VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

Doc. Ing. Jaroslav Sklenář, CSc.

NÁSTROJE DISKRÉTNÍ SIMULACE

DISCRETE SIMULATIONS TOOLS

ZKRÁCENÁ VERZE HABILITAČNÍ PRÁCE

BRNO 2004

KLÍČOVÁ SLOVA
diskrétní počítačová simulace, Petriho sítě, časové sítě, systémy hromadné obsluhy, sítě hromadné obsluhy, objektově-orientované programování, HTML, JavaScript

KEY WORDS
discrete computer simulation, Petri nets, time nets, queueing systems, queueing networks, object-oriented programming, HTML, JavaScript

MÍSTO ULOŽENÍ PRÁCE
Vědecké oddělení Fakulty informačních technologií VUT v Brně, Božetěchova 2, 612 66 Brno

© Jaroslav Sklenář, 2004
ISBN 80-214-2590-3
ISSN 1213-418X

OBSAH

PŘEDSTAVENÍ AUTORA ... 4

1 ÚVOD ... 5

2 PETRISIM ... 7

2.1 Petriho sítě ... 7
2.2 Sítě Petrisimu ... 7
2.2.1 Zavedení času ... 8
2.2.2 Zobecněné hrany ... 8
2.2.3 Typy míst a přechodů ... 9
2.3 Vytváření diskrétních simulačních modelů pomocí časových sítí ... 10
2.3.1 Aktivity ... 10
2.3.2 Události ... 10
2.3.3 Procesy ... 11
2.4 Funkce PetriSimu a jeho vývoj ... 12
2.4.1 Původní textové prostředí ... 12
2.4.2 Metodika simulace časovými sítěmi ... 13
2.4.3 Grafické prostředí PetriSimu ... 13
2.4.4 Použití mechanismu dědění ... 14
2.4.5 Vytváření uživatelských modelů ... 15
2.5 Simulace systémů hromadné obsluhy ... 15
2.5.1 Systémy s jednou frontou ... 15
2.5.2 Sítě hromadné obsluhy ... 19

3 JSSIM ... 20

3.1 Motivace ... 20
3.2 Objektově-orientované programování v JavaScriptu ... 21
3.3 Simulátor sítí hromadné obsluhy ... 23
3.4 Řídící program a podpůrné prostředky ... 25
3.5 Diskrétní simulace v JavaScriptu a vazba na HTML ... 27

4 ZÁVĚR ... 30

5 LITERATURA ... 30

5.1 Přehled prací tvořících habilitační práci ... 30
5.2 Vybraná Literatura ... 31

ABSTRACT ... 34

PŘEDSTAVENÍ AUTORA

Jaroslav Sklenář se narodil 9.3.1950 v Brně. V roce 1973 absolvoval s vyznamenáním studium na FE VUT v Brně v oboru Technická kybernetika. V letech 1974 až 1976 pracoval jako systémový programátor v Ústředí pro výpočetní techniku Tesla (ÚVTT) v Brně. Byl hlavním programátorem týmu, který navrhl a vytvořil operační systém pro práci v reálném čase RTOS 3 pro počítače RPP-16.

V roce 1976 nastoupil na řádnou vědeckou aspiranturu na Katedře telekomunikací FE VUT, kde v roce 1979 obhájil kandidátskou práci zaměřenou na aplikaci Obecné teorie systémů při simulaci. Během aspirantury a po ní implementoval v rámci spolupráce s Katedrou aplikované matematiky FE VUT jazyk Simscript na počítače řady Tesla 200. V roce 1978 nastoupil jako odborný asistent na Katedru telekomunikací FE VUT. Během působení na FE VUT učil řadu předmětů orientovaných na využití počítačů v telekomunikacích, na což se specializoval i ve výzkumu. Vedle simulace, která byla jeho hlavní specializací, se zabýval programováním kodeků zabezpečovacích kódů a použitím jazyka SDL (Specification and Description Language) při specifikaci ústředen s programovým řízením. Učil rovněž programování mikropočítačů a kursy programování ve vyšších jazycích. Pro praxi realizoval řadu simulačních studií, většinou se jednalo o komunikační protokoly na různých úrovních a paketové sítě. Byl školitelem desítek diplomových prací a dvou úspěšně obhájených kandidátských prací. V roce 1986 byl na měsíční přednáškové a studijní stáži v Mexiku.

V roce 1989 odjel na expertizu na Maltu, kde v letech 1989 až 1995 učil předmět Computers (Advanced level) na lyceu. Na částečný úvazek učil na maltské univerzitě řadu předmětů na informatických oborech. V roce 1995 nastoupil jako Lecturer na maltskou univerzitu, od roku 1998 je Senior Lecturer. Na nově zřízeném ústavu Department of Statistics and Operations Research zavedl obor Operations Research, kde učí v současné době řadu předmětů na čtyřletém bakalářském studiu (B.Sc. Hons.). Školí rovněž M.Sc. studenty.

Ve výzkumu se dále zabývá simulacemi, byl koordinátorem maltské části výzkumného úkolu MEDWATER „A Decision Support System for Water Management in the Mediterranean Region”. Snaží se přes velkou zátěž výukou pravidelně publikovat, byl v programovém výboru řady evropských a mezinárodních konferencí. Je autorem desítek příspěvků na konferencích, pravidelně publikuje v časopisech. Je členem Councilu ASU (Association of Simula Users) a co-editorem Simula Newsletteru. Je rovněž členem SCS (Society for Modeling and Simulation International). Pravidelně spolupracuje s VUT v Brně na úrovni přednáškových pobytů a ve výzkumu. S pracovníky VUT také publikoval několik společných prací.

Je autorem programových balíčků PetriSim pro vytváření simulačních modelů popsaných Petriho sítěmi a JSSim pro simulaci v JavaScriptu.

1 ÚVOD

Habilitační práce je tvořena souborem vybraných publikací spolu s úvodním textem, který je zařazuje do širšího kontextu vycházejícího z historie diskrétní počítačové simulace (dále jen simulace). Práce se zabývá dvěma hlavními tématy. Prvním je použití Petriho sítí při diskrétní simulaci. Články [A1] až [A5] v podstatě popisují vývoj a aplikace programu PetriSim, který je programovým prostředím pro budování diskrétních simulačních modelů vyjádřených ve formě modifikovaných Petriho sítí. Tímto tématem se zabývala habilitační přednáška. Druhým tématem je implementace diskrétních simulačních modelů v jazyce JavaScript integrovaných v HTML dokumentech. Články [B1] až [B4] popisují objektově-orientované programování v JavaScriptu a další problémy spojené s tvorbou těchto modelů. Výsledkem je simulační balíček nazvaný JSSim, který obsahuje prostředky pro psaní simulačních modelů v JavaScriptu. Tato práce obsahuje zkrácený úvodní text habilitační práce doplněný o základní myšlenky výše uvedených článků tak, aby vznikl logický celek.

