenné stroja môžu byť modifikované iba operáciami tohto stroja a nie operáciami iných strojov (okrem prípadu ak tieto operácie volajú operácie pôvodného stroja). Účelom tohto obmedzenia je najmä zjednodušenie povinných dôkazov stroja. Navyše to zapadá do konceptu B-stroja ako komponentu „vlastniaceho“ lokálne údaje a poskytujúceho operácie potrebné na manipuláciu s nimi a prístup k nim. Vo všeobecnosti má zápis abstraktného stroja (bez kompozičných klauzúl SEES,USES,...) formu:

MACHINE M(ST_PAR, sc_par)
CONSTRAINTS C
SETS St
ABSTRACT_CONSTANTS ak
CONCRETE_CONSTANTS kk
PROPERTIES B
ABSTRACT_VARIABLES av
CONCRETE_VARIABLES kv
DEFINITIONS D

INVARIANT I
ASSERTIONS A
INITIALISATION T
OPERATIONS
$y \leftarrow op(x) \mathrel{\widehat{=}}$ PRE P THEN S END
...
END

Nie každý B-stroj musí obsahovať všetky klauzuly. Ich obsah a význam je nasledujúci:

Hlavička stroja (klauzula MACHINE) obsahuje jeho meno M a parametre. Parametre rozlišujeme skalárne (sc_par) a množinové (ST_PAR). Hodnotou skalárneho parametra môže byť napríklad prirodzené číslo, alebo prvok z množiny definovanej v klauzule SETS či z množiny, ktorá je množinovým parametrom. Skalárne parametre sa používajú napríklad na obmedzenie rozsahu hodnôt premenných stroja. Množinový parameter je množina a predstavuje nový typ v danom abstraktnom stroji. Pri overení povinných dôkazov sú považované za neprázdne podmnožiny množiny celých čísel. Meno množinového parametra nesmie obsahovať malé písmená.

Obmedzenia (CONSTRAINTS) je klauzula definujúca logické vlastnosti parametrov p. Jej súčasťou musí byť pre každý skalárny parameter predikát určujúci jeho typ[5]. Obmedzenia stroja by nemali spájať rôzne množinové parametre, pretože tieto parametre sú považované za navzájom nezávislé.

Množiny (SETS). Klauzula SETS má tvar:

setdef_1;
...
setdef_n

Každá definícia setdef_i určuje novú množinu ako *enumeračnú* (enumerated set), alebo *odloženú* (deferred set). Enumeračná množina je určená vymenovaním jej prvkov a jej definícia (ak SS je meno množiny a počet jej prvkov je n) má tvar:

$$SS = \{val_1, \ldots val_n\}$$

[5]Typ výrazu v B-jazyku je najväčšia množina, ktorej je daný výraz prvkom.

Definícia odloženej množiny obsahuje iba meno množiny, bez bližšej informácie o jej prvkoch. O odloženej množine predpokladáme, že je konečná a neprázdna. Množiny sú tiež novými typmi a pri overení povinných dôkazov sú považované za neprázdne podmnožiny množiny celých čísel.

Abstraktné konštanty (ABSTRACT_CONSTANTS). Táto klauzula obsahuje zoznam identifikátorov $con_1, \ldots, con_n$, ktoré definujú n dátových prvkov, poskytovaných operáciám stroja v režime len na čítanie (read-only).

Konkrétne konštanty (CONCRETE_CONSTANTS alebo iba CONSTANTS). Podobne ako predchádzajúca klauzula obsahuje zoznam konštánt, ale v tomto prípade iba implementovateľných. Preto ich typom môže byť iba

$$S, \quad S \to T, \quad A_1 \times \ldots \times A_n \to T \quad \text{alebo} \quad \mathbb{P}(S),$$

kde S, T, $A_1 \ldots A_n$ sú skalárne množiny (tzn. množina prirodzených, prípadne celých čísel, množinové parametre či množiny definované v klauzule SETS). Tieto konštanty je možné používať aj v zjemneniach a implementácii stroja bez toho aby v nich boli explicitne uvedené.

Vlastnosti (PROPERTIES). Tu sú určené logické vlastnosti množín a konštánt (St, ak, kk). Obsahuje predikát, obyčajne konjunkciu formúl, zahŕňajúcu iba zavedené konštanty a množiny. Typové obmedzenie $c \in T$ (T je množina) alebo $c = hodnota$ musí byť prítomné pre každú konštantu c z ak, resp. kk.

Abstraktné stavové premenné (ABSTRACT_VARIABLES alebo iba VARIABLES) je zoznam $var_1, \ldots, var_n$ identifikátorov stavových premenných stroja. Stavové premenné sú lokálnymi údajmi stroja, vyjadrujú jeho stav, a ich hodnota nemôže byť priamo menená operáciami iného stroja (ich obsah možno „z vonku“ zmeniť len volaním príslušnej operácie toho B-stroja, v ktorom sú definované).

Konkrétne stavové premenné (CONCRETE_VARIABLES) je zoznam implementovateľných stavových premenných stroja. Implementovateľnosť znamená to isté, čo pre konkrétne konštanty.

Invariant (INVARIANT). Klauzula obsahuje predikát I určujúci vlastnosti stavových premenných (av aj kv) a teda vlastne invariantné (nemenné) vlastnosti celého stroja. Musí minimálne[6] definovať typy premenných. Práve platnosť I sa dokazuje povinnými dôkazmi.

Tvrdenia (ASSERTIONS). obsahuje ďalšie vlastnosti, odvodené z invariantu I. Zvyčajne ich uvádzame za účelom zjednodušenia overenia povinných dôkazov stroja. Keďže sa pri dôkaze platnosti tvrdenia využívajú tvrdenia uvedené pred ním, je ich poradie v rámci tejto klauzuly dôležité.

Definície (DEFINITIONS). Klauzula obsahuje zoznam makro-definícií v tvare:

abbdef_1;
...
abbdef_n

Každá definícia abbdef_i má tvar $data_i == expr_i$, kde $data_i$ je identifikátor alebo meno parametrizovanej funkcie $f(parametre)$ a $expr_i$ je výraz, predikát alebo GS v B-jazyku a môže obsahovať viditeľné premenné, konštanty a parametre f. Tieto definície slúžia ako makro-definície, napríklad

sqr(p) == ((p)*(p))

definuje funkciu druhej mocniny parametra p. Operátor „ ==“ znamená „prepísať na“. Makro-definície môžu byť uvedené aj v samostatnom súbore. Potom sa v tejto klauzule uvedie naň odkaz.

Inicializácia (INITIALISATION). Obsahuje zovšeobecnenú substitúciu, ktorej vykonaním sa každej stavovej premennej priradí počiatočná hodnota, čím sa B-stroj dostane do počiatočného stavu. Inicializácia môže byť aj nedeterministická.

Operácie (OPERATIONS). Obsahuje zoznam definícií operácií v tvare:

$\mathsf{opheader}_1 \mathrel{\widehat{=}} \mathsf{opdef}_1$;
...
$\mathsf{opheader}_n \mathrel{\widehat{=}} \mathsf{opdef}_n$

[6] Invarianty, ktoré len definujú typy premenných nazývame *elementárne*.

Hlavička operácie, opheader, má vo všeobecnosti formu:

$$y \leftarrow opname(x).$$

Vstupné parametre (x) a výstupné parametre (y) sú voliteľné a môžu chýbať. Časť opdef je zovšeobecnená substitúcia, ktorá má vo všeobecnosti tvar

PRE P THEN S END

kde P je predikát definujúci pre-podmienku operácie a S je GS tvoriaca telo operácie. P musí definovať typy parametrov z x. Ak $P = true$, zvykne sa opdef zapisovať iba ako S, alebo ako

BEGIN S END.

V rámci inicializácie a operácií abstraktného stroja môžu byť použité všetky zovšeobecnené substitúcie okrem sekvenčnej kompozície (;) a konštruktov cyklu WHILE. Dôvodom pre ich vylúčenie je požiadavka podrobne definovať operácie tak, aby úzko súviseli so stavovými prechodmi, ktoré majú byť dosiahnuté. Abstraktná špecifikácia by mala byť jasne zrozumiteľná a nemala by vyžadovať analýzu medzistavov, ktoré môžu vzniknúť použitím ; alebo WHILE. Ich vylúčenie tiež zdôrazňuje deklaratívnu povahu MACHINE: operácie abstraktného stroja majú určovať *čo* sa má urobiť a nie *ako* sa to má urobiť. V MACHINE nemožno používať ani konštrukt VAR, pretože nedefinuje typy lokálnych premenných, ktoré zavádza. Pre bližšiu ilustráciu v príklade 3.3.1 uvedieme jednoduchú abstraktnú špecifikáciu, pozostávajúcu z dvoch strojov.

Príklad 3.3.1 (parkovisko–abstraktné stroje)**.** Predstavme si, že je našou úlohou definovať jednoduchý riadiaci systém pre parkovisko, ktorý má vedieť parkovisko otvoriť, zatvoriť, vpustiť naň auto a dovoliť z neho autu odísť. Táto funkcionalita má byť prístupná cez operácie otvorParko, zatvorParko, prichodAuta a odchodAuta. Navyše majú platiť dve dôležité vlastnosti:

1. počet áut na parkovisku nesmie prekročiť stanovenú kapacitu a
2. iba prázdne parkovisko je možné zavrieť.

Riadiaci systém môžeme špecifikovať vo forme abstraktného stroja Parkovisko, ktorého zápis v B-jazyku je nasledujúci:

```
MACHINE Parkovisko(kapacita)
CONSTRAINTS kapacita:NAT1
SEES ParkoviskoMn
VARIABLES stavParko, auta
INVARIANT
  stavParko: STAVY_PARK & auta<:NAT &
  card(auta)<=kapacita &
  (stavParko=zatvorene => card(auta)=0)
INITIALISATION
  auta:={} || stavParko:=zatvorene

OPERATIONS

  otvorParko = BEGIN stavParko:=otvorene END;

  ok<--zatvorParko =
    PRE card(auta)=0
    THEN stavParko:=zatvorene||ok:=TRUE END;

  ok <-- prichodAuta(autoId) =
    PRE autoId:NAT & autoId/:auta &
        stavParko=otvorene & card(auta)<kapacita
    THEN auta:=auta\/{autoId}||ok:=TRUE END;

  ok <-- odchodAuta(autoId) =
    PRE autoId:NAT & autoId:auta & stavParko=otvorene
    THEN auta:=auta-{autoId}||ok:=TRUE END
END
```

Ako vidno, kapacitu parkoviska sme sa rozhodli reprezentovať skalárnym parametrom kapacita a dôležité vlastnosti vyjadrujú druhý a tretí riadok klauzuly INVARIANT. Jej prvý riadok určuje typy stavových premenných. Tie sú dve: stavParko uchováva stav parkoviska, ktorý je prvkom z množiny STAVY_PARK. Premenná auta uchováva množinu identifikátorov áut (prirodzených čísel), ktoré sú na parkovisku. Preto je identifikátor prichádzajúceho, resp. odchádzajúceho auta parametrom (autoId) posledných dvoch operácií. Množina STAVY_PARK je špecifikovaná v samostatnom stroji ParkoviskoMn:

```
MACHINE ParkoviskoMn
SETS STAVY_PARK = {otvorene, zatvorene} END
```

Stroj ParkoviskoMn je stroju Parkovisko sprístupnený mechanizmom SEES, bližšie popísaným v časti 3.6. To umožňuje stroju Parkovisko používať množinu STAVY_PARK.

□

3.3.1 Povinné dôkazy

Úlohou povinných dôkazov (POb) je overiť či v každom stave stroja budú platiť vlastnosti definované v klauzule INVARIANT. Tieto dôkazy sú uvedené v tabuľke 3.19. Prvé tri preverujú, či vôbec existujú údajové

(1)	$\exists(ST\_PAR, sc\_par).C$
(2)	$ST_{int} \wedge C \Rightarrow \exists(St, ak, kk).B$
(3)	$ST_{int} \wedge B \wedge C \Rightarrow \exists av, kv.I$
(4)	$ST_{int} \wedge B \wedge C \Rightarrow [T]I$
(5)	$ST_{int} \wedge B \wedge C \wedge I \wedge P \Rightarrow [S]I$

Tabuľka 3.19: Povinné dôkazy abstraktného stroja

členy vyhovujúce stanoveným podmienkam. Dôkazy (4) a (5) overujú nastolenie invariantu pri inicializácii a jeho zachovania operáciami stroja. ST_{int} je predikát tvrdiaci, že „všetky množinové parametre a množiny z klauzuly SETS sú neprázdne podmnožiny množiny celých čísel“. Je potrebné povedať, že názory na to, čo medzi POb patrí sa v literatúre líšia. Napríklad [43] uvádza všetkých päť dôkazov, zatiaľ čo [1] iba posledné dva. Slovne možno význam invariantov opísať nasledovne:

(1) *Existencia parametra.* Preveruje, či existujú hodnoty parametrov stroja spĺňajúce obmedzenia C. Ak nie, tak tento stroj nemôže byť správne inštanciovaný a neexistuje ani program vyhovujúci danej abstraktnej špecifikácii.

(2) *Existencia konštánt a množín.* Preveruje, či existujú konštanty ak, kk a množiny St vyhovujúce vlastnostiam B. V opačnom prípade neexistuje program vyhovujúci špecifikácii.

(3) *Existencia stavu stroja.* Overuje, či vzhľadom na vlastnosti B a obmedzenia C stroja, existujú nejaké hodnoty stavových premenných av a kv, ktoré spĺňajú invariant I.

(4) *Inicializácia.* Overuje, či inicializácia T zavádza platnosť invariantu I, vzhľadom na vlastnosti B a obmedzenia C stroja. Ak T túto podmienku nespĺňa, potom je narušený základný predpoklad platnosti invariantu I počas životnosti implementácie stroja, tzn. jeho platnosť v počiatočnom stave stroja.

(5) *Zachovanie invariantu.* Overuje, či každá operácia (ktorej telo tvorí zovšeobecnená substitúcia S) zachováva platnosť invariantu I, ak je volaná za platnosti svojej pre-podmienky P (a invariantu I). Ak to neplatí, potom nie je záruka, že invariant I je pravdivý počas celej životnosti implementácie stroja.

Dôkaz (5) je potrebné vykonať pre každú operáciu stroja.

(5)	$ST_{int} \wedge B \wedge C \wedge I \wedge P \wedge A \Rightarrow [S]I$
(6)	$ST_{int} \wedge B \wedge C \wedge I \Rightarrow A$

Tabuľka 3.20: Modifikované POb pre stroj s ASSERTIONS

Ak je použitá klauzula ASSERTIONS, modifikuje sa piaty a pridáva šiesty dôkaz (tab. 3.20).

Príklad 3.3.2 (parkovisko–povinné dôkazy)**.** Prvý až štvrtý POb pre B-stroj Parkovisko z príkladu 3.3.1 budú nasledujúce predikáty:

(1) $\exists(\mathsf{kapacita}).\mathsf{kapacita} \in \mathsf{NAT1}$
(2) $ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \Rightarrow true$
(3) $ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \Rightarrow \exists \mathsf{stavParko}, \mathsf{auta}.I$
(4) $ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \Rightarrow [\mathsf{auta}{:=}\{\} \| \mathsf{stavParko}{:=}\mathsf{zatvorene}]I$

kde I je predikát z klauzuly INVARIANT stroja Parkovisko. Nie je ťažké sa presvedčiť, že tieto tvrdenia sú pravdivé. Dôkaz (5) sa vykonáva pre každú operáciu stroja, ukážeme si ho pre zatvorParko:

$$ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \wedge I \wedge |\mathsf{auta}| = 0 \Rightarrow [\mathsf{stavParko}{:=}\mathsf{zatvorene} || \mathsf{ok}{:=}\mathsf{TRUE}]I \tag{3.9}$$

Po vyjadrení podľa tabuľky 3.12 bude mať (3.9) tvar (3.10).

$$\begin{array}{l} ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \wedge I \wedge |\mathsf{auta}| = 0 \Rightarrow \\ (\mathsf{zatvorene} \in \mathsf{STAVY\_PARK} \wedge \mathsf{auta} \subseteq \mathbb{N} \wedge |\mathsf{auta}| \leq \mathsf{kapacita} \wedge \\ (\mathsf{zatvorene} = \mathsf{zatvorene} => |\mathsf{auta}| = 0)) \end{array} \tag{3.10}$$

Keď v (3.10) eliminujme pravdivé tvrdenia a rozvinieme potrebné časti z I na ľavej strane (tzn. pred implikáciou) dostaneme (3.11).

$$\begin{array}{r} ST_{int} \wedge \mathsf{kapacita} \in \mathsf{NAT1} \wedge \mathsf{auta} \subseteq \mathbb{N} \wedge |\mathsf{auta}| = 0 \Rightarrow \\ (\mathsf{auta} \subseteq \mathbb{N} \wedge |\mathsf{auta}| \leq \mathsf{kapacita} \wedge |\mathsf{auta}| = 0) \end{array} \tag{3.11}$$

Je zrejmé, že tvrdenie (3.11) je pravdivé. K rovnakému záveru by sme prišli aj v prípade ostatných operácií. V tomto príklade sme použili matematický zápis výrazov a predikátov.