Existuje mnoho neformálních i formálních definic simulace. Jednou z prvních a velmi výstižných je definice Dahlova [2]: „Simulace je výzkumná metoda, jejíž podstata spočívá v tom, že zkoumaný dynamický systém nahradíme jeho modelem a s ním provádíme pokusy s cílem získat informaci o původním systému“. Potřeba simulace je dána známou skutečností, že pro mnoho problémů, které je vzhledem k jejich rozsáhlosti a složitosti nutné řešit pomocí počítače, není k dispozici matematický model, který by umožňoval přímý výpočet. Autoři simulačních modelů si brzy uvědomili, že psát simulační modely v obecných programovacích jazycích je příliš složité. Je to dáno zejména tím, že simulační model má ve srovnání s jinými programy jednu dimenzi navíc kterou je čas (předpokládám že čtenář je obeznámen se základními principy diskrétní simulace[1] orientované na události resp. procesy). Jelikož se jedná o problém, který je nutné řešit v každém simulačním modelu, vznikl zde prostor pro vytvoření simulačních jazyků a nástrojů, které práci s časem, a také řadu jiných prostředků, již obsahují buďto ve formě příkazů nebo ve formě knihovních procedur.

Simulační jazyky velmi ulehčily programování simulačních modelů. Vystoupil tak do popředí jiný problém - metodika simulace. Bez zacházení do detailů jde o to, že simulace je v podstatě dvoustupňová. Nejdříve vymezujeme tzv. simulovaný systém, který je určitým popisem té části objektivní reality, která nás zajímá[2]. Další fází je pak vytvoření simulačního modelu - programu. Téměř současně s vývojem simulačních jazyků proto začal vývoj prostředků určených k popisu simulovaných systémů[3]. Výrazových prostředků pro potřeby simulace dnes již existuje velmi mnoho, často se jedná právě o různé modifikace Petriho sítí, viz dále.

Myš, která přišla spolu s osobními počítači, přinesla z hlediska simulace změnu zcela zásadní. Grafické uživatelské rozhraní umožnilo vývoj simulačních prostředí[4] kde je programování velmi potlačeno, popřípadě není nutné vůbec. Jazyky, které byly v podstatě textovým popisem bloků,

---

[1] *Skutečnost, že se zabývám výlučně diskrétní simulací, vyplývá z mého zaměření a neznamená, že bych považoval simulaci spojitou za méně důležitou. Z programátorského hlediska je spojitá simulace v podstatě deklarační popis problému, ať už formou popisu jednotlivých bloků a jejich propojení, nebo výčtem rovnic. Spojité jazyky založené na prvním principu jsou dnes již nahrazeny grafickým rozhraním interaktivních simulačních prostředí.*

[2] *Poznamenejme pro úplnost, že předmětem simulace může být i systém abstraktní nebo dosud neexistující, např. ve fázi projektu.*

[3] *Někteří autoři ponechávají pojem „simulovaný systém“ k označení samotné části objektivní reality, její popis je pak označován jako „simulující systém“. V obou případech se však převádí formální určitým způsobem vyjádřený systém do simulačního programu.*

[4] *Zahraniční prameny [17] používají např. termín „Visual Interactive Modelling System“ (VIMS). Všeobecně akceptovaný termín neznám.*

jejich propojení a parametrů, již nejsou zapotřebí. To se týká většiny jazyků pro spojitou simulaci, jazyka Dynamo pro podporu techniky zvané Systémová dynamika (System Dynamics) - [8] a v oblasti simulace diskrétní jazyka GPSS. Pro spojitou simulaci dnes existují nástroje jako např. Simulink v Matlabu (MathWorks, Inc.), pro Systémovou dynamiku je k dispozici mimo jiné např. Powersim (Powersim Corporation). Jazyk GPSS je pak v podstatě nahrazen prostředky jako Arena [11] nebo Extend (ImagineThat, Inc.). Znamená to konec programování simulačních modelů? Určitě ne. Všechny výše uvedené grafické prostředky jsou velmi lákavé tím, že jejich používání je poměrně jednoduché a práci s menšímy modely lze začít ihned prakticky bez přípravy. To je velmi výhodné např. z hlediska výuky. Na druhé straně při budování rozsáhlejších modelů se rychle začnou projevovat nevýhody těchto prostředků. Především je to jejich velká míra specializace. Např. Extend, který je nabízen jako obecný simulátor, je v podstatě souborem několika specializovaných nástrojů. Každý je reprezentován souborem ikon, které představují typické bloky, které se vyskytují při simulaci v té či oné oblasti, např. integrátor nebo fronta. Je pravda, že mnoho modelů pak lze jednoduše vytvořit malováním myší na obrazovce. Jejich použití při simulaci je stejně pohodlné. Problémy začnou, když narazíme na blok nebo funkci, pro kterou není k dispozici ikona. Částečným řešením je ta skutečnost, že většinou lze programovat v podstatě libovolné uživatelské bloky. To je však již dosti pracné, protože to znamená vazbu na programové prostředí, jehož odstínění od uživatele je právě principem těchto prostředků. Další nevýhodou je statika topologie. Jen velmi těžko se vytváří modely, kde bloky vznikají a zanikají. Částečným řešením je možnost ovlivňovat průchod transakcí sítí na základě jejich atributů nebo systémových proměnných, ale i to znamená často neúměrný nárust složitosti modelu. Společným problémem je také práce s velkými sítěmi, ne všechny prostředky umožňují práci s hierarchickými bloky, které jsou uvnitř sítěmi a to na více úrovních. Často je pak výhodnější model naprogramovat.

Některá grafická simulační prostředí jsou založena na určitém formálním přesně definovaném matematickém jazyce. Potom lze využít příslušných teoretických výsledků např. k analýze modelu a získat tak o něm údaje bez nutnosti simulace. I pro simulaci může být výhodou existence obecného formálního výrazového jazyka. Typickým příkladem jsou prostředí založená na Petriho sítích. Jedním z nich je PetriSim, kterým se zabývá následující kapitola.

Po myši byl dalším výrazným stimulem pro rozvoj simulace Internet a to z několika hledisek. Internet v prvé řadě reprezentuje novou platformu, kterou musely akceptovat všechny operační systémy. Přes určité výhrady tak máme poprvé v historii univerzální jednotné prostředí, což lze využít např. k distribuci simulačních modelů doslova komukoliv, kdo je napojený na Internet. Dále lze poprvé skutečně využít při simulaci paralelismus. Myšlenka paralelní simulace není nová, ale až dosud byly víceprocesorové systémy běžnému uživateli nedostupné. Programovacími jazyky Internetu jsou zejména Java a JavaScript[5]. Pro Javu existuje několik simulačních knihoven. Pokud je mi známo, jediný prostředek pro simulaci v JavaScriptu je dále popsaný JSSim.

Zcela záměrně jsem jako výrazný stimul pro rozvoj simulace neuvedl rozšíření objektově-orientovaného programování koncem 80. a začátkem 90. let, které znamenalo stále ještě doznívající revoluci v programování samočinných počítačů. Je to proto, že simulace v prvé řadě techniky objektového programování (jako např. práci s entitami-objekty) používala od samého počátku. Dále pak samotné objektově-orientované programování bylo vynalezeno při vývoji jazyka Simula[6] koncem 60. let.

---

[5] *Název JavaScript vlastní firma Netscape, tentýž jazyk označuje Microsoft jako JScript. Oba jazyky by měly odpovídat normě ECMA-262 (ISO-16262), proto také existuje nepoužívaný název ECMAScript. Rozdíly sice existují, ale lze se jim vyhnout.*

[6] *Simula (Simple Universal Language) samotná je přísně vzato obecný objektově-orientovaný jazyk bez jakékoliv vazby na simulaci. Teprve tzv. systémová třída Simulation reprezentuje diskrétní procesově orientovaný simulační jazyk. Pokud tedy mluvíme o simulaci v Simule, myslíme tím dvojici Simula + Simulation nebo Simula + nějaká jiná simulační třída.*

2 PETRISIM

PetriSim, jehož jsem autorem, vznikl původně jako jednoduchý grafický editor a simulátor Petriho sítí pro potřeby výuky. Postupně byl obohacován tak, že ve dnešní verzi 4 lze PetriSim označit za prostředí určené k diskrétní simulaci založené na grafickém jazyce Petriho sítí.

2.1 PETRIHO SÍTĚ

V roce 1962 podal Karl Adam Petri doktorskou práci „Kommunikation mit Automaten“, kde představil speciální typ orientovaných grafů, které dnes nazýváme Petriho sítě. Graf je tvořený dvěma typy uzlů - tzv. místy (places) a přechody (transitions), které jsou propojené hranami (arcs). Místa mohou být označena tzv. značkami (tokens). Přechody lze za jistých podmínek provádět (fire), což v základní verzi znamená vyjmutí určitého počtu značek ze vstuních míst a přidání určitého počtu značek do míst výstupních. Počet je dán násobností příslušných hran. Takto je pro graf definována vnitřní dynamika závislá na topologii sítě a označení míst. Teorií a aplikacemi Petriho sítí se zabývá dnes již velmi rozsáhlá a stále se rozvíjející vědní disciplína. Sítě v původní verzi, které se dnes označují jako Place/Transition (Pl/Tr) Nets, měly silné analytické, avšak velmi slabé vyjadřovací (modelovací) schopnosti. To způsobilo vznik velmi mnoha modifikací, které původní definici vždy určitým způsobem obohacují. Tato rozšíření se většinou označují společným názvem High-level Petri Nets, některé mají vlastní název jako např. Coloured Petri Nets. V této práci se nebudu zabývat teorií Petriho sítí a různými jejich modifikacemi, zejména proto, že je zde k dispozici řada úvodních i specializovaných publikací. Předpokládám tedy, že autor je se základními principy Petriho sítí obeznámen. Úvod do této teorie obsahují např. monografie [16] a [20], rozšířeními se zabývají např. články [5], [15] nebo články ve sborníku [18], kde je i několik článků úvodních. Publikací zabývajících se Petriho a z nich odvozenými sítěmi jsou již tisíce, samotná orientace v mnoha typech sítí je náročná. Velmi mnoho dalších odkazů lze získat na webových stránkách jako např. [23]. V češtině je k dispozici práce [1]. Existují také desítky programů které lze označit jako prostředí pro práci s Petriho sítěmi. Stránky [23] obsahují databázi těchto programů, která obsahuje i PetriSim.

Aplikace Petriho sítí, viz např. sborník [19], lze rozdělit do dvou kategorií: analytické a simulační. Analytické aplikace založené na analytických schopnostech Petriho sítí postupují následovně: zkoumaný systém je vyjádřen Petriho sítí, ta je analyzována a podle výsledků pak lze usuzovat o vlastnostech původního systému. Např. je-li síť reprezentující komunikační protokol tzv. živá, znamená to, že protokol neobsahuje deadlock. Jiným příkladem je vlastnost zvaná *k*-omezenost (*k*-boundedness). Znamená, že žádné místo neobsahuje nikdy více než *k* značek. Pokud je Petriho sítí modelován výrobní systém, tato skutečnost může např. znamenat, že žádný dopravní pás nebude nikdy obsahovat více než *k* výrobků, což může být testovanou podmínkou. Největším problémem analýzy Petriho sítí je exploze počtu stavů. I poměrně malé sítě mohou mít tisíce různých označení, což může analýzu středních a velkých sítí i na dnešních počítačích zcela znemožnit. Dále se budu zabývat aplikacemi simulačními na které je PetriSim orientován. Ty lze charakterizovat tak, že simulovaný systém má formu nějaké Petriho sítě. Tu pak simulujeme a výsledky opět interpretujeme vzhledem k původnímu systému. Poznamenejme, že v kontextu Petriho sítí se simulací nazývá i výpočet, který neprobíhá v čase. Simulací je pak samotná posloupnost provádění přechodů, typicky s cílem zkoumat chování sítě nebo dokázat správnost nějakého protokolu.

2.2 SÍTĚ PETRISIMU

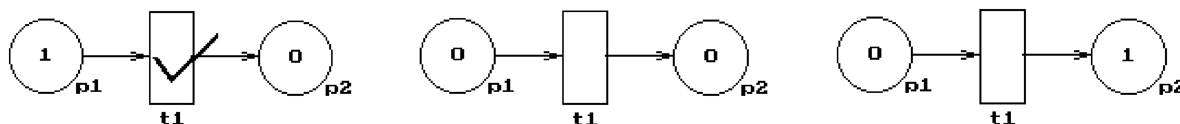
PetriSim pracuje se sítěmi, které obsahují vzhledem k Pl/Tr sítím pouze několik rozšíření, která vyplynula z aplikace těchto sítí při tvorbě diskrétních simulačních modelů. Jedná se v prvé řadě o zavedení času, pak o jistá zobecnění hran a také bylo výhodné definovat nový typ přechodu.

2.2.1 Zavedení času

Pokud má být Petriho síť použita jako grafický jazyk k vyjádření simulačního modelu, je nutné doplnit původní definici o čas. Přístupů existuje několik, souhrnně se pak těmto sítím říká Časové sítě (Timed Nets), ale terminologie není dosud sjednocena. Čas lze definovat ve vztahu k místům, přechodům, hranám i značkám. Viz např. popis v práci [4], který lze spolu s názvy sítí a často používanými zkratkami shrnout následovně.

- Čas ve značkách
  - Coloured Petri Nets (CPN)
- Čas v přechodech
  - Time-out nets
  - Generalized Distributed Timed Transition Stochastic Petri Nets
    - Timed Deterministic Petri Nets
    - Stochastic Petri Nets (SPN)
      - Generalized SPN
      - Phase Type SPN
      - Deterministic SPN ...
- Čas v místech a přechodech
  - Queueing Petri Nets (QPN)
    - Hierarchically Combined QPN

Sítě v PetriSimu jsou založeny na zpoždění v přechodech, což je nejčastější přístup. Mechanismus je tento: provedení přechodu trvá určitou dobu. Při zahájení provedení jsou značky vyjmuty ze vstupních míst, při ukončení provedení jsou značky umístěny do výstupních míst. Viz obr.1, který ukazuje přechod a místa před, během a po provedení přechodu. Na obr.1 je rovněž vidět, jak jsou v PetriSimu označeny přechody, které lze provést (enabled). Všechny obrázky Petriho sítí v této práci byly vytvořeny grafickým editorem PetriSimu. Trvání provedení definuje uživatel, může být konstantní, závislé na stavu sítě nebo náhodné[7]. Při zahájení a ukončení provádění lze rovněž aktivovat kód definovaný uživatelem.



Obr.1 Provádění časového přechodu

2.2.2 Zobecněné hrany

Známým problémem Pl/Tr sítí je nemožnost testovat nulové označení místa. Proto byly zavedeny tzv. inhibiční hrany, které nulové označení testují. PetriSim používá *zobecněné inhibiční hrany*, které mohou mít násobnost větší než 1. Provedení přechodu je pak umožněno tím, že místo obsahuje méně značek než je násobnost hrany, viz obr.2.