□

3.4 Zjemnenie - Refinement

Zjemňovanie je procesom prechodu od abstraktnej špecifikácie stroja k menej abstraktnej špecifikácii, pomocou transformácie jej operácií alebo údajov. Zjemňovaním vlastne postupne „prerábame“ abstraktnú špecifikáciu do implementovateľnej podoby, pričom je nutné, aby vonkajšie správanie zjemneného komponentu ostalo rovnaké, resp. nerozoznateľné od správania zjemňovaného komponentu. V rámci zjemňovania tiež môžeme do špecifikácie zahrnúť viac detailov z pôvodného, neformálneho, opisu systému. Krok zjemňovania špecifikácie môže byť od abstraktného stroja k zjemneniu (REFINEMENT), od zjemnenia k zjemneniu, alebo od zjemnenia k implementácii (obr. 3.1 b)). Špecifikácia zjemnenia N abstraktného stroja M má nasledujúce vlastnosti:

1. Musí obsahovať klauzulu REFINES M, čím označuje *jedinú* komponentu M, ktorú zjemňuje.
2. Musí mať identické parametre[7] a hlavičky operácií ako komponent, ktorý zjemňuje.
3. V operáciách a inicializácii možno použiť všetky zovšeobecnené substitúcie použiteľné v abstraktnom stroji a tiež sekvenčnú kompozíciu (;).

[7] Klauzula CONSTRAINTS sa vynecháva.

4. Niektorý z konjunktov invariantu zjemnenia identifikuje ako stav zjemnenia vyjadruje stav zjemňovaného komponentu.

5. V operáciách zjemnenia je možné volať opytovacie operácie[8] videného[9] stroja.

Platí, že iba jediný komponent môže byť zjemnený daným konkrétnym zjemnením a v klauzule REFINES zjemnenia (alebo implementácie) N sa neuvádzajú parametre. Je to preto, že zjemnenie a zjemňovaný komponent považujeme za dva alternatívne popisy toho istého systému, ktoré majú rovnaké vonkajšie správanie (rozhranie) z hľadiska parametrov, množiny operácií a mien operácií. Vnútorná implementácia operácií je však rozdielna.

V procese zjemnenia môže byť zmenený kód inicializácie a operácií (*procedurálne zjemnenie*) a tiež údajové členy komponentu. *Procedurálne zjemnenie* je formálne definované pomocou operátora „$\sqsubseteq$“ nasledovne:

Veta 3.4.1. *Nech S_1 a S_2 sú GS a P predikát. Potom S_2 zjemňuje S_1 (zapisujeme $S_1 \sqsubseteq S_2$) práve vtedy, ak*

$$\forall P.[S_1]P \Rightarrow [S_2]P.$$

Typickým procedurálnym zjemnením je oslabenie pre-podmienky či obmedzenie nedeterminizmu. Zápis zjemnenia N, ktoré zjemňuje abstraktný stroj $M(p)$ (kapitola 3.3) má v B-jazyku formu:

REFINEMENT N
REFINES M
SETS $St1$
ABSTRACT_CONSTANTS $ak1$
CONCRETE_CONSTANTS $kk1$
PROPERTIES $B1$
ABSTRACT_VARIABLES aw
CONCRETE_VARIABLES kw
DEFINITIONS $D1$
INVARIANT J
ASSERTIONS $A1$
INITIALISATION $T1$

[8] Opytovacie operácie (enquiry operations) sú operácie, ktoré vracajú hodnoty premenných stroja, ale nemenia ich.

[9] tzn. sprístupneného pomocou SEES

OPERATIONS
$y \leftarrow op(x) \mathrel{\widehat{=}}$
PRE $P1$
THEN $S1$
END
...
END

Predikát J v invariante zjemnenia obsahuje určenie typov a obmedzení lokálnych premenných aw, kw a *reláciu zjemnenia* medzi premennými aw, kw a av, kv. *Relácia zjemnenia*, R, vyjadruje vzťah medzi stavmi zjemňovaného komponentu a stavmi jeho zjemnenia.

Príklad 3.4.1 (parkovisko–zjemnenie)**.** Ako príklad uvedieme zjemnenie riadiaceho systému parkoviska z príkladu 3.3.1. V abstraktnom stroji Parkovisko sme, vzhľadom na jeho účel, zvolili zbytočne „štedrú“ údajovú reprezentáciu. Konkrétne, v premennej auta sme si pamätali údaje (identifikačné číslo) o každom aute na parkovisku, pritom postačí sledovať počet áut. Preto B-stroj Parkovisko zjemníme na zjennenie Parkovisko_r, kde premenná auta bude nahradená premennou pocetAut. Ide o zjemnenie údajov, ktoré spôsobuje aj procedurálne zjemnenie. Reláciou zjemnenia tu bude predikát pocetAut=card(auta). Špecifikácia zjemnenia je nasledujúca:

```
REFINEMENT Parkovisko_r(kapacita)
REFINES Parkovisko
SEES ParkoviskoMn
VARIABLES stavParko
CONCRETE_VARIABLES pocetAut
INVARIANT
  pocetAut: NAT & pocetAut<=kapacita &
  pocetAut=card(auta) &
  (stavParko=zatvorene => pocetAut=0)
INITIALISATION
  pocetAut:=0 || stavParko:=zatvorene

OPERATIONS
  otvorParko = BEGIN stavParko:=otvorene END;

  ok<--zatvorParko =
    PRE pocetAut=0
    THEN stavParko:=zatvorene||ok:=TRUE END;
```

```
  ok <-- prichodAuta(autoId) =
    PRE autoId:NAT & stavParko=otvorene &
        pocetAut<kapacita
    THEN pocetAut:=pocetAut+1||ok:=TRUE END;

  ok <-- odchodAuta(autoId) =
    PRE autoId:NAT & stavParko=otvorene & pocetAut>0
    THEN pocetAut:=pocetAut-1||ok:=TRUE END
END
```

Premennú pocetAut sme definovali ako konkrétnu, čo znamená že sa zachová nezmenená až do implementácie a v ďalším komponentoch ju už nebude potrebné deklarovať.

□

3.4.1 Povinné dôkazy

Povinné dôkazy zjemnenia sú uvedené v tabuľke 3.21. Význam dôkazov

(1)	$ST_{int} \wedge C \wedge B \wedge B1 \Rightarrow \exists(av, kv, aw, kw).(I \wedge J)$
(2)	$ST_{int} \wedge C \wedge B \wedge B1 \Rightarrow [T1]\neg[T]\neg J$
(3)	$ST_{int} \wedge C \wedge B \wedge B1 \wedge I \wedge J \wedge P \Rightarrow P1 \wedge [S1']\neg[S]\neg(J \wedge y' = y)$

Tabuľka 3.21: Povinné dôkazy zjemnenia

je nasledujúci:

(1) *Existencia zmiešaného stavu.* Existuje kombinovaný abstraktný a konkrétny stav, ktorý vyhovuje relácii zjemnenia a invariantu abstraktného stroja. V prípade neplatnosti nebude existovať vykonateľná implementácia abstraktnej špecifikácie prostredníctvom tohto zjemnenia.

(2) *Zjemnenie inicializácie.* Ak platí, tak inicializácia zjemnenia $T1$ vytvorí situáciu, v ktorej inicializácia špecifikácie stroja T nemôže zlyhať pri stanovení podmienok J.

(3) *Zjemnenie operácie.* Zabezpečenie korektnosti operácií a verifikácia, či operácia $S1$ zjemnenia vytvorí situáciu, v ktorej operácia S zjemňovaného komponentu nemôže zlyhať pri splnení pod-

mienok invariantu J. $S1'$ je $S1$ v ktorej sú výstupné premenné y nahradené[10] premennými y', čo sú premenné y s doplneným znakom „′“. Dôkaz je potrebné vykonať pre každú operáciu zjemnenia.

Jedným z dôsledkov uvedených povinných dôkazov je, že zjemňovať možno iba plne uskutočniteľnú (feasible) operáciu. GS totiž možno zjemniť iba na tak isto, alebo menej uskutočniteľnú GS a operácie implementácie musia byť plne uskutočniteľné. Preto je dobré sa ešte pred zjemňovaním presvedčiť, či sú plne uskutočniteľné všetky operácie B-stroja. Formálny dôkaz uskutočniteľnosti operácií (tzn. či ich $fis = true$) sa medzi POb abstraktného stroja nenachádza iba kvôli jeho vysokej náročnosti. Tá môže byť rovná vypracovaniu implementácie danej operácie.

3.5 Implementácia - Implementation

Implementácia stroja reprezentuje v B-metóde posledný krok vývoja systému. Z nej je možné priamo generovať vykonateľný kód. Implementácie určené na preklad zvyčajne importujú abstraktnú špecifikáciu existujúceho (už implementovaného) systému, a to pomocou klauzuly IMPORTS, popísanej v časti 3.6. Špeciálna klauzula VALUES v implementácii zabezpečuje, aby sa všetkým konštantám zjemňovaného komponentu priradila presná hodnota, vyhovujúca ich definíciám. V tejto klauzule sa tiež bližšie špecifikujú odložené množiny. V implementácii môžu byť použité iba konkrétne konštanty a premenné. Vstupom a výstupom operácie implementácie môžu byť taktiež iba parametre implementovateľných typov.

Importované stroje (musia existovať ich korektne vykonateľné implementácie) musia byť inštanciované konkrétnymi množinami a skalárnymi parametrami pri zavedení do implementácie stroja. Tieto parametre často súvisia s parametrami implementovaného stroja. Ak je potrebné viacnásobné importovanie toho istého stroja, použije sa premenovanie na odlíšenie jeho kópií (pozri príklad 4.2.2). *Povinné dôkazy* implementácie sú zhodné s dôkazmi pre zjemnenie s tým rozdielom, že v nich nefigurujú abstraktné údaje.

[10]Nahradenie je potrebné, aby boli jednoznačne odlíšené výstupné premenné operácie zjemneného (y') a zjemňovaného komponentu (y).

V operáciách implementácie stroja sú povolené nasledujúce formy GS:

- priradenie ($v := e$),
- WHILE E DO S INVARIANT I VARIANT ve END,
- IF E THEN S_1 ELSE S_2 END a iné podmienkové príkazy, ktoré sú variantmi alebo komplexnejšími kombináciami tohto príkazu,
- VAR v IN S END taký, že lokálne premenné v sú inicializované pred prvým čítaním,
- volania operácií importovaných strojov a opytovacích operácií videných strojov a
- sekvenčná kompozícia (;).

Týchto šesť zovšeobecnených substitúcií je postačujúcich na implementáciu ľubovoľného algoritmu.

Príklad 3.5.1 (parkovisko–implementácia)**.** V implementácii Parkovisko_i nášho riadenia parkoviska sme sa rozhodli vymeniť premennú stavParko za importovaný B-stroj ParkStav:

```
MACHINE ParkStav
SEES ParkoviskoMn
VARIABLES stav
INVARIANT stav: STAVY_PARK
INITIALISATION stav:=zatvorene

OPERATIONS
  otvor = BEGIN stav:= otvorene END;
  zatvor= BEGIN stav:=zatvorene END;

  odp <-- jeOtvorene =
    IF stav = otvorene THEN odp:=TRUE
    ELSE odp:=FALSE END;

  odp <-- jeZatvorene =
    IF stav = zatvorene THEN odp:=TRUE
    ELSE odp:=FALSE END
END
```

Stroj ParkStav je zjemnený na implementáciu ParkStav_i, ktorá sa od neho líši zmenou domény stavovej premennej zo STAVY_PARK na $\mathbb{N}$:

```
IMPLEMENTATION ParkStav_i
REFINES ParkStav
SEES ParkoviskoMn
CONCRETE_VARIABLES stav_i
INVARIANT
    stav_i:0..1 &
    ((stav=zatvorene) <=> (stav_i = 0)) &
    ((stav=otvorene) <=>  (stav_i = 1))
INITIALISATION stav_i := 0

OPERATIONS
  otvor  = BEGIN stav_i:=1 END;
  zatvor = BEGIN stav_i:=0 END;

  odp <-- jeOtvorene =
    IF stav_i = 1 THEN odp:=TRUE
    ELSE odp:=FALSE END;

  odp <-- jeZatvorene =
    IF stav_i = 0 THEN odp:=TRUE
    ELSE odp:=FALSE END
END
```

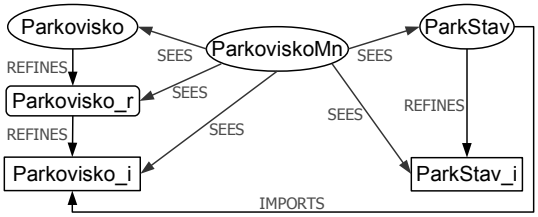
Implementácia Parkovisko_i stroj ParkStav importuje a jej relácia zjemnenia, stavParko=stav, vyjadruje vzťah medzi premennou stavParko z Parkovisko_r a stav z ParkStav.

```
IMPLEMENTATION Parkovisko_i(kapacita)
REFINES Parkovisko_r
SEES ParkoviskoMn
IMPORTS ParkStav
INVARIANT stavParko=stav
INITIALISATION
   pocetAut := 0; zatvor

OPERATIONS
  otvorParko = otvor;

  ok <-- zatvorParko =
    IF pocetAut = 0 THEN zatvor; ok := TRUE
```

```
    ELSE ok := FALSE END;

  ok <-- prichodAuta(autoId) =
    VAR sp IN
      sp<--jeOtvorene;
      IF sp = TRUE & pocetAut < kapacita THEN
        pocetAut := pocetAut + 1; ok := TRUE
      ELSE ok := FALSE END
    END;

  ok <-- odchodAuta(autoId) =
    VAR sp IN
      sp<--jeOtvorene;
      IF sp = TRUE & pocetAut > 0 THEN
        pocetAut := pocetAut - 1; ok := TRUE
      ELSE ok := FALSE END
    END
END
```

□

Celková špecifikácia riadiaceho systému parkoviska, predstavená v príkladoch 3.3.1 až 3.5.1, teda pozostáva zo šiestich komponentov v B-jazyku. Ich vzájomné prepojenie je znázornené na obr. 3.2. Na obr. 3.2

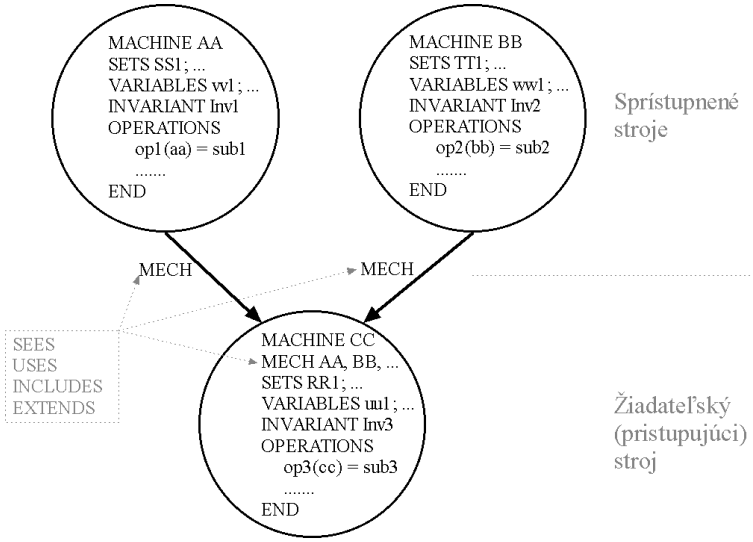


Obr. 3.2: Diagram kompozície špecifikačných komponentov riadiaceho systému parkoviska

sme typy komponentov odlíšili podobne ako na obr. 3.1: elipsy, resp. kruhy, sú abstraktné stroje, zaoblené obdĺžniky zjemnenia a obdĺžniky implementácie. Túto konvenciu dodržíme aj v nasledujúcich častiach.

3.6 Kompozičné mechanizmy

B-metóda poskytuje 6 mechanizmov na vzájomné prepojenie (sprístupnenie) abstraktných strojov (M), zjemnení (R) a implementácií (I). V závislosti na spôsobe sprístupnenia, tzn. na zvolenom mechanizme, sa rôzne časti (operácie, premenné,...) sprístupňovaného (accessed) stroja stávajú na rôznych úrovniach „viditeľné“ v pristupujúcom (accessing) stroji, alebo je s nimi nakladané, akoby boli priamo definované v pristupujúcom stroji. Všeobecná schéma sprístupnenia prostriedkov jedného stroja iným strojom je znázornená na obr. 3.3 („MECH“ je názov použitého mechanizmu). Na reprezentáciu sprístupnenia jedného



Obr. 3.3: Všeobecná schéma sprístupnenia abstraktných strojov

stroja inému stroju sa v grafickom zobrazení používajú hrany, označené názvom daného mechanizmu, orientované od sprístupneného k pristupujúcemu stroju. Do špecifikácie pristupujúceho stroja pribudne klauzula (klauzuly - jedna pre každý použitý mechanizmus) v tvare

MECH m_1, ..., m_k

$MECH$ je názov mechanizmu a $m_1, \ldots, m_k$ zoznam strojov týmto mechanizmom sprístupnených. K dispozícii sú mechanizmy SEES, USES,

INCLUDES, EXTENDS, REFINES a IMPORTS:

SEES je povolený v M, R aj I[11], no videný môže byť iba M. V M sprístupňuje množiny, konštanty a premenné tak, ako to ukazuje tab. 3.22. V R a I je možné použiť aj opytovacie operácie videného stroja. V I však možno použiť (abstraktné) premenné videného stroja iba v invariantoch cyklov. Viac ako jeden M, R, I môže vidieť daný abstraktný stroj. Hlavným účelom mechanizmu je podporiť oddelený vývoj subsystému, ktorý je (pomocou SEES) zdieľaný v read-only režime inými subsystémami v rámci aplikácie. Premenné videného stroja totiž v pristupujúcom komponente možno iba čítať. Ak je stroj videný viacerými implementáciami, potom bude zvyčajne importovaný práve jednou.

USES je povolený len v M. Jeden stroj môže byť použitý viacerými strojmi. Hlavným účelom USES je sprostredkovanie zdieľaného prístupu k stavovým údajom ako SEES, z dôvodu uľahčenia konštrukcie špecifikácie vývoja subsystému. Ak stroj A je používaný viacerými komponentmi špecifikácie, ktoré budú obsiahnuté (included) alebo rozšírené (extended) jednoduchou špecifikáciou stroja, tak stroj A bude zvyčajne obsiahnutý v tomto stroji.
USES predstavuje silnejšiu formu prístupu ako SEES - premenné použitého stroja síce tiež možno iba čítať, no dajú sa použiť aj v invariante pristupujúceho stroja, čo neplatí pre SEES. Táto silnejšia forma sprístupnenia zabraňuje použitým a používajúcim strojom, aby boli oddelene zjemnené až do vykonateľného kódu. Podobne ako u SEES ani v prípade USES nemôže pristupujúci stroj použiť aktualizačné operácie (update operations) sprístupneného a meniť tak jeho stav (hodnoty jeho premenných).
Stav použitého stroja môže byť zmenený operáciami subsystému v ktorom sa vyskytuje, pretože hlavný stroj, ktorý špecifikuje subsystém, zvyčajne obsiahne alebo rozšíri tento stroj a tým získa prístup k jeho operáciám.

INCLUDES (obsiahnutie) je možné použiť v M a R, sprístupniť sa však dá iba M. Ak stroj M_1 vložíme do M_2 (M_2 INCLUDES M_1), potom vlastne získame nový stroj, ktorého klauzuly SETS až INITIALISATION vzniknú zreťazením či konjunkciou obsahov príslušným klauzúl M_1 a M_2.

[11] M - abstraktný stroj, R - zjemnenie, I - implementácia

Operácie M_1 je možné volať z operácií M_2, avšak maximálne 1 operáciu M_1 z danej operácie M_2. Volanie operácie má rovnakú syntax ako hlavička operácie. Samozrejme, formálne vstupné a výstupné parametre operácie sú nahradené skutočnými. Niektoré operácie M_1 sa môžu stať operáciami M_2 (tzn. môžu byť zavedené do M_2) - je ich však potrebné uviesť v klauzule PROMOTES v M_2. Pri vložení sú formálne parametre M_1 inštanciované (nahradené) aktuálnymi. Ak je potrebné viacnásobné vloženie toho istého stroja, priradia sa jednotlivým jeho kópiám jednoznačné mená (tzv. premenovanie).

Iba jeden stroj môže obsiahnuť daný stroj. Hlavným prínosom použitia INCLUDES je podpora hierarchického rozvrstvenia subsystémov (vývoja subsystému), Skupina strojov obsiahnutá alebo rozšírená daným strojom sa označuje pojmom *siblings*.

REFINES sa používa v R a I a identifikuje komponent (M alebo R), ktorý je zjemnený komponentom v ktorom sa táto klauzula nachádza. Stroj alebo zjemnenie môže byť zjemnené dvoma alebo viacerými spôsobmi, ale zjemnenie a implementácia môžu byť zjemnením práve jedného komponentu.

IMPORTS sa používa iba v I. Importovať je možné iba abstraktné stroje. Voliteľná klauzula PROMOTES označuje zavedené operácie z importovaného stroja do importujúcej implementácie. Práve jedna implementácia v konkrétnom vývoji systému môže importovať daný stroj.

Z hľadiska viditeľnosti je IMPORTS podobný INCLUDES, líši sa však v tom, že abstraktné premenné importovaného stroja sú v operáciách importujúcej implementácie viditeľné iba v invariantoch cyklov. Podobne ako u INCLUDES aj tu možno použiť premenovanie.

EXTENDS (rozšírenie) je použiteľné v M, R aj I. V M a R EXTENDS znamená INCLUDES s automatickým zavedením všetkých operácií sprístupneného stroja, v I IMPORTS s automatickým zavedením. Dá sa povedať [43], že EXTENDS korešponduje s dedením v objektovo orientovanom programovaní.

Prehľad sprístupnenia poskytovaného jednotlivými klauzulami na úrovni abstraktných strojov poskytuje tab. 3.22. V tabuľke predpo-

kladáme, že stroj M_2 pristupuje k stroju M_1. To, že dané prvky z M_1 sú prístupné príslušnej klauzule M_2 je indikované uvedením prvého znaku názvu mechanizmu v danom políčku tabuľky. Stĺpec „INCL.EXT. param.“ znamená parametre strojov sprístupnených stroju M_2 pomocou INCLUDES alebo EXTENDS.

M_2 / M_1	INCL.EXT. param.	PROPERTIES	INVARIANT	OPERATIONS
parametre			U	U
množiny	S	S,U,I	S,U,I	S,U,I
konštanty	S	S,U,I	S,U,I	S,U,I
stavové premenné			U,I	S,U,I (iba čítanie)
operácie				I

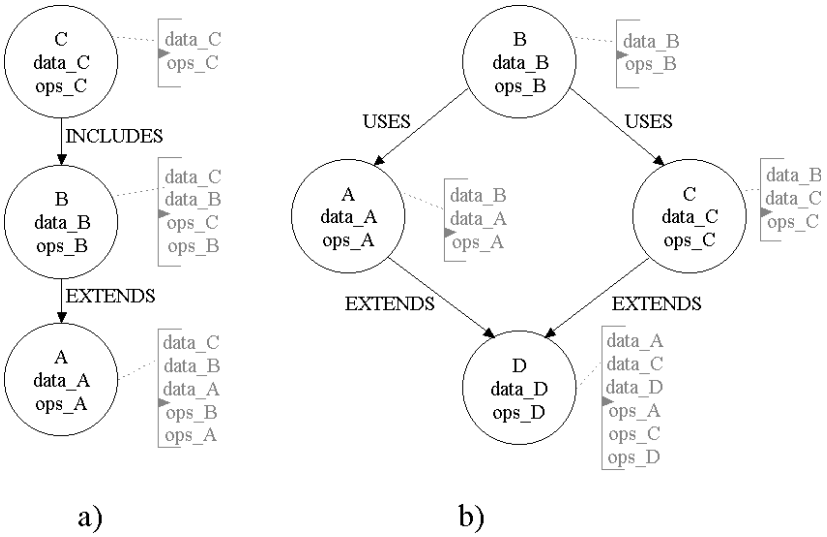
Tabuľka 3.22: Sprístupnenie na úrovni abstraktných strojov

USES, INCLUDES a EXTENDS sú mechanizmy *polovičného utajenia* (semi-hiding), čo znamená, že umožňujú čítať premenné sprístupneného komponentu operáciami pristupujúceho, ale nepovoľujú ich priamu aktualizáciu pristupujúcim komponentom. Tieto mechanizmy zabraňujú oddelenému zjemneniu sprístupneného komponentu nezávisle od pristupujúceho, pretože pristupujúci komponent závisí od reprezentácie údajov (od množiny premenných), ktorá je prítomná v sprístupnenom.

REFINES, SEES (v implementáciách) a IMPORTS sú mechanizmy *úplného utajenia* (full-hiding), pretože neumožňujú ani čítajúci prístup k premenným sprístupneného komponentu v operáciách pristupujúceho, mimo invariantu cyklu. Tieto mechanizmy podporujú nezávislé zjemnenie.

USES a SEES sú *intranzitívne* v zmysle, že ak A používa (uses) B a B používa C, potom A nemá prístup k žiadnemu prvku definovanému v C. Intranzitivita je zavedená kvôli úlohe týchto mechanizmov podporovať zdieľaný prístup ku komponentom. Konkrétne, ak umožníme dvom komponentom A a B pristúpiť pomocou USES alebo SEES k tretiemu komponentu C a ak A a B sú obsiahnuté (included) v komponente D, tak by pri absencii intranzitivity existovali dve kópie prvkov C v komponente D.

Na ilustráciu uvedieme dva príklady, objasňujúce použitie a efekt me-



Obr. 3.4: Príklady použitia kompozičných mechanizmov

chanizmov:

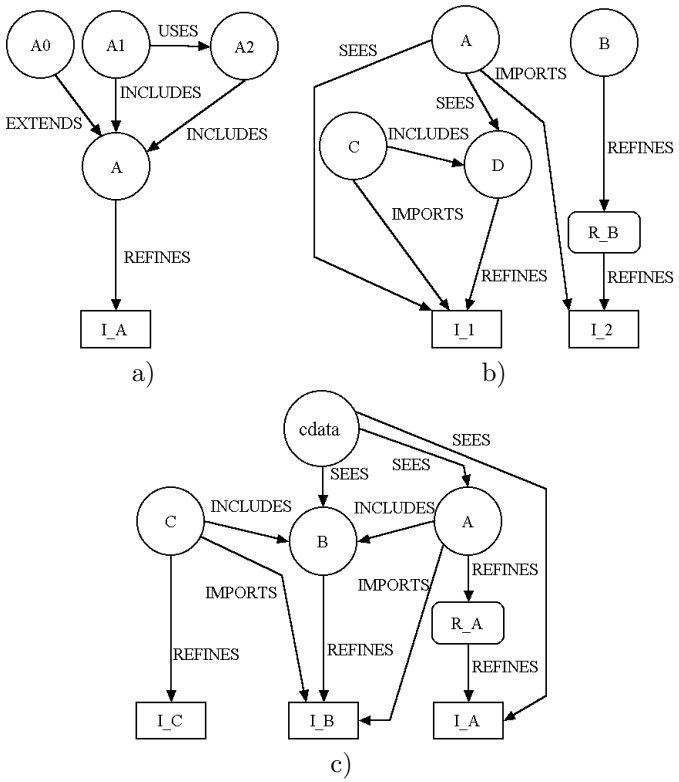
A EXTENDS *B; B* INCLUDES *C* (obr. 3.4 a)): *A* obsahuje všetky premenné, ktoré sú explicitne deklarované v *A*, plus premenné deklarované v *B* a *C*. Podobne pre ostatné údajové prvky. Avšak operácie *A* budú iba tie, ktoré sú explicitne deklarované v *A* alebo v *B*.

A USES *B; C* USES *B; D* EXTENDS *A*,*C* (obr. 3.4 b)): *D* obsahuje údajové prvky explicitne deklarované v *D*, *A* alebo *C*, ale nie v *B*. Operácie *D* sú tiež iba tie, ktoré sú explicitne deklarované v *D*, *A* a *C*.

3.6.1 Stratégie vývoja v B

Medzi dekompozíciou systému na úrovni abstraktnej špecifikácie a na úrovni implementácie nemusí byť žiaden priamy vzťah. Na základe toho aký vzťah medzi nimi existuje rozlišujeme [43] 3 stratégie vývoja v B (obr. 3.5):

Monolitický prístup (*monolithic approach*) - na úrovni implementácie je len jeden systém (jedna implementácia), ktorý importuje



Obr. 3.5: Príklady diagramov kompozície špecifikačných komponentov pre jednotlivé stratégie vývoja v B: monolitický prístup (a), samostatná dekompozícia (b) a nepretržitosť štruktúry (c)

množinu abstraktných strojov, tvoriacich systém na úrovni abstraktnej špecifikácie. Na implementačnej úrovni teda neexistuje dekompozícia, preto možno pri vytváraní špecifikácie použiť všetky dostupné kompozičné mechanizmy.

Samostatná dekompozícia (*separate decomposition*) - dekompozícia na úrovni abstraktnej špecifikácie a na úrovni implementácie spolu priamo nesúvisia, počas vývoja sa objavujú nové subsystémy. O úplnej nezávislosti týchto dvoch dekompozícii možno však

uvažovať len v prípade, ak na úrovni špecifikácie sú použité len mechanizmy úplného utajenia. Táto stratégia je použitá v špecifikácii riadiaceho systému parkoviska v príkladoch 3.3.1 až 3.5.1 (pozri obr. 3.2).

Nepretržitosť štruktúry (*continuity of structure*) - dekompozícia implementácie je (takmer) identická s dekompozíciou abstraktnej špecifikácie. Každý abstraktný stroj je samostatne zjemnený až do vykonateľného kódu. Ak je nejaký stroj A (pomocou INCLUDES) obsiahnutý v inom stroji B, potom implementácia B väčšinou importuje stroj A. To spôsobí, že všetky množiny a konštanty z A budú dva krát viditeľné v implementácii B (raz cez klauzulu IMPORTS a druhý krát cez INCLUDES/EXTENDS). Aby sme predišli takejto situácii, vyberieme všetky množiny a konštanty z A do samostatného stroja[12], ktorý bude videný strojmi A, B aj implementáciou stroja B. Nepretržitosť štruktúry vykazuje špecifikácia v príklade 4.2.2 v nasledujúcej kapitole (pozri obr. 4.5 b)).

3.7 Formalizácia diagramatických modelov

Aj keď je možné priamo, na základe analýzy požiadaviek, zostaviť formálnu špecifikáciu systému, v praxi sa často najprv vytvorí analytický diagramatický model, pozostávajúci z množiny diagramov, popisujúcich statické, dynamické a funkčné vlastnosti systému.

V tejto časti uvedieme postupy transformácie statických a dynamických modelov, ktoré sa využívajú pri objektovo orientovanom návrhu metódou OMT (Object Modeling Technique), prevzaté z [43]. Konkrétne pôjde o objektový diagram v prípade statického údajového modelu a o stavový diagram v prípade dynamického modelu. Podobné postupy sa používajú aj na transformáciu z diagramov jazyka UML [42].

3.7.1 Formalizácia statického údajového modelu

Objektový diagram pozostáva z pomenovaných obdĺžnikov, reprezentujúcich jednotlivé entity – triedy objektov. Každá trieda má uvedené

[12]Na obrázku 3.5 c) je takýto stroj označený "cdata". Jeho implementácia nie je znázornená z priestorových dôvodov.

meno a zoznam atribútov spolu s ich typmi. Vzťahy medzi triedami sú znázornené čiarami spájajúcimi príslušné obdĺžniky.

Proces mapovania objektového modelu do B-jazyka pozostáva z týchto krokov:

1. Identifikácia rodiny typov entít v údajovom modeli. Rodina typov entít je taká skupina typov, ktoré sú podtypmi nejakého typu T, ktorý nemá žiaden nadradený typ.