---

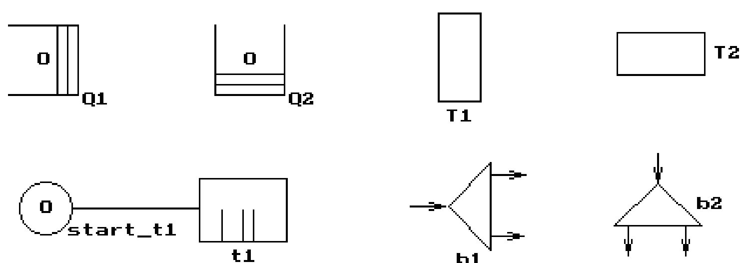
[7] *Tzv. Stochastické Petriho sítě (SPN) s exponenciálním rozdělením trvání provedení přechodů umožňují přímý výpočet parametrů bez potřeby simulace. Lze je reprezentovat Markovským procesem se spojitým časem. Problémem je opět exploze počtu stavů. Jejich použitím se zabývají např. práce [10] a [12].*

Zobecněná testovací hrana pouze testuje, zda místo obsahuje daný počet značek, při provedení přechodu značky nejsou vyjmuty. Pokud je provedení okamžité, pak testovací hrana nahrazuje pár hran obyčejných, viz obr.2. Toto neplatí pro časové přechody, kdy pár obyčejných hran znamená, že značky jsou po dobu provádění dočasně z místa vyjmuty. Z obr.2 je rovněž zřejmé rozdílné zobrazení tří typů hran.

Obr.2 Zobecněné inhibiční a testovací hrany

2.2.3 Typy míst a přechodů

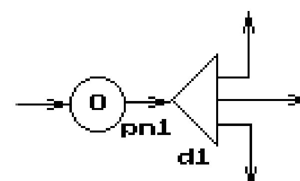
V Pl/Tr sítích existuje pouze jeden typ míst, která jsou kreslena kroužky a jeden typ přechodů kreslených krátkými silnými úsečkami, viz obr.2. V PetriSimu lze místům a přechodům přiřadit tzv. typ, který zatím, až na jedinou výjimku, slouží pouze k vytváření názorných diagramů. Mimo základních typů kroužek a úsečka je zatím pro místa definován typ *fronta* a pro přechody *časový přechod* (obdelník), *generátor značek* a *větvení*. Ikony jsou na obr.3, kde ke generátoru značek je připojeno místo, které ho pomocí testovací hrany může aktivovat.



Obr.3 Typy míst a přechodů

Přechod typu větvení slouží k výběru výstupního místa po provedení přechodu. Značky jsou umístěny pouze do jednoho výstupního místa podle hodnoty vlastnosti *Branch*. Její implicitní hodnota 0 způsobí náhodné umístění do libovolného výstupního místa, všechna se stejnou pravděpodobností. Kód uživatele aktivovaný typicky při ukončení provádění přechodu však může tuto hodnotu změnit a realizovat tak různá náhodná nebo deterministická větvení. Viz obr.4, kde text napsaný kurzívou je generován PetriSimem. Typicky je tento mechanismus použit při řízení průchodu zákazníků sítí hromadné obsluhy.

```
Procedure _PE9; { Ending snippet of t9 (d1) }
Var x : Real;
Begin
  x := Random;
  If x<0.3 then t^.Branch := 1      { 30% up }
  Else if x<0.4 then t^.Branch := 2 { 10% right }
  Else t^.Branch := 3               { 60% down }
End;
```



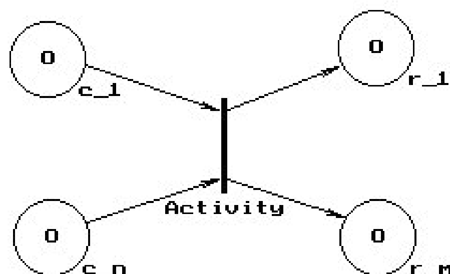
Obr.4 Programováni přechodu větvení

2.3 VYTVÁŘENÍ DISKRÉTNÍCH SIMULAČNÍCH MODELŮ POMOCÍ ČASOVÝCH SÍTÍ

Trvání provedení přechodu lze použít k řízení časové osy u všech tří základních principů časování disktrétních simulačních modelů.

2.3.1 Aktivity

Simulační jazyky založené na aktivitách (Activity Oriented Languages) nepoužívají explicitní plánování příštích událostí. Pro každou činnost (aktivitu) v modelu je zadán soubor podmínek, při jejichž splnění může být aktivita provedena. Použití Petriho sítí při této formalizaci dynamiky je naprosto přirozené. Aktivita = provedení přechodu je podmíněna přítomností značek ve vstupních místech přechodu, které tak modelují splnění daných podmínek - viz. obr.5. Posun v čase je realizován trváním provedení vhodných přechodů, kterými nemusí být nutně přechody modelující jednotlivé aktivity. Dále popisované modely systémů hromadné obsluhy jsou v podstatě všechny založeny na tomto vidění dynamiky.

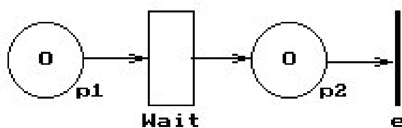


Obr.5 Model podmíněné aktivity Petriho sítí

Metodika modelování při tomto přístupu, který je také označován jako Interrogative Scheduling, tedy spočívá v definici aktivit (přechodů), podmínek jejich provedení (označení vstupních míst) a výsledků těchto provedení (označení výstupních míst). Toto je také základní princip modelování pomocí Petriho sítí.

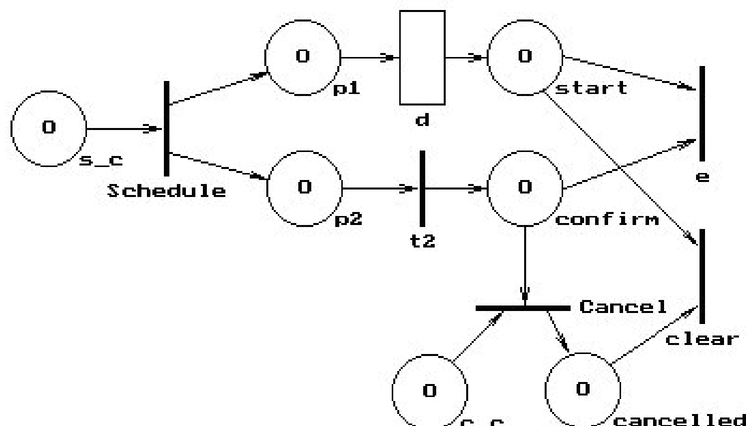
2.3.2 Události

Při simulaci orientované na události řídíme pohyb v čase pomocí dvou primitiv: naplánování výskytu určité události *e* v čase $t_1 \geq t$, kde *t* je okamžitý čas a zrušení výskytu již naplánované události *e*. První z nich se vyskytuje ve dvou variantách: „*schedule e at* t_1“ nebo „*schedule e after d*“, kde *d* je nezáporné zpoždění. Je zřejmé, že první formu lze převést na druhou: „*schedule e after* (t_1-*t*)“ a naopak: „*schedule e at* (*t*+*d*)“. Simulační jazyky orientované na události mají příkazy, které toto plánování provádí přímo. Např. v Simscriptu přesně tak, jak je zde uvedeno. Máme-li k dispozici pouze Petriho síť se zpožděním v přechodech, je nutné naplánování události převést na vytvoření procesu, který provede následující: „*wait d*; *activate e*“ - viz obr.6, kde provedení přechodu *Wait* trvá *d*. Aktivace události = přechodu *e* pak následuje okamžitě.