2. Identifikácia prístupových ciest medzi rodinami entít, potrebných pre ich operácie a atribúty.

3. Vypracovanie orientovaného acyklického grafu na základe identifikovaných ciest, ktorého vrcholmi sú rodiny a hrany vyjadrujú vzťahy typu USES a SEES medzi rodinami.

4. Definovanie abstraktných strojov pre každú rodinu podľa postupu popísaného nižšie a zahrnutie strojov do iných strojov podľa vzťahov identifikovaných v kroku 3.

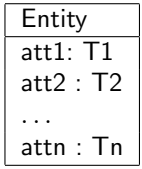
Postup definovania abstraktného stroja pre jednoduchý prípad entity (triedy) bez podtypov (zdedených tried), vrátane operácie vytvárajúcej novú inštanciu entity, je na obrázku 3.6.

Entitu (triedu) *Entity* v B teda vyjadríme ako abstraktný stroj, ktorý v sebe zahŕňa množiny všetkých možných inštancií (množina identity objektov $ENTITY$) a všetkých existujúcich inštancií triedy (množina *entities*). Operácia vytvárajúca novú inštanciu entity (createEntity) vyberie ľubovoľnú ešte nevybranú (neinicializovanú) inštanciu z $ENTITY$, priradí jej hodnoty atribútov a zaradí ju do množiny *entities*.

Ak sú medzi entitami v modeli väzby (vzťahy), využívame na sprístupnenie príslušných strojov mechanizmy SEES alebo USES. SEES v prípade, že iba potrebujeme použiť množinu identity objektov (ENTITY) jedného stroja v invariante druhého. Ak potrebujeme v invariante či operáciách použiť aj množinu existujúcich objektov (entities) iného stroja, použijeme USES. Dedičnosť modelujeme mechanizmom EXTENDS, agregáciu mechanizmom INCLUDES.

3.7.2 Formalizácia dynamického modelu

Stavový diagram pozostáva z vrcholov, predstavujúcich stavy systému, ktoré sú poprepájané hranami, znázorňujúcimi stavové prechody. Pre-



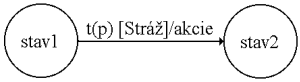
a)

MACHINE *Entity*
SETS $ENTITY$
VARIABLES $entities, att1, att2, \ldots, attn$
INVARIANT $entities \subseteq ENTITY \wedge att1 \in entities \rightarrow T1$
$\wedge \ldots \wedge attn \in entities \rightarrow Tn \wedge \ldots$
INITIALISATION $entities := \varnothing \parallel att1 := \varnothing \parallel \ldots \parallel attn := \varnothing$
OPERATIONS
ee ← createEntity(av1, ..., avn) $\widehat{=}$
PRE $av1 \in T1 \wedge \ldots \wedge avn \in Tn \wedge entities \neq ENTITY$
THEN ANY oo WHERE $oo \in ENTITY - entities$ THEN
$ee := oo \parallel entities := entities \cup \{oo\} \parallel$
$att1(oo) := av1 \parallel \ \ldots \ \parallel attn(oo) := avn$
END
END
END

b)

Obr. 3.6: Trieda *Entity* v objektovom diagrame (a) a zodpovedajúci abstraktný stroj (b)

chodom je pridelená strážna podmienka a (voliteľne) tiež akcie, ktoré sa vykonajú pri jeho realizácii.



a)

t(e,p) $\widehat{=}$
PRE Stráž $\wedge$ status_flag(e)=stav1 $\wedge$ $p \in T \wedge e \in entities$ THEN
status_flag(e):=stav2 $\parallel$ ... akcie prechodu ...
END

b)

Obr. 3.7: Fragment stavového diagramu (a) a prepis stavového prechodu $t(p)$ do operácie v B-jazyku (b)

Proces transformácie stavového diagramu do B-jazyka, vhodný aj pre veľký počet stavov, je nasledovný:

1. Vytvoriť abstraktný stroj *Entity* pre každú rodinu typov entít v údajovom modeli (ako v predchádzajúcom postupe) a premennú $status\_flag_i$, definovanú ako funkciu $status\_flag_i : \ entities \rightarrow STATUS\_SET_i$. Táto premenná bude zaznamenávať stav i-teho faktora[13] stavového diagramu pre *Entity*. $STATUS\_SET_i$ je množina (reprezentantov) stavov v i-tom faktore.[14]

2. Vytvoriť operáciu stroja *Entity* pre každý prechod a aktivitu v každom stavovom diagrame, ktorá vykoná akcie prechodu a zmení stav zmenou hodnoty príslušnej (alebo príslušných, ak sa prechod vyskytuje vo viacerých diagramoch) $status\_flag_i$ premennej (premenných). Vytvorenie operácie znázorňuje obr. 3.7.
   Operácia vytvorenia novej inštancie entity (createEntity) musí každú z funkcií $status\_flag_i$ inicializovať tak, aby jej hodnota reprezentovala počiatočný stav daného faktora diagramu.

3. Vyjadrenie synchronizácie medzi subsystémami operáciami, ktoré volajú operácie strojov reprezentujúcich subsystémy a to s použitím operátora "$\|$".

[13] Faktor H grafu G je taký podgraf G, ktorý má množinu vrcholov zhodnú s G (formálne $V(H) = V(G)$).

[14] V jednoduchších prípadoch, keď máme len jeden diagram, si vystačíme s jedinou množinou (reprezentantov) stavov $STATUS\_SET$ a teda aj s jedinou premennou $status\_flag : \ entities \rightarrow STATUS\_SET$.

Kapitola 4

Virtuálne prostredie pre formálne vyvinutý softvér

Súčasný trend masifikácie vysokoškolského vzdelávania, ktorý sa aj v našom školstve v posledných rokoch výrazne prejavuje, vyžaduje venovať zvýšenú pozornosť tomu ako motivovať študentov aby si tú či onú problematiku osvojili a získané poznatky efektívne preniesli do praxe. V rámci výučby softvérového inžinierstva je správna motivácia obzvlášť dôležitá práve v predmetoch venovaných formálnym metódam (FM), najmä tým ťažkým, medzi ktoré patrí aj B-metóda. Tie sú totiž, na rozdiel od bežných techník ako je návrh s využitím diagramov UML či objektovo-orientované programovanie, v praxi používané iba niekoľkými spoločnosťami na svete a aj to len v špecifických oblastiach, kde je ich zvýšená náročnosť odôvodnená. Samozrejme, ako akademici uvedomujúci si pozitívny vplyv formálnych metód na spoľahlivosť systémov, by sme ich radi videli nasadené čo najčastejšie. Tu možno súhlasiť s [9], že spôsob ako to dosiahnuť je vychovať rigoróznych softvérových inžinierov, ktorí ovplyvnia spôsob akým je softvér v praxi vyvíjaný v prospech formálnych metód. Je našim presvedčením, že kritickým z hľadiska motivácie je správny výber problémov, ktoré budú počas výučby študentmi pomocou FM riešené a príkladov na ktorých bude ich použitie demonštrované. Ako uvádzajú [47, 51] tieto príklady

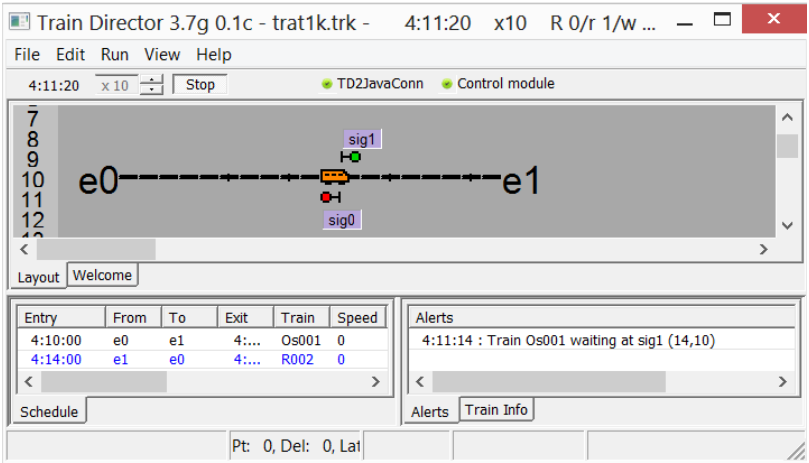
by mali jasne ukazovať prínosy FM, konkrétne čo je možné dosiahnuť len použitím FM a nie iných techník. Vhodné je použiť príklady odvodené z priemyselnej praxe [44]. Podľa štúdií [56, 13] nájdeme najviac prípadov úspešného praktického nasadenia FM v oblastiach hromadnej dopravy, najmä železničnej, financií a obrany. Ideálne by teda bolo priblížiť študentom FM na príkladoch a problémoch z niektorej z nich. Tu však narážame na ich fyzickú nedostupnosť pre potreby výučby. Z tohoto dôvodu sme sa rozhodli využiť simulačnú počítačovú hru ako virtuálnu reprezentáciu vybranej oblasti, konkrétne železničnej dopravy. Upravili a o ďalšie softvérové vybavenie sme doplnili existujúci železničný simulátor Train Director [68] tak, aby umožňoval riadenie traťových zariadení, semaforov a výhybiek, samostatným programom. A práve takéto riadiace programy študenti v rámci nášho predmetu vyvíjajú pomocou B-metódy, resp. sú na nich rôzne aspekty B-metódy objasňované. Softvér a riadiace programy, ktoré v tejto časti predstavíme, sú spolu s ďalšími príkladmi dostupné zo stránky [65].

4.1 Train Director a TS2JavaConn ako virtuálna železnica

Pre použitie železničného simulátora sme sa rozhodli z dôvodu, že so železničnou dopravou má takmer každý študent skúsenosť, vo svete už existujú plne automatizované trate a nie je ťažké si predstaviť katastrofické dôsledky zlyhania ich riadiacich systémov. Boli zvažované viaceré simulátory [39], Train Director (TD) bol vybraný pre jeho dostatočnú funkcionalitu a dostupnosť zdrojových kódov. Okrem úpravy TD bola vyvinutá samostatná aplikácia, nazvaná TS2JavaConn, zabezpečujúca komunikáciu medzi TD a riadiacim programom. Táto virtuálna reprezentácia železnice pracuje nasledujúcim spôsobom: Najprv sa spustia obe aplikácie, TD aj TS2JavaConn a nadviaže sa medzi nimi spojenie (automaticky). Potom sa v simulátore otvorí scenár, pozostávajúci zo železničnej trate a grafikonu vlakovej dopravy, a v TS2JavaConn načíta riadiaci program. Ďalej je možné spustiť simuláciu, kde semafory a výhybky v trati budú na základe požiadaviek vlakov ovládané riadiacim programom.

4.1.1 Train Director

Hra *Train Director* [68] je simulátorom diaľkovo ovládaného zabezpečovacieho zariadenia (centralized traffic control) a pôvodnou úlohou hráča v nej je nastavovať výhybky a semafory v trati tak, aby vlaky premávali podľa grafikonu. Hra má vlastnú logiku, ktorá predchádza kolíziám a automaticky nastavuje niektoré semafory. Tiež má implementované jednoduché rozhranie pre komunikáciu s iným programom, ktorému tak umožňuje ovládať dianie v simulátore emuláciou kliknutí myšou. Aby



Obr. 4.1: Modifikovaný Train Director počas simulácie

poslúžila nášmu účelu, boli v TD vykonané rozsiahle úpravy:

1. Bola eliminovaná vnútorná logika TD, ktorá zabraňovala kolíziám vlakov a automaticky nastavovala niektoré semafory a výhybky. Tým sa otvorila cesta k prevzatiu úplnej kontroly nad traťou externým riadiacim programom.

2. Používateľské rozhranie bolo doplnené tak, aby umožňovalo zadať a zobraziť mená podstatných traťových zariadení, konkrétne výhybiek a semaforov. Tieto mená sa potom používajú aj v riadiacom programe. Do používateľského rozhrania bola tiež pridaná indikácia spojenia s TS2JavaConn a riadiacim programom.

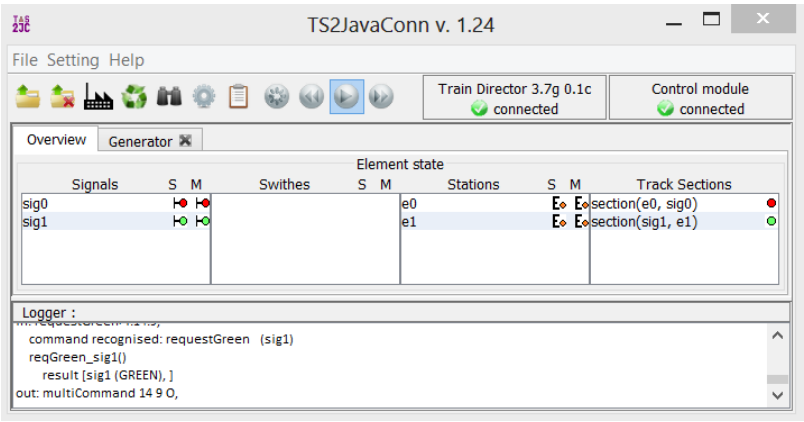
3. Komunikačné rozhranie bolo rozšírené tak, aby umožňovalo výmenu parametrických správ s aplikáciou TS2JavaConn.

Rozšírenie komunikačného rozhrania bolo najrozsiahlejšou a najpodstatnejšou úpravou v TD. Upravený TD posiela správu TS2JavaConn vždy, keď vlak zastane pred semaforom svietiacim na červeno (správa typu requestGreen), keď chce vstúpiť do trate z niektorého vstupného bodu a ten nie je otvorený (requestEnter) alebo opustiť stanicu (requestDepartureStation). Tieto správy obsahujú parametre ako sú meno príslušného semafora, vstupného bodu či stanice, meno vlaku a zoznam staníc, ktoré má vlak podľa grafikonu ešte navštíviť. Vstupný bod je koncový bod trate a má dvojakú interpretáciu. Keď vlak vstupuje do trate cez daný vstupný bod, správa sa tento ako semafor (ak je uzavretý („červený“), vlak vstúpiť nemôže a musí o vstup žiadať správou requestEnter). Keď cezeň odchádza, správa sa ako stanica (vstupný bod je v cestovnom poriadku poslednou zastávkou vlaku). Parametre sú potrebné na to, aby sa riadiaci program vedel kvalifikovane rozhodnúť ako vlaku nastaviť ďalšiu cestu. Aplikácii TS2JavaConn sú ešte zasielané správy typu sectionLeave, a to keď vlak opustí traťový úsek, a sectionEnter keď do úseku vstúpi. Traťové úseky sú vždy ohraničené semaforom, výhybkou alebo vstupným bodom. Správy, ktoré TD od TS2JavaConn prijíma sú príkazmi pre spustenie, zastavenie a reštart simulácie a pre zmenu stavu semaforov a výhybiek.

Vzhľad TD po úprave je možné vidieť na obr. 4.1. Ten zobrazuje simulátor počas simulácie jednoduchého scenára, kde trať pozostáva z jediného priameho úseku s dvoma vstupnými bodmi e0 a e1 a dvoma semaformi sig0 a sig1. Trať je teda tvorená úsekmi e0_sig0 a sig1_e1. Jej riadiaci program je predstavený nižšie, v príklade 4.2.1.

4.1.2 TS2JavaConn

TS2JavaConn (obr. 4.2) je Java aplikácia, ktorá tvorí komunikačné rozhranie medzi TD a riadiacim programom. Aj keď bolo možné toto rozhranie implementovať priamo v TD, bola zvolená samostatná aplikácia a to z dvoch dôvodov. Po prvé, riadiace programy sú tiež Java aplikácie a ich priame prepojenie s TD by bolo problematické. Po druhé, takto je možné relatívne jednoducho TD nahradiť iným simulátorom kde stačí implementovať analogické komunikačné rutiny. Java bola ako jazyk pre riadiace programy zvolená preto, lebo ho podporujú nástroje pre väčšinu súčasných formálnych metód pre vývoj softvéru. V nástroji



Obr. 4.2: Aplikácia TS2JavaConn počas simulácie

TS2JavaConn je možné nahrať riadiaci program (prvé tlačidlo na paneli nástrojov na obr. 4.2), zrušiť riadiaci program (2. tlačidlo), otvoriť kartu s generátorom riadiacich programov (3. tlačidlo), reštartovať spojenie so simulátorom (4. tlačidlo) či ovládať chod simulácie v TD (okrúhle tlačidlá). V časti „Element State“ karty „Overview“ môže používateľ sledovať aký je stav semaforov, výhybiek a úsekov v simulátore (S) a riadiacom programe (M), časť „Logger“ zobrazuje prijaté a odoslané správy. Generátor riadiacich programov umožňuje vytvoriť kostru riadiaceho programu a príslušný konfiguračný súbor. Generátor v súčasnosti podporuje jazyk Java, B-jazyk a jazyk formálnej metódy Perfect Developer [10, 70].

4.1.3 Riadiaci program a jeho vykonanie

Riadiaci program je Java aplikácia, pozostávajúca z jednej alebo viacerých tried. Jej rozsah a zložitosť nie je obmedzená, musí sa v nej však nachádzať jedna *„hlavná“ trieda*, ktorá obsahuje metódy zodpovedajúce správam prijatým od TD a opytovacie metódy, ktoré vracajú hodnoty reprezentujúce stav semaforov, výhybiek a úsekov.

Mapovanie správ na tieto metódy je dané *konfiguračným súborom*. Možnosti konfigurácie sú naozaj široké: Okrem takmer neobmedzenej

voľby názvov metód a označenia hodnôt pre stavy semaforov a výhybiek je možné zvoliť si, či parametre zo správ budú odovzdávané ako súčasť mena metódy alebo ako klasické parametre a aj to, ktoré z nich budú odovzdávané. Napríklad riadiaci program pre scenár z obr. 4.1 by mohol na správu requestGreen pre sig1 reagovať volaním neparametrickej metódy reqGreen_sig1() (tak je to v príklade 4.2.1) alebo parametrickej reqGreen(sig1) či reqGreen(sig1,e1) (parametrický prístup ilustruje príklad 4.2.2). Takéto rozsiahle možnosti nastavenia boli implementované, aby vytvorené riešenie bolo použiteľné pre čo najširšie spektrum nástrojov formálnych metód bez nutnosti dodatočne modifikovať kód nimi generovaný.

Na obrázkoch 4.1 a 4.2 je možné vidieť aj priebeh komunikácie medzi oboma aplikáciami počas simulácie. Obrázky zachytávajú situáciu tesne po žiadosti o zelenú na semafor sig1 od vlaku Os001. V časti „Logger“ možno vidieť, že TS2JavaConn reaguje volaním metódy reqGreen_sig1(). Tá v riadiacom programe zmení hodnotu premennej reprezentujúcej sig1 a táto hodnota sa po volaní príslušnej opytovacej metódy (getSig_sig1()) v simulátore prejaví zmenou sig1 na zelenú.

4.2 Prípadová štúdia: riadiace systémy vyvinuté B-metódou

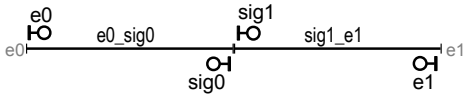
Pre železničné scenáre simulované softvérom opísaným v predchádzajúcej časti je možné (napr. v B-metóde) vytvoriť rôzne riadiace systémy, od jednoduchých s bezparametrickými operáciami, tvorených jedným B-strojom a implementáciou až po sofistikované so zložitými algoritmami a znovu použiteľnými komponentmi. Na podporu tohto tvrdenia teraz uvedieme dva príklady riadiacich programov pre konkrétne trate, vyvinutých pomocou B-metódy. Prvý, v príklade 4.2.1, je prípadom jednoduchého programu s operáciami bez parametrov. Riadi trať zobrazenú na obr. 4.1. Druhý príklad má parametrické operácie a ukazuje znovupoužitie špecifikačných komponentov pre podobné časti trate.

V oboch príkladoch riadiace programy nastavia na začiatku (tzn. po vykonaní inicializácie) všetky semafory a vstupné body na červenú a pri kladnom vybavení žiadosti o zelenú sa táto nastaví iba na danom semafore, tzn. vlak ako keby „skákal“ od jedného semafora k druhému. Dôsledkom toho je, že vlaky môžu uviaznuť keď sa stretnú na hranici dvoch úsekov. Nezrazia sa, ale pokračovať nemôžu. Takéto programy

sme zvolili kvôli zachovaniu dostatočnej jednoduchosti a kompaktnosti príkladov. Samozrejme, nič nebráni napísať riadiaci program tak, aby nastavil naraz viac semaforov a výhybiek a predchádzal uviaznutiam.

V príkladoch budeme hovoriť, že riadiaci program riadi trať (a nie scenár pre TD). Myslíme tým, že riadiaci program je použiteľný s akýmkoľvek scenárom, ktorý pozostáva z danej trate a nejakého grafikonu.

Príklad 4.2.1 (jednoduchý riadiaci program)**.** Tento riadiaci program riadi trať predstavenú už na snímke simulátora Train Director (obr. 4.1). Schéma trate s vyznačením úsekov a vstupných bodov (ako semaforov) je na obr. 4.3. Špecifikácia pozostáva z dvoch komponentov, B-stroja trat1k a jeho implementácie trat1k_i. Keďže sa implementácia od B-stroja líši len zámenou paralelnej kompozície (||) za sekvenčnú (;), uvedieme tu iba B-stroj:



Obr. 4.3: Schéma trate pre príklad 4.2.1

```
MACHINE trat1k
SETS
  PROP_SIGNAL={green, red};
  PROP_SWITCH={switched, none};
  PROP_SECTION={free, occup}
CONCRETE_VARIABLES
  e0, e1, sig1, sig0, e0_sig0, sig1_e1
INVARIANT
  e0:PROP_SIGNAL & e1:PROP_SIGNAL &
  sig1:PROP_SIGNAL & sig0:PROP_SIGNAL &
  e0_sig0:PROP_SECTION & sig1_e1:PROP_SECTION &
  (e0=red or sig0=red) & (e1=red or sig1=red) & //SC1
  ((e0=red   & sig0=red) or e0_sig0=free) &     //SC2
  ((sig1=red &   e1=red) or sig1_e1=free)       //SC3
INITIALISATION
  e0:=red || e1:=red ||    sig1:=red || sig0:=red ||
  e0_sig0:= free || sig1_e1:= free
```

```
OPERATIONS
  ss <-- getSig_sig0 = BEGIN ss:=sig0 END;
  ss <-- getSig_sig1 = BEGIN ss:=sig1 END;
  ss <-- getSig_e0 = BEGIN ss:=e0 END;
  ss <-- getSig_e1 = BEGIN ss:=e1 END;

  reqGreen_e0= IF sig0=red & e0_sig0=free
               THEN e0:=green END;
  reqGreen_e1= IF sig1=red & sig1_e1=free
               THEN e1:=green END;
  reqGreen_sig0= IF e0=red & e0_sig0=free
                 THEN sig0:=green END;
  reqGreen_sig1= IF e1=red & sig1_e1=free
                 THEN sig1:=green END;

  enterNI_e0_sig0=
    BEGIN e0_sig0:=occup || e0:=red || sig0:=red END;
  enterIN_sig1_e1=
    BEGIN sig1_e1:=occup || sig1:=red || e1:=red END;
  enterNI_e1_sig1=
    BEGIN sig1_e1:=occup || sig1:=red || e1:=red END;
  enterIN_sig0_e0=
    BEGIN e0_sig0:=occup || e0:=red || sig0:=red END;

  leaveNI_e0_sig0 = BEGIN e0_sig0:=free END;
  leaveIN_sig1_e1 = BEGIN sig1_e1:=free END;
  leaveNI_e1_sig1 = BEGIN sig1_e1:=free END;
  leaveIN_sig0_e0 = BEGIN e0_sig0:=free END
END
```

Premenné e0 a e1 reprezentujú vstupné body, sig0 a sig1 semafory a e0_sig0 a sig1_e1 úseky. Ako uvádzajú prvé tri riadky klauzuly INVARIANT, ich hodnoty sú z enumeračných množín definovaných v klauzule SETS. Zvyšné tri riadky klauzuly sú kritické vlastnosti, ktoré riadiaci program musí spĺňať, aby nepripustil kolíziu vlakov:

- Iba jeden zo semaforov strážiacich vstup do úseku môže byť zelený (riadok označený komentárom *SC1*).
- Semafor strážiaci vstup do úseku môže byť zelený iba ak je úsek voľný (riadky *SC2* a *SC3*).

B-stroj obsahuje 16 operácii, z ktorých budú po preklade do Java rovnomenné metódy. Tieto operácie (metódy) sú volané prostredníctvom TS2JavaConn na základe požiadaviek zo scenára simulovaného v TD:

- Opytovacie operácie getSig_sig0 až getSig_e1 sú volané po volaní niektorej z ďalších operácií a na základe hodnôt, ktoré vrátia sa nastavia semafory a vstupné body v trati. Mohli by sme mať aj opytovacie operácie pre úseky, no tie nie sú nutné.
- Operácie reqGreen_e0 a reqGreen_e1 sú volané po prijatí správy requestEnter pre daný vstupný bod. Podobne reqGreen_sig0 a reqGreen_sig1 po prijatí requestGreen pre daný semafor.
- Operácie enterNI_e0_sig0 až enterIN_sig0_e0 sú volané po prijatí správy sectionEnter pre daný úsek. Tu sú zvlášť operácie pre vstup vlaku zľava do prava (prvé dve) a zprava doľava (druhé dve), vo všobecnosti to tak nemusí byť.
- Operácie leaveNI_e0_sig0 až leaveIN_sig0_e0 sú volané po prijatí správy sectionLeave pre daný úsek. Znova sú zvlášť operácie pre každý smer.

Pre zaujímavosť uvedieme aj časť Java kódu, vygenerovaného z trat1k_i prekladačom BKPI [30]:

```
public class trat1k {
  public enum PROP_SIGNAL { green(0), red(1),;
   public final int index;
   PROP_SIGNAL(int index){ this.index = index;}
  }
  public enum PROP_SWITCH     {...}
  public enum PROP_SECTION  {...}

  private trat1k.PROP_SIGNAL e0;
  ...
  private trat1k.PROP_SECTION sig1_e1;

  public trat1k() {
    e0 = PROP_SIGNAL.red;
    ...
    sig1_e1 = PROP_SECTION.free;
  }
```

```
  public trat1k.PROP_SIGNAL getSig_sig1() {
    trat1k.PROP_SIGNAL ss; ss = sig1;
    return ss;
  }

  ...

  public void reqGreen_e0() {
    if ((sig0 == PROP_SIGNAL.red &&
         e0_sig0 == PROP_SECTION.free))
              { e0 = PROP_SIGNAL.green; }
  }

  ...
}
```

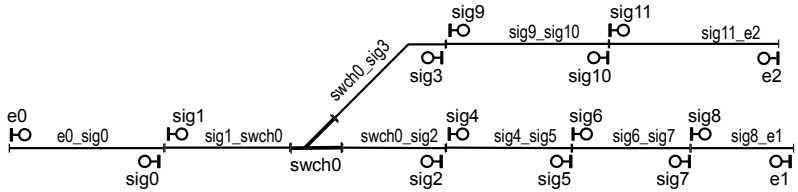
Prekladač BKPI bol taktiež vyvinutý pod vedením autorov a je dostupný zo stránky [65].

□

Riadiaci program z príkladu 4.2.1 je síce dosť jednoduchý, no možno na ňom jasne demonštrovať, čo formálne metódy medzi ktoré patrí aj B-metóda dokážu a čo nedokážu zabezpečiť. Z príkladu je zrejmé, že dokážu preveriť, či kritické vlastnosti uvedené v klauzule INVARIANT naozaj platia v každom stave systému. To je veľmi cenné, keďže odpadá ich testovanie, ktoré je častokrát neúplné. Na druhej strane takáto FM nedokáže overiť, či sme uviedli tie správne vlastnosti (tzn. overiť ich voči neformálnym požiadavkám na riadiaci program) a či je tento program správne prepojený so svojím okolím. Napríklad, ak by sme vymenili názvy operácií reqGreen_sig0() a reqGreen_sig1(), tak by invariantné vlastnosti stále platili, no program by nepracoval správne, keďže by sa pri žiadostiach o zelenú na sig0, resp. sig1 volali nesprávne metódy.

Príklad 4.2.2 (kompozitný riadiaci program)**.** Pre druhý príklad sme vybrali trať na obrázku 4.4. Oproti príkladu 4.2.1 je mierne zložitejšia, obsahuje aj výhybku (swch0) a úseky sú troch typov: vstupný bod-semafor (napr. e0_sig0) semafor-výhybka (napr. sig1_swch0) a semafor-semafor (napr. sig4_sig5).

Riadiaci program by sme mohli riešiť rovnakým spôsobom ako ten v predchádzajúcom príklade, no viedlo by to k príliš veľkému počtu



Obr. 4.4: Schéma trate pre príklad 4.2.2

operácií. Jednak je tu už relatívne veľa semaforov, jednak musíme pri rozhodovaní brať do úvahy aj cieľovú stanicu vlaku. A to minimálne v prípade žiadosti na semafore sig1, kde je na základe cieľovej stanice nastavovaná výhybka swch0. Z týchto dôvodov sme sa rozhodli operácie riešiť ako parametrické a to nasledovne:

- Opytovacie operácie[1] na stav semaforov, úsekov[2] a výhybiek budú getSig, getSec a getSwch, ich parametrom bude meno daného zariadenia, resp. úseku podľa obr. 4.4.
- Na správy typu requestGreen a requestEnter sa bude reagovať volaním operácie reqGreen, kde prvým parametrom (sg) bude príslušný semafor a druhým (st) cieľová „stanica“ (vstupný bod) žiadajúceho vlaku.
- Operácia enter bude volaná pri vstupe vlaku do úseku (tzn. prijatí správy typu sectionEnter) a to z oboch smerov. Parametrom je názov úseku.
- Pri opustení úseku vlakom v smere zľava doprava bude volaná operácia leaveR, pri odchode opačným smerom operácia leaveL. Parametrom je v oboch názov úseku.

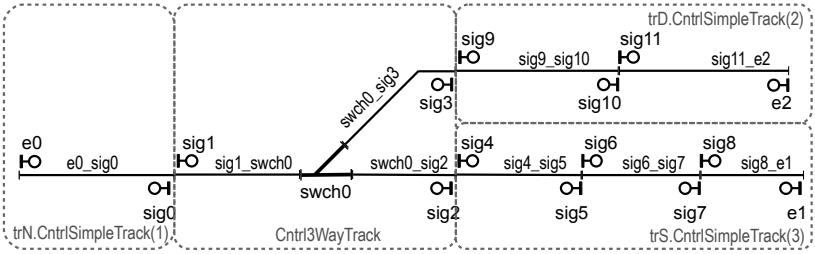
Otázka teraz znie, či riadiaci program riešiť ako jediný B-stroj alebo ho dekomponovať na viaceré stroje, podľa možnosti znovu použiteľné. Pri bližšom skúmaní trate zistíme, že takúto dekompozíciu umožňuje; pozostáva totiž z dvoch typov podúsekov:

[1] O operácie ide v špecifikácii v jazyku B-metódy, vo finálnom riadiacom programe sú to, samozrejme, rovnomenné metódy.

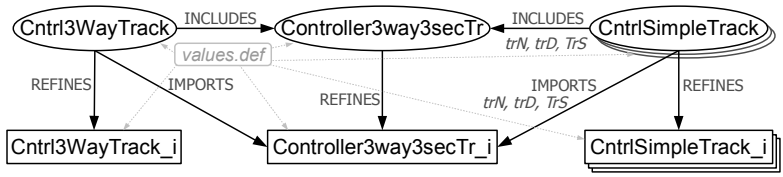
[2] V tomto príklade budeme mať aj tieto aj keď, rovnako ako v predchádzajúcom, nie sú nutné.

1. úsek s výhybkou, chránený semaformi sig1, sig2 a sig3 a
2. rovná trať, zložená z atomických úsekov chránených vždy dvojicou semaforov, pričom ich dĺžka (tzn. počet atomických úsekov) môže byť rôzna.