Obr.6 Plánování události Petriho sítí

Zrušení naplánované události „*cancel e*“ je již složitější, částečné řešení je uvedeno na obr.7, kde přechod *Schedule* zahájí plánování. Jediný časový přechod je *d*, čili značka je do místa *confirm* umístěna okamžitě, do místa *start* až po uplynutí prodlevy. Událost *e* pak následuje pokud obě místa *start* a *confirm* obsahují značku. Zrušení naplánované události provede přechod *Cancel*, který vyjme značku z místa *confirm*. Po uplynutí prodlevy pak přechod *clear* uvede celou síť do původního stavu.



Obr.7 Plánování události s možností zrušení

Jedná se pouze o řešení částečné, další událost totiž lze naplánovat až po uplynutí původní prodlevy, i když minulá událost byla již předtím zrušena. Tento problém nelze vyřešit bez přijetí dalších předpokladů o chování Petriho sítě, protože zatím platí, že jednou zahájené provedení přechodu už nelze přerušit. Tento nedostatek nepovažuji za závažný, protože rušení naplánovaných událostí se provádí poměrně velmi zřídka.

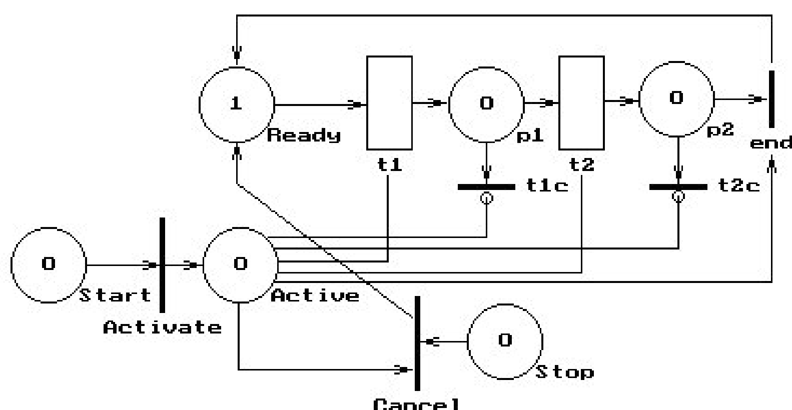
2.3.3 Procesy

Proces je v podstatě posloupnost událostí oddělených prodlevami, které jsou generovány samotným procesem, popřípadě závisí na okamžitém stavu modelu a ostatních procesů. Poměrně primitivní grafický jazyk časových sítí nemůže přímo modelovat všechny sémanticky bohaté formy příkazů pro práci s procesy, které jsou k dispozici v procesově orientovaných simulačních jazycích. Pokusme se přesto naznačit, jak vyjádřit časovou sítí procesově orientované příkazy, resp. metody třídy Simulation jazyka Simula.

P.Hold(*x*) je provedení metody *Hold* procesu *P*, která vygeneruje časovou prodlevu délky *x* v jeho životě. Implementace časovým přechodem je evidentní - viz např. přechod *Wait* na obr.6.

Activate P je nejjednodušší forma příkazu, který aktivuje proces *P*. Řešení je naznačeno na obr.8, kde provedení přechodu *Activate* nastartuje proces, který je tvořen pro jednoduchost pouze přechody *t1* a *t2*. Přítomnost značky v místě *Ready* znamená, že proces může být aktivován. Testovací hrany na obr.8 testují, zda místo *Active* obsahuje značku. Pokud ano, viz deaktivaci procesu dále, přechody *t1* a *t2* realizují proces, přechod *end* ho uvede do počátečního stavu.

Cancel(*R*) je provedení procedury, která deaktivuje (přerušuje provádění) procesu *R*. K přerušení procesu je na obr.8 použit přechod *Cancel*, který také uvede proces do počátečního stavu. Vzhledem k nepřerušitelnosti provádění přechodu se opět jedná o řešení částečné. Proces na obr.8 lze ukončit pouze po provedení přechodů *t1* nebo *t2*. Při přerušení procesu je také nutné vyjmout značky z míst *p1* resp. *p2*. K tomu jsou použity inhibiční hrany, které testují, zda je místo *Active* prázdné. Pokud ano, přechod *t1c* resp. *t2c* tak vyjme značku z místa *p1* resp. *p2*.



Obr.8 Aktivace a ukončení procesu.

2.4 FUNKCE PETRISIMU A JEHO VÝVOJ

V této kapitole je stručně naznačen vývoj PetriSimu a implementace jeho základních funkcí se zvláštním zřetelem na využití objektově-orientovaného programování.

2.4.1 Původní textové prostředí

Předchůdce PetriSimu, nazvaný pracovně PETSIM, byl naprogramován v jazyce Logos [21], který je rozšířením jazyka Logo. Logos byl vytvořen pro výuku objektově-orientovaného programování, práce s třídami je proto velmi názorná a uživatelsky orientovaná. Nevýhodou je samozřejmě primitivnost hostitelského Loga a pouze textová komunikace mezi uživatelem a programem. Logos byl vybrán proto, že v té době nebyl obecně k dispozici přístup k jinému objektově-orientovanému jazyku. Po příchodu Object Pascalu v rámci Turbo Pascalu verze 5.5 byl k implementaci PetriSimu použit Turbo Pascal, PetriSim ve dnešní verzi 4 je naprogramován v Turbo Pascalu resp. Borland Pascalu 7.

Článek [A1] ukazuje, jak reprezentovat znalost o sítích jako takových, místech a přechodech pomocí tříd jazyka Logos. Následuje popis prostředí pro práci s Petriho sítěmi. Pomocí mechanismu dědění (inheritance) pak byly definovány tzv. barvené sítě (Coloured Petri Nets)[8], se krerými už bylo možné provádět jednoduchou simulaci. Význam článku [A1] je v tom, že ukazuje využití technik objektově-orientovaného programování k vytvoření a obohacení jistého prostředí v době, kdy používání těchto technik bylo s výjimkou Simuly teprve v začátcích.

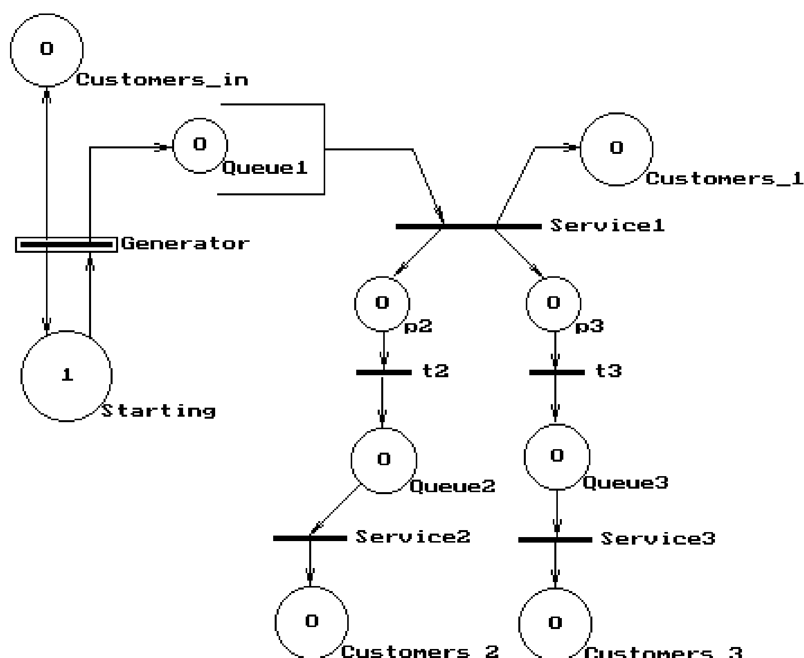
Poznamenejme, že jazyk Logos měl jednu vlastnost, kterou klasické kompilované objektově-orientované jazyky jako např. Simula nebo Java nemají - možnost modifikovat (individualizovat) instance tříd na úrovni kódu. V klasických objektově-orientovaných jazycích jsou instance modifikovány pouze hodnotami atributů, kód (metody) je pro všechny instance shodný. Možnost modifikovat kód instancí je velmi výhodná pro simulaci. Např. přechod časové sítě v PETSIMu měl mimo jiné dvě metody aktivované při zahájení provedení a při ukončení provedení přechodu. Tyto metody pak bylo možné definovat pro každý přechod zvlášť podle potřeby simulačního modelu spolu s metodou, která vracela prodlevu provedení.

---

[8] *Poznamenejme, že se zde jedná o původní jednoduché barvené sítě definované v práci [16]. Pojmem barvené sítě (Coloured Petri Nets) se dnes označují sítě vytvořené na Aarhuzské Universitě v Dánsku, které jsou sítěmi vysoké úrovně, kde značky mohou být libovolné datové struktury, možnost provedení přechodu je testována predikáty a samotné provedení může zahrnovat nejrůznější operace na datech. Viz stránku [27].*

2.4.2 Metodika simulace časovými sítěmi

Článek [A2] shrnuje zkušenosti získané používáním PetriSimu verze 2. Na příkladu sítě hromadné obsluhy tvořené třemi uzly je ukázán celý postup. Síť hromadné obsluhy je nejdříve vyjádřena časovou sítí - viz. obr.9, ke které je pak doplněn kód v Pascalu. Jsou vyjmenovány procedury, které doplňuje uživatel a standardní procedury PetriSimu, které podporují uživatelské programování. Konkrétně se jedná o generování prodlevy při realizaci přechodu, což je základní nástroj práce s časem, získání označení místa a času modelu a možnost modifikace označení.



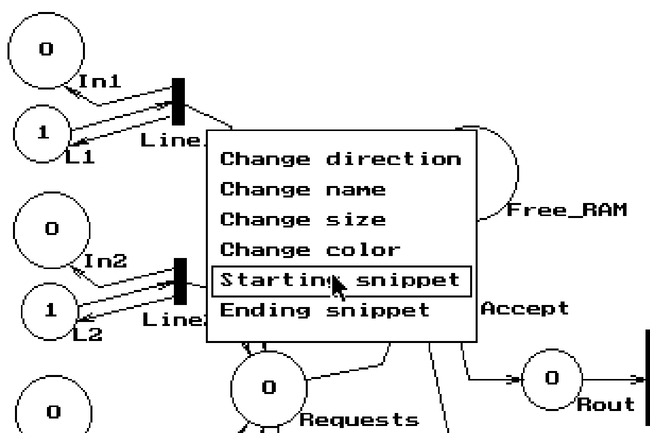
Obr.9 Síť hromadné obsluhy vyjádřená časovou Petriho sítí

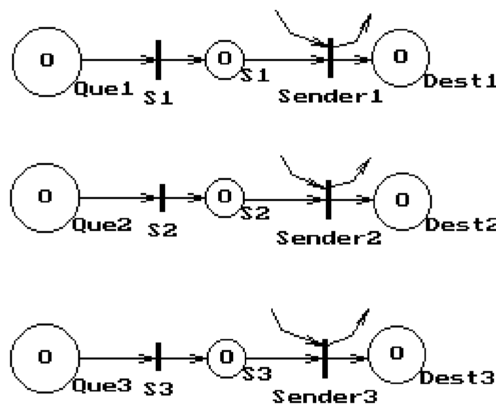
Článek rovněž vysvětluje, jak pracovat se statisticky monitorovanými objekty, s náhodnými čísly a s uživatelskými daty, napojenými na model. Ta totiž byla nutná k získání časových údajů jako je např. průměrná doba čekání ve frontě. Je to dáno tím, že v časové síti nejsou značky navzájem odlišeny. Známe počty značek, ale ne doby jejich setrvání v místě = frontě. Řešením bylo vytvořit frontu (seznam) a při každém umístění značky do místa vytvořit o této značce záznam s časem příchodu a tento záznam zařadit do fronty. Práce s uživatelskými daty je tak řízena a synchronizována událostmi v časové síti. Součástí PetriSimu byl proto unit Pascalu pro práci se seznamy velmi podobný systémové třídě Simset jazyka Simula. Přestože práce s frontami tím byla velmi zjednodušena, práce s nimi nebyla triviální a rovněž bylo nutné se nejdříve seznámit s unitem Simset. Jeho používání bylo velmi jednoduché pouze pro toho, kdo znal stejně nazvanou třídu Simuly. Proto v dnešní verzi PetriSimu je již statistika o době setrvání značek v místech poskytována automaticky, uživatel programuje pouze měření času setrvání v celé síti, nebo jejich částech. Nevýhodou práce s PertiSimem verze 2 bylo oddělení uživatelského kódu od sítě. Tento kód byl umístěn v samostatném unitu psaném v prostředí Turbo Pascalu. Další vývoj byl proto orientován na vytvoření interaktivního prostředí, kde celý model včetně uživatelského kódu je vytvořen editorem PetriSimu.

2.4.3 Grafické prostředí PetriSimu

Programování simulačních modelů v PetriSimu lze označit jako programování řízené událostmi (Event Driven Programming), což platí pro většinu aplikací pod operačními systémy Windows. Znamená to, že programátor píše úseky kódu, které definují co se má stát v případě určité události

jako je např. stisknutí tlačítka myši. V případě PetriSimu jsou těmito událostmi zahájení a ukončení provedení přechodu. Uživatelský kód je tak tvořen řadou navzájem téměř nezávislých úseků (code snippets), z nichž prakticky všechny jsou tvořeny sekvenční posloupností pouze několika příkazů. Logický celek, kterým je simulační model, je z těchto úseků vytvořen časovou sítí, která tak vytváří složité výpočetní struktury, které jsou při použití klasických simulačních jazyků naprogramovány pomocí speciálních k tomu určených příkazů. Programování je proto po vytvoření časové sítě už relativně snadné a pro výuku je podstatná ta skutečnost, že se uživatel nemusí učit žádné speciální příkazy. Předpokladem je pouze programování v Turbo Pascalu na běžné úrovni. Článek [A3] popisuje implementaci grafického uživatelského prostředí (GUI) PetriSimu. Zejména je popisována technika, jak jsou jednotlivé úseky kódu sestaveny do programu, jehož překladem vzniká proveditelný program *petrisim.exe*, který obsahuje celé prostředí a jeden uživatelský simulační model. Samotné úseky kódu jsou psány editorem, který je aktivován z PetriSimu. Obr.10 ukazuje situaci těsně před přechodem do textového editoru, kterému je předána kostra procedury (viz obr.4, kurzíva) realizující kód aktivovaný při zahájení provedení přechodu *Line1*. Editor si vybírá uživatel sám, standardně je používán editor *edit.com*, který je součástí Windows.