Úseky druhého typu sú v trati tri, o dĺžke 1, 2 a 3. Pre riadenie úseku prvého typu navrhneme B-stroj Cntrl3WayTrack a pre druhý typ stroj CntrlSimpleTrack, ktorého parametrom bude dĺžka úseku. Stroje oboch typov budú potom vložené do stroja Controller3way3secTr, v ktorom bude definovaných vyššie uvedených sedem operácií. Komponenty riadiaceho programu a ich pokrytie trate znázorňuje obr. 4.5.



a)



b)

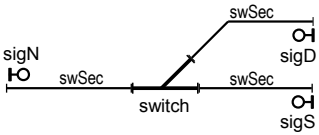
Obr. 4.5: Schéma trate s vyznačením pokrytia úsekov strojmi obsiahnutými strojom Controller3way3secTr (a) a diagram kompozície špecifikačných komponentov celého riadiaceho programu (b)

Predstavovanie samotných B-strojov začneme Cntrl3WayTrack. Ten bude riadiť prístup do úseku prvého typu, semafory, ktoré ho chránia vo všeobecnosti označíme ako sigN, sigS a sigD (obr. 4.6). Označenie N, S a

D sme zvolili na základe štandardného pomenovania koncov dvojcestnej výhybky v angličtine: Narrow, Straight a Diverging. Riešenie stroja je podobné príkladu 4.2.1, no tu budeme všetky úseky chápať ako jeden (logický) úsek, reprezentovaný premennou swSec. Hodnoty pre stavy semaforov, výhybiek aj úsekov tu budú číselné (0, 1), na zabezpečenie lepšej čitateľnosti pre ne však v súbore values.def zadefinujeme alternatívne názvy:

```
DEFINITIONS red==0; green==1; diverg==0; straight==1;
                    free==0; occup==1
```

Hodnota diverg tu znamená nastavenie výhybky do strany (na obr. 4.6 k sigD), straight rovno (k sigS).



Obr. 4.6: Všeobecná schéma trate pre B-stroj Cntrl3WayTrack

Špecifikácia stroja Cntrl3WayTrack je nasledovná:

```
MACHINE Cntrl3WayTrack
DEFINITIONS "values.def"
CONCRETE_VARIABLES sigN, sigS, sigD, swSec, switch
INVARIANT
 sigN:0..1 & sigS:0..1 & sigD:0..1 &                    //3TP1
 swSec:0..1 & switch:0..1                               //3TP2
 &
 (sigN=green =>(sigD=red & sigS=red))  &                //3SC1
 (sigD=green =>(sigN=red & sigS=red))                   //3SC2
 (sigS=green =>(sigD=red & sigN=red))  &                //3SC3
 (sigN=green =>swSec=free) &                            //3SC4
 (sigD=green =>(swSec=free & switch=diverg)) &          //3SC5
 (sigS=green =>(swSec=free & switch=straight))          //3SC6
INITIALISATION
  sigN:=red || sigS:=red || sigD:=red ||
  swSec:=free || switch := straight

OPERATIONS
```

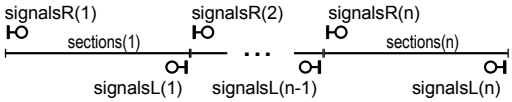
```
  res<—getSigN = BEGIN res:=sigN END;
  res<—getSigS = BEGIN res:=sigS END;
  res<—getSigD = BEGIN res:=sigD END;
  res<—getSwitch=BEGIN res:=switch END;
  res<—get3wSec =BEGIN res:=swSec END;

  reqGreenN(dir)= PRE dir:0..1 THEN
    IF sigS=red & sigD=red & swSec=free
    THEN sigN:=green || switch:=dir END
  END;

  reqGreenS= IF sigN=red & sigD=red & swSec=free
      THEN sigS:=green || switch:=straight
  END;

  reqGreenD= IF sigN=red & sigS=red & swSec=free
      THEN sigD:=green || switch:=diverg
  END;

  enter3w= BEGIN
   sigN:=red || sigS:=red || sigD:=red || swSec:=occup
  END;

  leave3wAll = BEGIN swSec:=free END
END
```

V invariante stroja Cntrl3WayTrack prvé dva riadky (označené komentármi *3TP1* a *3TP2*) definujú typy premenných, zvyšné sú kritické vlastnosti. Riadky *3CSC1* až *3SC3* požadujú aby iba jeden z troch semaforov mohol byť naraz zelený a *3CSC4* až *3SC6* aby mohol byť zelený iba ak je celý priestor medzi semaformi voľný. *3CSC5* a *3SC6* navyše požadujú správne nastavenie výhybky pri povolení vstupu sprava. Parameter dir operácie volanej pri žiadosti o zelenú na sigN (reqGreenN) určuje smer do ktorého nastaviť výhybku. Nastavenie výhybky môže byť súčasťou inej operácie, tu sme zvolili reqGreenN. Slovíčko "All"v názve operácie pre opustenie úseku (leave3wAll) zdôrazňuje, že musí byť volaná iba ak vlak opustí celú oblasť reprezetovanú komponentom, operácia pre vstup (enter3w) môže byť volaná aj keď ide o prechod cez výhybku. Požiadavka na volanie leave3wAll je dôvodom prečo sú v Controller3way3secTr dve operácie pre opustenie úseku (podľa smeru) a iba jedna pre vstup do úseku.

Podobne, ako v predchádzajúcom príklade sa implementácia od stroja líši najmä v zámene „||“ za „;“. V operáciách s parametrami ešte odpadá definícia ich typov (tie sú prevzaté z B-strojov). Napríklad operácia reqGreenN má v implementácii Cntrl3WayTrack_i nasledujúcu podobu:

```
reqGreenN(dir)= BEGIN
  IF sigS=red & sigD=red & swSec=free
  THEN sigN:=green ; switch:=dir END
END;
```

Úsek druhého typu môže byť rôznej dĺžky, preto semafory a atomické úseky v ňom budeme v B-stroji CntrlSimpleTrack definujúcom jeho riadenie reprezentovať poliami signalsR (semafory strážiace vstup do atomických úsekov zľava doprava), signalsL (strážiace vstup sprava doľava) a sections (atomické úseky). Situácia je znázornená na obr 4.7.



Obr. 4.7: Všeobecná schéma trate pre B-stroj CntrlSimpleTrack

V stroji CntrlSimpleTrack sú tieto polia definované ako úplné funkcie (riadky *stTI1* až *stTI3* invariantu). Kritické vlastnosti sú vyjadrené riadkami *stSC1* a *stSC2*. Riadky *stSC1* zodpovedajú *SC1* z príkladu 4.2.1 a *stSC2* zodpovedajú *SC2* a *SC3*. Špecifikácia stroja je nasledovná:

```
MACHINE CntrlSimpleTrack(length)
CONSTRAINTS  length:NAT & length>0
DEFINITIONS "values.def"
CONCRETE_VARIABLES  sections, signalsR, signalsL
INVARIANT
 sections:(1..length)-->(0..1) & //stTI1
 signalsR:(1..length)-->(0..1) & //stTI2
 signalsL:(1..length)-->(0..1)   //stTI3
 &
 !xx.((xx:(1..length)) => ((signalsR(xx)=red)  //stSC1
                        or (signalsL(xx)=red)))
```

```
 &
 !xx.((xx:(1..length)) => ((signalsR(xx)=red & //stSC2
        signalsL(xx)=red) or sections(xx)=free))

INITIALISATION
  sections := (1..length)*{free} ||
  signalsR := (1..length)*{red}  ||
  signalsL := (1..length)*{red}

OPERATIONS
  res<--getSignalR(no) =
       PRE no:1..length THEN res:=signalsR(no) END;
  res<--getSignalL(no) =
       PRE no:1..length THEN res:=signalsL(no) END;
  res<--getSection(no) =
       PRE no:1..length THEN res:=sections(no) END;

  reqGreenR(sigNo)= PRE sigNo:1..length THEN
    IF signalsL(sigNo)=red & sections(sigNo)=free
    THEN signalsR(sigNo):=green END
  END;

  reqGreenL(sigNo)= PRE sigNo:1..length THEN
    IF signalsR(sigNo)=red & sections(sigNo)=free
    THEN signalsL(sigNo):=1 END
  END;

  enterSec(secNo)= PRE secNo:1..length THEN
    sections(secNo):=occup ||
    signalsR(secNo):=red || signalsL(secNo):=red
  END;

  leaveSec(secNo)= PRE secNo:1..length THEN
    sections(secNo):=free
  END
END
```

Najzložitejším B-strojom je „hlavný“ stroj Controller3way3secTr. Ten obsahuje sedem vyššie uvedených operácií, ktoré volajú príslušné operácie v ňom vložených strojov typu Cntrl3WayTrack (jeden) CntrlSimpleTrack (tri, trN, trD a trS). Zodpovednosť vložených strojov za jednotlivé časti trate je zrejmá z obr. 4.5 a).

```
MACHINE Controller3way3secTr
INCLUDES
  Cntrl3WayTrack, trN.CntrlSimpleTrack(1),
  trD.CntrlSimpleTrack(2), trS.CntrlSimpleTrack(3)
SETS
  SIGNALS={e0,sig0,sig1,...,sig10,e2};
  SWITCHES = {swch0};
  SECTIONS = {e0_sig0, sig1_swch0,..., sig11_e2};
  STATIONS = {NE, S_e0, S_e1, S_e2}
DEFINITIONS
  "values.def";
  E0 == trN.signalsR(1); Sig0 == trN.signalsL(1);
  E0_sig0 == trN.sections(1);
  Sig1==sigN; Sig2==sigS; Sig3==sigD; Swch0==switch;
  Sig4==trS.signalsR(1); ... Sig8_e1==trS.sections(3);
  Sig9==trD.signalsR(1); ... Sig11_e2==trD.sections(2)
ABSTRACT_VARIABLES
  trNSigR, trNSigL, trNSc,//pre trN.CntrlSimpleTrack(1)
  trSSigR, trSSigL, trSSc,//pre trS.CntrlSimpleTrack(3)
  trDSigR, trDSigL, trDSc //pre trD.CntrlSimpleTrack(2)
INVARIANT
  //cnTP:
  trNSigR:SIGNALS>+>{1} & trNSigL:SIGNALS>+>{1} &
  trNSc:SECTIONS>+>{1} &
  trSSigR:SIGNALS>+>(1..3) & trSSigL:SIGNALS>+>(1..3) &
  trSSc:SECTIONS>+>(1..3) &
  trDSigR:SIGNALS>+>(1..2) & trDSigL:SIGNALS>+>(1..2) &
  trDSc:SECTIONS>+>(1..2)
  &
  //cnConst:
  trNSigR={e0|->1} & trNSigL={sig0|->1} & ...
  &
  //cnSets:
  dom(trNSigR) \/ dom(trNSigL) \/ {sig1, sig2, sig3} \/
  dom(trSSigR) \/ dom(trSSigL) \/
  dom(trDSigR) \/ dom(trDSigL) = SIGNALS &
  dom(trNSc)\/ {sig1_swch0, swch0_sig2, swch0_sig3}
  \/ dom(trSSc) \/ dom(trDSc) = SECTIONS
  &
  //cnSCtrN:
  (E0=red or Sig0=red)&((E0=red & Sig0=red) or
                                  E0_sig0=free)
```

```
  &
  //cnSC3Way:
  (Sig1=green => (Sig3=red & Sig2=red))  &
  (Sig3=green => (Sig1=red & Sig2=red))  &
  (Sig2=green => (Sig3=red & Sig1=red))  &
  (Sig1=green => swSec=free) &
  (Sig3=green => (swSec=free & Swch0=diverg)) &
  (Sig2=green => (swSec=free & Swch0=straight))
  &
  //cnSCtrS:
  (Sig4=red or Sig5=red) & (Sig6=red or Sig7=red) &
  (Sig8=red or E1=red) &
  ((Sig4=red & Sig5=red) or Sig4_sig5=free) &
  ((Sig6=red & Sig7=red) or Sig6_sig7=free) &
  ((Sig8=red & E1=red) or Sig8_e1=free)
  &
  //cnSCtrD:
  (Sig9=red or Sig10=red) & (Sig11=red or E2=red) &
  ((Sig9=red & Sig10=red) or Sig9_sig10=free) &
  ((Sig11=red & E2=red) or Sig11_e2=free)

INITIALISATION
  trNSigR:={e0|->1} || trNSigL:={sig0|->1} ||
  trNSc:={e0_sig0|->1} ||
  trSSigR:={sig4|->1, sig6|->2, sig8|->3} ||
  trSSigL:={sig5|->1, sig7|->2, e1|->3} ||
  trSSc:={sig4_sig5|->1, sig6_sig7|->2, sig8_e1|->3}||
  trDSigR:={sig9|->1, sig11|->2} ||
  trDSigL:={sig10|->1, e2|->2} ||
  trDSc:={sig9_sig10|->1, sig11_e2|->2}

OPERATIONS
 res <-- getSig(sg)= PRE sg:SIGNALS THEN
   IF sg:dom(trNSigR) THEN
        res:=trN.signalsR(trNSigR(sg))
   ELSIF sg:dom(trNSigL) THEN
        res:=trN.signalsL(trNSigL(sg))
   ELSIF sg:dom(trSSigR) THEN
        res:=trS.signalsR(trSSigR(sg))
   ELSIF sg:dom(trSSigL) THEN
        res:=trS.signalsL(trSSigL(sg))
   ELSIF sg:dom(trDSigR) THEN
```

```
        res:=trD.signalsR(trDSigR(sg))
   ELSIF sg:dom(trDSigL) THEN
        res:=trD.signalsL(trDSigL(sg))
   ELSIF sg=sig1 THEN res:=sigN
   ELSIF sg=sig2 THEN res:=sigS
   ELSE res:=sigD //sg=sig3
   END
END;

res <-- getSec(sc)= PRE sc:SECTIONS THEN
 IF    sc:dom(trNSc) THEN res:=trN.sections(trNSc(sc))
 ELSIF sc:dom(trSSc) THEN res:=trS.sections(trSSc(sc))
 ELSIF sc:dom(trDSc) THEN res:=trD.sections(trDSc(sc))
 ELSE res:=swSec //sig1_swch0, swch0_sig2, swch0_sig3
 END
END;

res <-- getSwch(sw)= PRE sw:SWITCHES
                     THEN res:=switch END;

reqGreen(sg, st)= PRE sg:SIGNALS & st:STATIONS THEN
 IF    sg:dom(trNSigR) THEN trN.reqGreenR(trNSigR(sg))
 ELSIF sg:dom(trNSigL) THEN trN.reqGreenL(trNSigL(sg))
 ELSIF sg:dom(trSSigR) THEN trS.reqGreenR(trSSigR(sg))
 ELSIF sg:dom(trSSigL) THEN trS.reqGreenL(trSSigL(sg))
 ELSIF sg:dom(trDSigR) THEN trD.reqGreenR(trDSigR(sg))
 ELSIF sg:dom(trDSigL) THEN trD.reqGreenL(trDSigL(sg))
 ELSIF sg=sig1 THEN
  IF st=S_e2 THEN reqGreenN(diverg)
  ELSE reqGreenN(straight) END
 ELSIF sg=sig2 THEN reqGreenS
 ELSIF sg=sig3 THEN reqGreenD
 END
END;

enter(sc)= PRE sc:SECTIONS THEN
 IF    sc:dom(trNSc) THEN trN.enterSec(trNSc(sc))
 ELSIF sc:dom(trSSc) THEN trS.enterSec(trSSc(sc))
 ELSIF sc:dom(trDSc) THEN trD.enterSec(trDSc(sc))
 ELSE enter3w END
END;
```

```
 leaveR(sc)= PRE sc:SECTIONS THEN
  IF     sc:dom(trNSc) THEN trN.leaveSec(trNSc(sc))
  ELSIF sc:dom(trSSc) THEN trS.leaveSec(trSSc(sc))
  ELSIF sc:dom(trDSc) THEN trD.leaveSec(trDSc(sc))
  ELSIF sc=sig1_swch0 THEN skip
  ELSE  leave3wAll //sc=swch0_sig2, sc=swch0_sig3
  END
 END;

 leaveL(sc)= PRE sc:SECTIONS THEN
  IF     sc:dom(trNSc) THEN trN.leaveSec(trNSc(sc))
  ELSIF sc:dom(trSSc) THEN trS.leaveSec(trSSc(sc))
  ELSIF sc:dom(trDSc) THEN trD.leaveSec(trDSc(sc))
  ELSIF sc=sig1_swch0 THEN leave3wAll
  END
 END

END
```

Množiny stroja Controller3way3secTr (klauzula SETS) definujú zariadenia a úseky tak, ako sú pomenované v príslušnej trati a sú typmi parametrov jeho operácií. V tomto stroji sa naplno prejavila aj dualita vstupných bodov: Tie nájdeme v množine SIGNALS (napr. e0) ale aj STATIONS (napr. S_e0). Hodnota NE označuje nedefinovanú stanicu.

Dôležitá je tu aj klauzula DEFINITIONS. Tá nielen vkladá súbor values.def ale definuje alternatívne, zmysluplné mená pre prvky z vložených strojov. Tieto mená sú rovnaké ako prvky z množín zo SETS, no s veľkým počiatočným písmenom.