Obr.10 Grafické prostředí PetriSimu - aktivace textového editoru

2.4.4 Použití mechanismu dědění

PetriSim byl postupně budován rozšiřováním modelovacích schopností časových sítí a přidáváním podpůrných prostředků pro usnadnění psaní uživatelských modelů. Článek [A4] ukazuje, jak lze pro tento účel s výhodou využít mechanismus dědění (inheritance), který je základní technikou objektově-orientovaného programování. Je naznačeno, jak byly s minimálními zásahy do stávajícího kódu implementovány inhibiční hrany a rozšíření časových sítí s cílem simulovat sítě hromadné obsluhy[9]. V podstatě je zde popsána implementace typů míst a přechodů, viz kapitolu 2.2.3.

[9] *Poznamenejme, že myšlenka modelovat sítě hromadné obsluhy pomocí Petriho sítí není nová. Pro tento účel byly na Universitě v Dortmundu vytvořeny speciální sítě na vysoké úrovni nazvané Queueing Petri Nets, kde jednotlivá místa mohou být obslužnými uzly s frontou a vícekanálovou obsluhou. Obsluha je tvořena směsí exponenciálních rozdělení, což umožňuje kvantitativní analýzu pomocí aparátu Markovských procesů. Tyto sítě nejsou přímo určeny k simulaci. Viz stránku [28].*

2.4.5 Vytváření uživatelských modelů

V této kapitole je uveden stručný přehled funkcí a procedur PetriSimu určených k podpoře programování uživatelských modelů a k vyhodnocení experimentu..

MarkingOf(*p*) je funkce, která vrací značení místa *p*. Používá se v případech, kdy chování přechodů závisí na stavu sítě.

ChangeMarking(*p,m,ok*) je procedura, která změní označení místa *p* na hodnotu *m*, *ok* obsahuje výsledek. Samozřejmě se jedná o akci velmi nebezpečnou, která by měla být použita jen výjimečně.

Terminate ukončí při nejbližší příležitosti experiment.

FireDelay(*d*) generuje dobu provádění přechodu. Předpokládá se, že tato procedura je volána ve startovacích úsecích časových přechodů. Parametrem je typicky funkce, která vrací hodnotu generovanou podle nějakého náhodného rozdělení. Lze ovšem generovat prodlevu závislou na stavu sítě, času, apod.

PNTime vrací aktuální čas modelu.

ShowNet zobrazí síť v okamžitém stavu. Přechody ve stavu provádění jsou vyznačeny spolu s časem ukončení. Tato procedura se používá k ladění modelu v krokovém režimu.

Většina modelů používá pouze proceduru *FireDelay* ke generování prodlev a popřípadě některé z následujících procedur, pokud uživateli nestačí standardní výstup, který je uveden dále na obr.13. Použití některých procedur je ukázáno v následujících příkladech.

Presults(*p*, ...) vrací statistiku místa *p*.

Tresults(*t*, ...) vrací statistiku přechodu *t*.

GG1Results(*p*, *t*, ...) vrací výsledky systému G/G/1 kde *p* je fronta a *t* je obsluha.

PReport(*f*, *p*) uloží formátované výsledky místa *p* do textového souboru *f*.

TReport(*f*, *t*) uloží formátované výsledky přechodu *t* do textového souboru *f*.

GG1Report(*f*, *p*, *t*) uloží formátované výsledky systému G/G/1 trořeného frontou *p* a obsluhou *t* do textového souboru *f*. Viz obr.11 a obr.12.

Preparefile vrací jméno nově otevřeného textového souboru.

ShowThemAll(*f*) zobrazí a uloží do textového souboru *f* výsledky všech míst a všech přechodů odděleně.

ShowThemInTable(*f*) zobrazí a uloží do textového souboru *f* výsledky všech míst a přechodů přehledně ve dvou tabulkách. Tento výstup je implicitní, viz obr.13.

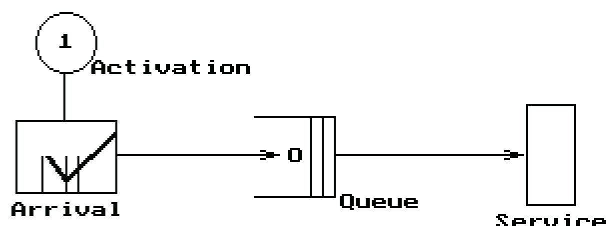
2.5 SIMULACE SYSTÉMŮ HROMADNÉ OBSLUHY

Systémy hromadné obsluhy jsou velmi často simulovány, protože analytické metody většinou buďto neexistují, nebo jsou založeny na nesplněných předpokladech, jako je např. exponenciální rozdělení intervalů mezi příchody zákazníků nebo trvání obsluhy. Článek [A5] se proto snaží ukázat, jak lze tyto sítě simulovat pomocí časových sítí PetriSimu. Článek ukazuje, jak lze vyjádřit pomocí časových sítí jednotlivé uzly různých typů a jak je propojovat do sítě.

2.5.1 Systémy s jednou frontou

Nejdříve je sítí tvořenou dvěma místy a dvěma přechody simulován systém G/G/1, viz obr.11. Po nakreslení sítě stačí k vytvoření modelu napsat 3 řádky kódu. Dva řádky, viz. obr.11, generují prodlevy mezi příchody zákazníků a trvání obsluhy. Poznamenejme, že text psaný kurzívou je generován PetriSimem. Funkce *Normal3Sig* generuje náhodné veličiny podle normálního rozdělení, ovšem vrací pouze ty, které jsou v rozmezí $\pm\sigma$, čili v uvedeném příkladu v [1, 7]. Třetí

řádek je aktivací procedury, která vypisuje standardní výsledky simulace systému G/G/1 do zadaného souboru, viz obr.12. Uživatel musí zadat jméno místa a jméno přechodu, protože ze struktury sítě nelze odvodit, která dvojice místo-přechod tento systém tvoří.




```
Procedure _PS1; { Starting snippet of Arrival }
Begin
  FireDelay(UniformR(2,8));
End;
Procedure _PS2; { Starting snippet of Service }
Begin
  FireDelay(Normal3Sig(4,1));
End;
```

Obr.11 Model systému G/G/1

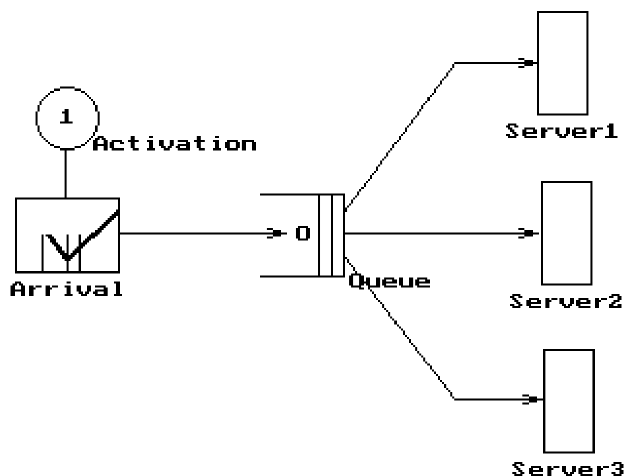
```
Report on G/G/1 made of queue Queue & server Service
Queue length
Average: 0.218  Min: 0.00  Max:  5.00  StdDev: 0.475
Queue waiting time
Average: 1.09   Min: 0.00  Max: 19.42  StdDev: 1.777
Server utilization: 0.800
Service duration
Average: 4.006  Min: 1.002363 Max: 6.992 StdDev: 0.989
Average system size:      1.017
Average system wait:      5.096
Effective arrival rate:   0.1997
```