Premenné stroja mapujú hodnoty z množín SECTIONS a SIGNALS na indexy 1 až 3 (pozri riadky *cnTP* invariantu) a používajú sa pri volaní operácií z trN, trD a trS. V podstate to ani nie sú premenné ale konštanty, ich hodnoty sa počas činnosti stroja nemajú meniť. To je vyjadrené časťou *cnConst* v invariante, ktorá obsahuje to isté, čo klauzula INITIALISATION, iba s „||“ zameneným za „&“. Dôvod prečo boli definované ako premenné je, že keď boli konštantami tak sa nepodarilo v nástroji Atelier B dokázať POb stroja. Ide teda o pragmatické rozhodnutie, prispôsobenie sa možnostiam softvéru pre B-metódu. Správne by mali byť konštantami. Časť *cnSets* invariantu hovorí, že domény premenných spolu s menami zariadení a atomických úsekov z úseku s výhybkou majú spolu dať množiny SIGNALS, resp. SECTIONS. Kvôli tejto časti nie je možné POb stroja dokázať v Atelier B úplne automa-

ticky.

Zvyšné časti invariantu, *cnSCtrN* až *cnSCtrD*, sú kritickými vlastnosťami pre celú trať. Sú uvedené postupne, po jednotlivých úsekoch. Inicializácia sa stará o správne priradenie indexov názvom semaforov a úsekov, vložené stroje sú inicializované automaticky.

Všetky operácie majú rovnakú štruktúru: podľa premenných stroja sa zistí, ktorému úseku zodpovedajú hodnoty parametrov a na základe toho sa zavolá príslušná operácia z vložených strojov.

Implementácia hlavného stroja, Controller3way3secTr_i, jeho premenné nahrádza konkrétnymi konštantami SigMap a SecMap. Mierne sa mení aj kód operácií, kde sú teraz v podmienkach explicitne vymenované príslušné hodnoty parametrov. Na ilustráciu uvedieme začiatok implementácie po prvú operáciu (vrátane):

```
IMPLEMENTATION Controller3way3secTr_i
REFINES Controller3way3secTr
IMPORTS
  Cntrl3WayTrack, trN.CntrlSimpleTrack(1),
  trD.CntrlSimpleTrack(2), trS.CntrlSimpleTrack(3)
DEFINITIONS "values.def"
CONSTANTS
  SigMap, SecMap
PROPERTIES
  SigMap:SIGNALS-->1..3 & SecMap:SECTIONS-->1..3
VALUES
  SigMap={e0|->1, sig0|->1,
          sig1|->1, sig2|->1, sig3|->1,
          sig4|->1, sig6|->2, sig8|->3,
          sig5|->1, sig7|->2, e1|->3,
          sig9|->1, sig11|->2, sig10|->1, e2|->2};
  SecMap={e0_sig0|->1,
          sig1_swch0|->1,swch0_sig2|->1,swch0_sig3|->1,
          sig4_sig5|->1, sig6_sig7|->2, sig8_e1|->3,
          sig9_sig10|->1, sig11_e2|->2}

OPERATIONS

 res <-- getSig(sg)= BEGIN
   IF sg=e0 THEN
        res<--trN.getSignalR(SigMap(sg))
   ELSIF sg=sig0 THEN
        res<--trN.getSignalL(SigMap(sg))
```

```
    ELSIF sg=sig4 or sg=sig6 or sg=sig8 THEN
          res<--trS.getSignalR(SigMap(sg))
    ELSIF sg=sig5 or sg=sig7 or sg=e1 THEN
          res<--trS.getSignalL(SigMap(sg))
    ELSIF sg=sig9 or sg=sig11 THEN
          res<--trD.getSignalR(SigMap(sg))
    ELSIF sg=sig10 or sg=e2 THEN
          res<--trD.getSignalL(SigMap(sg))
    ELSIF sg=sig1 THEN res<--getSigN
    ELSIF sg=sig2 THEN res<--getSigS
    ELSE res<--getSigD //sg=sig3
    END
  END;

  ...

END
```

□

Riadiace programy, ktoré sme tu uviedli sú v jazyku B-metódy. V rámci experimentálneho overenia použiteľnosti TD a TS2JavaConn bol však realizovaný vývoj aj pomocou iných formálnych metód. Konkrétne boli pre viaceré trate vyvinuté riadiace programy v metóde Perfect Developer (PD) [10, 70] a pre trať z príkladu 4.2.1 aj v metódach Event-B [2] a VDM++ [14]. Tieto riadiace programy mali bezparametrické operácie, resp. metódy. Na generovanie Java kódu bol v prípade PD použitý nástroj Escher Verification Studio [70], pre VDM++ nástroj VDM++ Toolkit [71] a pre Event-B prekladač EB2J [66]. Získaný Java kód po kompilácií pracoval s TD a TS2JavaConn bez problémov, iba vo výstupe z EB2J bolo potrebné doplniť explicitný konštruktor.

4.3 Použitie vo výučbe

V rámci výučby formálnych metód je vytvorené softvérové riešenie možné použiť jednak na prednáškach, jednak počas cvičení. Na prednáškach môžu byť všetky aspekty danej metódy ukázané na príkladoch vytvorených ako riadiace programy pre TD. Aj veľmi jednoduchý program (napr. príklad 4.2.1) je vhodný na demonštráciu tak výhod ako aj nevýhod formálnych metód. Pokročilé koncepty ako zjemňovanie či tvorba kompozitných špecifikácií so znovu použiteľnými komponentmi

sú tu taktiež prezentovateľné. Napríklad v abstraktnej špecifikácii riadiaceho programu môžu premenné presne zodpovedať traťovým zariadeniam, zatiaľ čo v jej zjemnení môže byť použitá efektívnejšia, no menej intuitívna údajová reprezentácia. Pri kompozitných špecifikáciách sa program môže „vyskladať“ z modulov pre typizované celky tratí (ako v príklade 4.2.2).

Na cvičeniach TD a TS2JavaConn plnia úlohu virtuálneho laboratória, kde študenti vypracúvajú svoje zadania. V našich podmienkach tento proces prebieha nasledovne: Najprv učiteľ vytvorí scenár a predloží ho študentovi s úlohou preň vypracovať korektný kontrolný program. Študent sa následne so scenárom oboznámi v simulátore TD a nechá si pomocou TS2JavaConn vygenerovať kostru špecifikácie riadiaceho programu v jazyku B-metódy. Špecifikáciu doplní o podmienky, ktoré majú platiť počas celej činnosti programu (invariant) a o telá operácií, dokáže že stanovené podmienky naozaj budú stále platiť (POb) a pomocou procesu zjemňovania prevedie špecifikáciu do implementovateľnej podoby, ktorú potom preloží do jazyka Java, skompiluje, načíta v TS2JavaConn a spustí so simulátorom TD.

Predstavené riešenie bolo na pracovisku autorov úspešne použité počas dvoch rokov výučby predmetu „Formálne špecifikácie systémov“. Predmet sa zameriava na obe formálne metódy opísané v tejto publikácii, na Petriho siete aj B-metódu. Petriho siete boli ilustrované abstraktnými modelmi z oblasti synchronizačných problémov a protokolov, zatiaľ čo väčšina príkladov k B-metóde bola vytvorená s použitím TD a TS2JavaConn. Praktické skúsenosti potvrdili naše presvedčenie vyslovené v úvode. V porovnaní s predchádzajúcimi rokmi boli študenti viac zaujatí prácou s B-metódou a aj tí, ktorí neuspeli veľmi dobre v iných teoreticky orientovaných predmetoch bez väčších problémov zvládali vypracovanie zadaní, ktoré zahŕňali použitie vyvinutých nástrojov. Viacerí označili časť predmetu zaoberajúcu sa B-metódou za pútavejšiu ako tú zaoberajúcu sa Petriho sieťami a to napriek tomu, že časť o B-metóde bola náročnejšia. Dôvodom bolo práve preukázanie jej praktického využitia. Až prekvapujúco pozitívne vyzneli záverečné momenty pri vypracovávaní zadaní, keď študenti mohli vidieť svoje riadiace programy skutočne pracovať so simulovanými scenármi. Samozrejme, vyskytli sa aj aspekty vnímané negatívne. V prvom roku to bola najmä nutnosť písať celú špecifikáciu riadiaceho modulu a aj konfiguračný súbor. To bolo v druhom roku eliminované implementáciou generátora do TS2JavaConn. Ako nevýhoda bola vnímaná aj relatívna

zdĺhavosť prechodu od implementovateľnej špecifikácie v B-metóde k riadiacemu programu, načítanému v TS2JavaConn. Ten vyžadoval tri kroky, konkrétne preklad z B-jazyka do Java pomocou prekladača BKPI, preklad Java zdrojového kódu do spustiteľnej podoby a načítanie spustiteľného programu v TS2JavaConn. Namiesto toho by študenti radi videli jediné tlačidlo po stlačení ktorého by všetky tri kroky prebehli automaticky. Proces bol však takto „komplikovane“ nastavený zámerne, aby sa študenti neuchyľovali k príliš častému spúšťaniu programu na overenie správnosti zmien namiesto formálneho overenia pomocou dôkazov.

4.4 Iné prístupy

Nami vyslovené presvedčenie podporujú aj ďalší odborníci pôsobiaci v oblasti výučby formálnych metód. Napríklad práca [51] potvrdzuje náš názor na význam motivácie, a to najmä vzhľadom na súčasný trend masifikácie vzdelávania, a tiež zdôrazňuje, že študenti budú vidieť iba zanedbateľný prínos v použití formálnych metód pri vývoji bežných (nekritických) systémov. A autori práce [7] si cenia význam experimentálnej platformy vhodnej pre použitie formálnych metód, čo je presne to, čím sa naše riešenie snaží byť, až natoľko, že jej existenciu zaradili ako jeden z princípov výučby formálnych metód v softvérovom inžinierstve (princíp 6). Naše riešenie podporuje aj ďalšie z princípov sformulovaných v [7]: umožňuje zamerať sa na jednu formálnu metódu (princíp 1) s podporou silného a v praxi využívaného softvérového nástroja (princípy 3 a 5). Navyše má so železnicou skúsenosť väčšina študentov a cieľový programovací jazyk, Java, je vo výučbe softvérového inžinierstva veľmi rozšírený (princíp 7).

Z hľadiska realizácie a účelu je naše riešenie najviac podobné práci [3], ktorá rozširuje na hry zamerané simulačné prostredie Greenfoot [59]. Účelom Greenfoot je výučba objektovo-orientovaného programovania v Java a v [3] bol rozšírený tak, že umožňuje do aplikácií vyvinutých pre Greenfoot zabudovať stavové stroje, ktoré sú formálnou metódou. Toto riešenie je založené na myšlienke podobnej našej, a teda využiť existujúci simulačný a vizualizačný softvér pre programy vyvinuté pomocou FM. Líši sa však v tom, že v [3] je pevne určená FM (automaty), ale nie tematika hry s ktorou sú prepojené a nástroje pre FM sú používané v smere od konkrétnych, implementovaných, modelov (v Greenfoot) k abstraktným (v nástroji UPPAAL [60]).

Kapitola 5

Záver

Práve ste dočítali monografiu, ktorej cieľom bolo v kompaktnej forme priblížiť tri typy Petriho sietí, P/T siete, farbené a hodnotiace Petriho siete, a B-metódu, formálnu metódu, určenú pre vývoj softvéru. Okrem všeobecných poznatkov o týchto metódach bol priestor venovaný aj dvom pôvodným výsledkom, ktoré boli autormi dosiahnuté, resp. vznikli pod ich vedením.

Prvým bol algoritmus riešenia problému dosiahnuteľnosti pre P/T siete a formalizácia a konkretizácia jeho de/kompozičnej varianty, založenej na delení siete na podsiete so spoločnými prechodmi (časti 2.1.6, 2.1.7). Ďalší rozvoj tohto riešenia by sa mal uberať cestou implementácie de/kompozičného algoritmu v paralelnom výpočtovom prostredí a bližšiemu preskúmaniu javov pozorovaných u niektorých sietí, napríklad toho, že z existencie riešenia inštancií dosiahnuteľnosti pre podsiete, pri rovnakom počte spoločných prechodov v oboch riešeniach, vyplývala existencia riešenia pre celú sieť. Pôvodným výsledkom sú aj hodnotiace Petriho siete [17] a úprava ich definícií, predstavená v časti 2.3.

Druhým výsledkom, skôr výučbového charakteru, bolo virtuálne železničné prostredie tvorené upraveným simulátorom riadenia železničnej premávky Train Director a novo vyvinutým nástrojom TS2JavaConn (kapitola 4). Pre toto prostredie je možné pomocou B-metódy alebo iných formálnych metód vytvárať formálne verifikované riadiace programy. Dva konkrétne príklady takýchto programov boli v príslušnej kapitole aj predstavené. V súčasnosti je vo vývoji modifikácia tohto riešenia, kde je namiesto Train Director-a použitý 3D simulátor Open

Rails (TS2JavaConn je už s ním použiteľný). Open Rails (OR) [67] je kompatibilný so simulátorom Microsoft Train Simulator, pre ktorý existuje množstvo tratí, verne modelujúcich tie skutočné. OR navyše oveľa detailnejšie a realisticky simuluje železničnú premávku ako TD. Preto je naším zámerom pre OR implementovať aj riadenie vlakov programami vyvinutými pomocou FM. V dlhodobejšom horizonte je našou ambíciou posunúť toto riešenie z polohy výučbového prostredia do podoby virtuálneho laboratória pre vývoj reálnych riadiacich systémov železnice.

Výskumná a vývojová aktivita autorov a ich spolupracovníkov v oblasti opísaných formálnych metód však týmito výsledkami nekončí. Všetky tri typy Petriho sietí a B-metódu spája vypracovaná teória [29] prevodov medzi Petriho sieťami a B-metódou, založená na definovaní homomorfizmov zobrazujúcich hodnotiace (a tým aj P/T) siete na B-stroje a B-stroje na farbené Petriho siete. Vďaka prvému zobrazeniu je možné zjemniť systém navrhnutý ako Petriho sieť vývojovým procesom B-metódy do podoby vykonateľnej implementácie [36], či využiť povinné dôkazy na verifikáciu Petriho sietí [34], vďaka druhému analyzovať špecifikácie v B-jazyku prostriedkami Petriho sietí [32]. Ďalšie výsledky spadajú do oblasti simulácie systémov s diskrétnymi udalosťami za účelom analýzy výkonnosti. Tu sme navrhli sadu simulačných modelov [35, 37] pre proces spracovania 3D scény metódou sledovania lúča (ray tracing) [53] v paralelnom prostredí. Modely boli realizované ako farbené Petriho siete s časovaním a použitím stochastických funkcií a boli na nich realizované simulačné experimenty posudzujúce rôzne prístupy k optimalizácii modelovaného procesu. V rámci tejto činnosti bol využitý prevod P/T siete na B-stroj [34] a vyvinuté dve verzie nástroja CPN Assistant [38, 33], umožňujúce naplánovať a realizovať paralelné simulácie farbených Petriho sietí v prostredí počítačového klastra.

Veríme, že vám táto publikácia pomohla nahliadnuť do sveta formálnych metód či rozšíriť svoje povedomie o ňom. Vzhľadom na čoraz väčšiu závislosť ľudskej spoločnosti na počítačových systémoch je kritické aby sa k ich vývoju pristupovalo seriózne a disciplinovane. A práve formálne metódy ponúkajú prostriedky pre takýto vývoj.

Na záver by sme radi poďakovali našim kolegom, ktorých názory, usmernenia a rady prispeli k vzniku tejto publikácie, jej finálnej podobe a dosiahnutiu výsledkov v nej popísaných. Naša vďaka patrí Branislavovi Sobotovi, Jaroslavovi Porubänovi, Marekovi Výrostovi, Jánovi

Genčimu, obom recenzentom a ďalším. Zo študentov, ktorí sa podieľali na vývoji s týmito výsledkami spojeným softvérovým vybavením sú to najmä Ján Sorád, Marek Janotka, Ján Marcinčin, Jozef Doboš a Peter Antal.

Zoznam skratiek a typografické konvencie

V publikácii je použitých viacero skratiek, najfrekventovanejšie z nich uvádzame v nasledujúcom zozname. V zátvorke za slovenským významom je u väčšiny uvedený aj anglický.

B – B-metóda (B-method)

CPN – farbená Petriho sieť (Coloured Petri Net)

EvPN – hodnotiaca Petriho sieť (Evaluative Petri Net)

FM – formálna metóda (Formal Method)

GPN – zovšeobecnená Petriho sieť (Generalised Petri Net)

GS – zovšeobecnená substitúcia (Generalized Substitution)

GSL – jazyk zovšeobecnenej substitúcie (Generalized Substitution Language)

PS – Petriho sieť (Petri Net)

P/T – Miestovo/Prechodové (Place/Transition)

RP – problém dosiahnuteľnosti (Reachability Problem)

TD – Train Director

VAS – vektorový adičný systém (Vector Addition System)

Typografické konvencie

Pre štandardný text je použité pätkové písmo typu Times tak, ako v tejto vete. Dôležité pojmy sú zvýraznené *kurzívou*. Vety, definície a príklady začínajú príslušným slovom, nasledovaným číslom odvodeným z číslovania kapitol. Podobne sú číslované tabuľky a obrázky. Príklady, dôkazy a tiež definície a vety, ktorých koniec nie je jednoznačne identifikovateľný sú ukončené symbolom „□“, zarovnaným vpravo. Ďalej sú v publikácii použité nasledujúce typy písma:

- `Neproporcionálne pätkové písmo` je použité v časti 2.2 na označenie deklarácií a prvkov farbených PS v príkladoch 2.2.1 a 2.2.2 a v časti 3.2 na ASCII zápis predikátov a výrazov.
- KAPITÁLKY sú v kapitolách 3 a 4 použité pre kľúčové slová B-jazyka v bežnom texte, tabuľkách a obrázkoch.
- Bezpätkovým písmom sú vytlačené zdrojové texty všetkých špecifikačných komponentov v príkladoch v kapitolách 3 a 4, vrátane kľúčových slov, a prvky týchto komponentov (názvy operácií, premenné, ...) nachádzajúce sa v bežnom texte.

Na odlíšenie od ostatného textu sú zdrojové texty navyše orámované.

Literatúra

[1] Abrial, J.-R.: *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996, ISBN 0-521-49619-5.

[2] Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, prvné vydanie, 2010, ISBN 0521895561, 9780521895569.

[3] Balz, M.; Goedicke, M.: Teaching Programming with Formal Models in Greenfoot. V: *CSEDU (2)*, editovali J. A. M. Cordeiro; B. Shishkov; A. Verbraeck; M. Helfert, INSTICC Press, 2010, ISBN 978-989-674-024-5, s. 309–316.

[4] Bowen, J.; Hinchey, M.: Ten commandments of formal methods... ten years later. *Computer*, ročník 39, č. 1, Jan 2006: s. 40–48, ISSN 0018-9162, doi:10.1109/MC.2006.35.

[5] Bowen, J. P.; Hinchey, M. G.: Seven More Myths of Formal Methods. *IEEE Softw.*, ročník 12, č. 4, Júl 1995: s. 34–41, ISSN 0740-7459, doi:10.1109/52.391826.

[6] Bowen, J. P.; Hinchey, M. G.: Ten Commandments of Formal Methods. *IEEE Computer*, ročník 28, č. 4, 1995: s. 56–63.

[7] Cerone, A.; Roggenbach, M.; Schlinglo, B.-H.; ai.: Teaching Formal Methods for Software Engineering – Ten Principles. V: *Proceedings of Fun With Formal Methods, Workshop affiliated with the 25th Int. Conf. on Computer Aided Verification*, 2013.

[8] ClearSy: *B Language Reference Manual.*
URL http://www.tools.clearsy.com/resources/Manrefb_en.pdf

[9] Cristiá, M.: Teaching Formal Methods in a Third World Country: What, Why and How. V: *Proceedings of the 2006 Conference on Teaching Formal Methods: Practice and Experience*, TFM'06, Swinton, UK, UK: British Computer Society, 2006, s. 10–10.

[10] Crocker, D.: Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm. V: *Practical Elements of Safety*, editovali F. Redmill; T. Anderson, Springer London, 2004, ISBN 978-1-85233-800-8, s. 19–41, doi:10.1007/978-0-85729-408-1_2.

[11] Desel, J.; Reisig, W.: Place/trandsition Petri Nets. V: *Lectures on Petri Nets I: Basic Models*, *Lecture Notes in Computer Science*, ročník 1491, editovali W. Reisig; G. Rozenberg, Springer Berlin Heidelberg, 1998, ISBN 978-3-540-65306-6, s. 122–173, doi:10.1007/3-540-65306-6_15.

[12] Dijkstra, E. W.: *A Discipline of Programming.* Prentice-Hall, 1976, ISBN 0-13-215871-X.

[13] Fitzgerald, J.; Bicarregui, J.; Larsen, P.; ai.: Industrial Deployment of Formal Methods: Trends and Challenges. V: *Industrial Deployment of System Engineering Methods*, editovali A. Romanovsky; M. Thomas, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-33169-5, s. 123–143, doi:10.1007/978-3-642-33170-1_10.

[14] Fitzgerald, J.; Larsen, P. G.; Mukherjee, P.; ai.: *Validated Designs for Object–oriented Systems.* Springer, New York, 2005, ISBN 1-85233-881-4.

[15] Genrich, H. J.; Lautenbach, K.: System modelling with high-level Petri nets. *Theoretical Computer Science*, ročník 13, č. 1, 1981: s. 109 – 135, ISSN 0304-3975, doi:http://dx.doi.org/10.1016/0304-3975(81)90113-4, special Issue Semantics of Concurrent Computation.
URL http://www.sciencedirect.com/science/article/pii/0304397581901134

[16] Hall, A.: Seven Myths of Formal Methods. *IEEE Softw.*, ročník 7, č. 5, Sept. 1990: s. 11–19, ISSN 0740-7459, doi:10.1109/52.57887.

[17] Hudák, Š.: Rozšírenia Petriho Sietí (habilitačná práca). 1980.

[18] Hudák, Š.: De/compositional Reachability Analysis. *Journal of Electrical Engineering*, ročník 45, č. 11, 1994: s. 424–431, ISSN 1335-3632.

[19] Hudák, Š.: *Reachability Analysis of Systems Based on Petri Nets*. Košice, Slovenská republika: elfa, 1999, ISBN 80-88964-07-5.

[20] Hudák, Š.: Verification of Systems: Deadlock Analysis Based on Petri Nets. V: *Proceedings of Workshop on Algebraic, Logical, and Algorithmic Methods of System Modeling, Specification and Verification, ICTERI 2012*, Kherson, Ukrajina, 2012, s. 321–343.
URL http://ceur-ws.org/Vol-848/

[21] Jensen, K.: Coloured petri nets and the invariant-method. *Theoretical Computer Science*, ročník 14, č. 3, 1981: s. 317–336, ISSN 0304-3975, doi:http://dx.doi.org/10.1016/0304-3975(81)90049-9.
URL http://www.sciencedirect.com/science/article/pii/0304397581900499

[22] Jensen, K.: High-Level Petri Nets. V: *Applications and Theory of Petri Nets*, *Informatik-Fachberichte*, ročník 66, editovali A. Pagnoni; G. Rozenberg, Springer Berlin Heidelberg, 1983, ISBN 978-3-540-12309-5, s. 166–180, doi:10.1007/978-3-642-69028-0_12.

[23] Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. V: *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, London, UK, UK: Springer-Verlag, 1994, ISBN 3-540-58043-3, s. 230–272.

[24] Jensen, K.: *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Volume 3*. Springer, 1997, ISBN 978-3-642-64556-3, doi:10.1007/978-3-642-60794-3.

[25] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1*. Coloured Petri Nets, Springer, 1997, ISBN 978-3-540-60943-8.

[26] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 2.* Coloured Petri Nets, Springer, 1997, ISBN 978-3-540-58276-2.

[27] Jensen, K.; Kristensen, L. M.: *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009, ISBN 9783642002847.

[28] Johnsonbaugh, R.; Murata, T.: Petri Nets and Marked Graphs-Mathematical Models of Concurrent Computation. *The American Mathematical Monthly*, ročník 89, č. 8, 1982: s. 552–566, ISSN 0002-9890.

[29] Korečko, Š.: *Integrácia Petriho sietí a B-Metódy pre mFDT prostredie.* Dizertačná práca, Technická Univerzita v Košiciach, 2006.

[30] Korečko, Š.; Dancák, M.: Some Aspects of BKPI B Language Compiler Design. *Egyptian Computer Science Journal*, ročník 35, č. 3, 2011: s. 33–43, ISSN 1110-2586.

[31] Korečko, Š.; Hudák, Š.; Doboš, J.; ai.: Decompositional Mw Automaton-Based Reachability Algorithm. *Egyptian Computer Science Journal*, ročník 37, č. 6, 2013: s. 19–32, ISSN 1110-2586.

[32] Korečko, Š.; Hudák, Š.; Šimoňák, S.: Analysis of B-machine based on Petri Nets. V: *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering*, Vysoké Tatry, Slovenská republika, 2008, ISBN 978-80-8086-092-9, s. 24–33.

[33] Korečko, Š.; Marcinčin, J.; Slodičák, V.: CPN Assistant II: A Tool for Management of Networked Simulations. V: *Application and Theory of Petri Nets*, editovali S. Haddad; L. Pomello, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31130-7, s. 408–417, doi:10.1007/978-3-642-31131-4_23.

[34] Korečko, Š.; Sobota, B.: Building Parallel Raytracing Simulation Model with Petri Nets and B-method. V: *Proceedings of Eurosim 2010*, Praha, Česká republika, 2010.

[35] Korečko, Š.; Sobota, B.: Using coloured Petri nets for design of parallel raytracing environment. *Acta Universitatis Sapientiae. Informatica*, ročník 2, č. 1, 2010: s. 28–39, ISSN 1844-6086.

[36] Korečko, Š.; Sobota, B.: Petri Nets to B-Language Transformation in Software Development. *Acta Polytechnica Hungarica*, ročník 11, č. 6, 2014: s. 187–206, ISSN 1785-8860.

[37] Korečko, Š.; Sobota, B.; Janošo, R.: Evaluation of Parallel Raytracing Strategy Improvements by Petri Nets. *Journal of Computer Science and Control Systems*, ročník 3, č. 1, 2010: s. 87–92, ISSN 1844-6043.

[38] Korečko, Š.; Sobota, B.; Szabó, C.: Performance analysis of processes by automated simulation of Coloured Petri nets. V: *Proceedings of 10th International Conference on Intelligent Systems Design and Applications*, Nov 2010, s. 176–181, doi:10.1109/ISDA.2010.5687271.

[39] Korečko, Š.; Sorád, J.; Sobota, B.: An External Control for Railway Traffic Simulation. V: *Proceedings of the Second International Conference on Computer Modelling and Simulation*, Brno, Česká republika, 2011, s. 68–75.

[40] Kosaraju, S. R.: Decidability of Reachability in Vector Addition Systems (Preliminary Version). V: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, New York, NY, USA: ACM, 1982, ISBN 0-89791-070-2, s. 267–281, doi:10.1145/800070.802201.

[41] Kostin, A. E.: A Reachability Algorithm for General Petri Nets Based on Transition Invariants. V: *MFCS*, *Lecture Notes in Computer Science*, ročník 4162, editovali R. Kralovic; P. Urzyczyn, Springer, 2006, ISBN 3-540-37791-3, s. 608–621.

[42] Laleau, R.; Polack, F.: Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development. V: *ZB*, *Lecture Notes in Computer Science*, ročník 2272, editovali D. Bert; J. P. Bowen; M. C. Henson; K. Robinson, Springer, 2002, ISBN 3-540-43166-7, s. 517–534.

[43] Lano, K.: *The B Language and Method: A Guide to Practical Formal Development*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., prvné vydanie, 1996, ISBN 3540760334.

[44] Larsen, P.; Fitzgerald, J.; Riddle, S.: Practice-oriented courses in formal methods using VDM++. *Formal Aspects of Computing*, ročník 21, č. 3, 2009: s. 245–257, ISSN 0934-5043, doi:10.1007/s00165-008-0068-5.

[45] Leroux, J.: The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. *Logical Methods in Computer Science*, ročník 6, č. 3, 2010, doi:10.2168/LMCS-6(3:22)2010.

[46] Leroux, J.: Vector Addition Systems Reachability Problem (A Simpler Solution). V: *Turing-100*, *EPiC Series*, ročník 10, editovali A. Voronkov, 2012, ISSN 2040-557X, s. 214–228.

[47] Liu, S.; Takahashi, K.; Hayashi, T.; ai.: Teaching Formal Methods in the Context of Software Engineering. *SIGCSE Bull.*, ročník 41, č. 2, Jún 2009: s. 17–23, ISSN 0097-8418, doi:10.1145/1595453.1595457.

[48] Mayr, E. W.: An Algorithm for the General Petri Net Reachability Problem. V: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, New York, NY, USA: ACM, 1981, s. 238–246, doi:10.1145/800076.802477.

[49] Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, ročník 77, č. 4, Apríl 1989: s. 541–580, ISSN 0018-9219, doi:10.1109/5.24143.

[50] Petri, C. A.: *Kommunikation mit Automaten*. Dizertačná práca, Universität Hamburg, 1962.

[51] Reed, J. N.; Sinclair, J. E.: Motivating Study of Formal Methods in the Classroom. V: *Teaching Formal Methods*, *Lecture Notes in Computer Science*, ročník 3294, editovali C. Dean; R. Boute, Springer Berlin Heidelberg, 2004, ISBN 978-3-540-23611-5, s. 32–46, doi:10.1007/978-3-540-30472-2_3.

[52] Schneider, S.: *The B-method: An Introduction*. Cornerstones of computing, Palgrave, 2001, ISBN 9780333792841.

[53] Sobota, B.; Straka, M.; Perháč, J.: A visualization in cluster environment. V: *Proceedings of the 3rd International Workshop on*

Grid Computing for Complex Problems - GCCP'2007, Bratislava, Slovenská republika, 2007, s. 68–73.

[54] Tonţ, G.: The Algorithmic Procedure of Petri Net Decomposition. *Acta Electrotehnica*, ročník 46, č. 4, 2005: s. 173–177.

[55] Woodcock, J.; Davies, J.: *Using Z: Specification, Refinement, and Proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996, ISBN 0-13-948472-8.

[56] Woodcock, J.; Larsen, P. G.; Bicarregui, J.; ai.: Formal Methods: Practice and Experience. *ACM Comput. Surv.*, ročník 41, č. 4, Okt. 2009: s. 19:1–19:36, ISSN 0360-0300, doi:10.1145/1592434.1592436.

[57] Yen, H.-C.: Introduction to Petri Net Theory. V: *Recent Advances in Formal Languages and Applications*, *Studies in Computational Intelligence*, ročník 25, editovali Z. Ésik; C. Martín-Vide; V. Mitrana, Springer, 2006, ISBN 978-3-540-33460-6, s. 343–373.

[58] Yen, H.-C.: Path Decomposition and Semilinearity of Petri Nets. *International Journal of Foundations of Computer Science*, ročník 20, č. 04, 2009: s. 581–596, doi:10.1142/S0129054109006759. URL http://www.worldscientific.com/doi/abs/10.1142/S0129054109006759

[59] Web stránka nástroja Greenfoot. 2014.
URL http://www.greenfoot.org/

[60] Web stránka nástroja UPPAAL. 2014.
URL http://www.uppaal.org/

[61] Web stránka nástroja Atelier B. 2014.
URL http://www.atelierb.eu/

[62] Web stránka nástroja CPN Tools. 2014.
URL http://cpntools.org/

[63] Web stránka nástroja TIme petri Net Analyzer. 2014.
URL http://projects.laas.fr/tina

[64] Web stránka Petri Nets World. 2014.
URL http://www.informatik.uni-hamburg.de/TGI/PetriNets/

[65] Web stránka projektu “Aplikácia technológií virtuálnej reality ako inovačného prostriedku pri výučbe formálnych metód”. 2014.
URL https://kega2012.fm.kpi.fei.tuke.sk/

[66] Web stránka sady prekladačov EB2ALL. 2014.
URL http://eb2all.loria.fr/

[67] Web stránka simulátora Open Rails. 2014.
URL http://www.openrails.org/

[68] Web stránka simulátora Train Director. 2014.
URL http://www.backerstreet.com/traindir/trdireng.htm

[69] Web stránka spoločnosti ClearSy. 2014.
URL http://www.clearsy.com/

[70] Web stránka spoločnosti Escher Technologies a jej formálnej metódy Perfect Developer. 2014.
URL http://www.eschertech.com/

[71] Web stránka VDM. 2014.
URL http://www.vdmtools.jp/en/

Register

Štefan Korečko a Štefan Hudák

Formálne metódy pre diskrétne systémy: Petriho siete a B-metóda

prvé vydanie

náklad: 100 ks

rozsah: 140 strán

Technická univerzita v Košiciach
2015

ISBN 978-80-553-1895-0