Obr.12 Výsledky simulace systému G/G/1

Zobecnění na vícekanálovou obsluhu zařazením více obsluh = přechodů je evidentní. Model systému G/G/c je na obr.13, kde je také ukázán standardní výstup poskytovaný PetriSimem. Pro každé místo je vypsána statistika hodnoty označení, statistika doby setrvání značek v místě a celkový počet přidaných značek. Pro každý přechod je vypsáno jeho využití (podíl doby kdy probíhá provedení k celkové délce simulace), statistika trvání provedení a počet provedení. K vytvoření tohoto modelu tedy bylo nutné po nakreslení sítě pouze napsat 4 řádky kódu pro generování prodlev, podobně jako na obr.11. Poznamenejme, že intervaly mezi příchody byly v tomto případě rovnoměrné v [2, 8], obsluhy normální s parametry (11,3), (12,3) a (14,4).

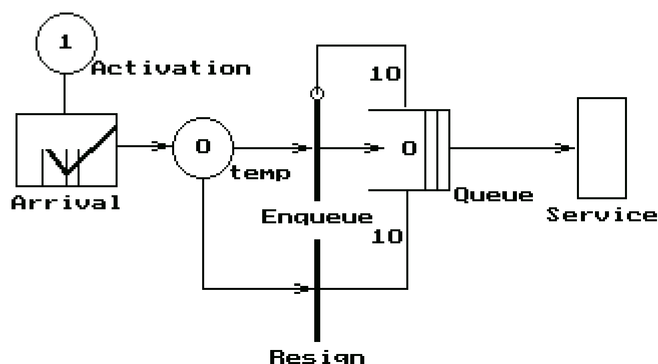
Následuje model systému G/G/1/K s omezenou kapacitou, viz obr.14, jehož zobecnění na G/G/c/K je evidentní. Testovací hrana aktivuje při plné frontě rezignaci, inhibiční hrana zařazuje zákazníka do fronty. Poznamenejme, že je testována délka fronty, nikoliv stav systému, čili na obr.14 je model systému G/G/1/11 s maximální frontou 10. Model tedy nelze přímo zobecnit na nulovou frontu G/G/c/c. Zde by bylo nutné zařadit místo modelující připravenost obsluhy (prázdné

během jejího provádění). Toto místo by inhibiční hranou aktivovalo rezignaci, normální hranou pak samotnou obsluhu. Fronta by v tomto modelu neexistovala.




```
Report on the net  :   G/G/3 model
Experiment duration:   100000.000
Current time       :   100005.248
Places:
Name Marking Mean Min Max StDev AvWait  Max StDev Added
=======================================================
Queue    1  0.197   0   5 0.493 0.987 22.44 2.034 19987
Activ.   1  1.000   1   1 0.000 0.000  0.00 0.000     0
Transitions:
Name Utilization MeanFire   Min     Max  StDev  Firings
=======================================================
Arrival     1.000     5.004 2.000104  8.000 1.731   19987
Server1     0.796    10.966 2.030113 19.998 2.969    7261
Server2     0.815    12.028 3.120290 20.903 2.935    6779
Server3     0.828    13.934 2.776136 25.653 3.973    5945
```

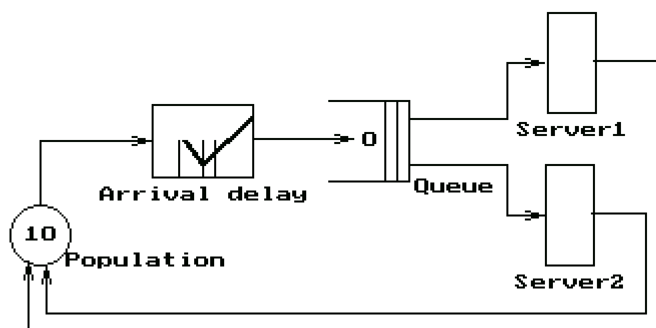
Obr.13 Model systému G/G/c a výsledky simulace



Obr.14 Model systému G/G/1/K

Poněkud složitější je modelování systémů s omezenou populací zákazníků, kde intervaly mezi příchody závisí nepřímo na počtu zákazníků mimo obsluhu. V prvé řadě je nutné tuto závislost definovat. Model na obr.15 je založen na předpokladu teoretických modelů omezené populace. Pro

každého zákazníka je definována četnost příchodů λ (např. pro stroje je obvyklé udávat průměrnou dobu mezi poruchami $1/\lambda$). Je-li M celkový počet zákazníků a n okamžitý stav systému, pak četnost příchodů je $(M - n)\lambda$. Obr.15 modeluje systém s 10 zákazníky, neomezenou kapacitou a dvěma obslužnými uzly. Příchod zákazníků modeluje přechod *Arrival delay*, jehož prodleva je v tomto případě rozdělená exponenciálně, což na rozdíl od teoretických modelů samozřejmě není podmínkou. Na obr.15 je vidět, jak je střední doba rozdělení odvozena od označení místa *Population*, které obsahuje zákazníky mimo obsluhu. V tomto případě je četnost λ=0.1 pro každého zákazníka, jednička je přidána proto, že značka už je v době provádění této procedury vyjmuta..




```
Procedure _PS1; { Starting snippet of Arrival delay }
var mean:real;
Begin
   mean := 1/((MarkingOf('Population')+1)*0.1);
   FireDelay(Exponential(mean));
End;
```

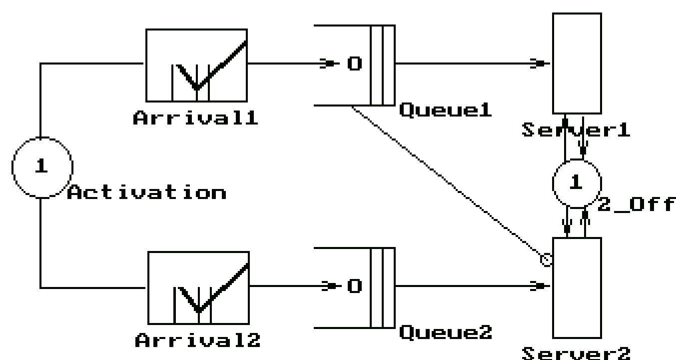
Obr.15 Model systému G/G/2 s omezenou populací

Dalším zobecněním jsou systémy s příchody a/nebo obsluhou po dávkách (Bulk Input, Bulk Service). Situace je triviální v případě, kdy velikost dávek je konstantní a obsluha vždy čeká na celou dávku. Dávky lze za těchto předpokladů modelovat násobnými hranami. V případě náhodných dávek nebo v případě, kdy je možná obsluha dávek menších, je nutné modifikovat označení programem, což je vždy nebezpečné, protože se vlastně jedná o porušení mechanismu Petriho sítě. Programování příchodu náhodné dávky v [1, 5] je ukázáno na obr.16.

```
Procedure _PE1; { Ending snippet of Arrival }
var OK:boolean;
Begin
  ChangeMarking('Queue',MarkingOf('Queue') + UniformI(1,5),OK);
End;
```

Obr.16 Programování příchodu náhodné dávky

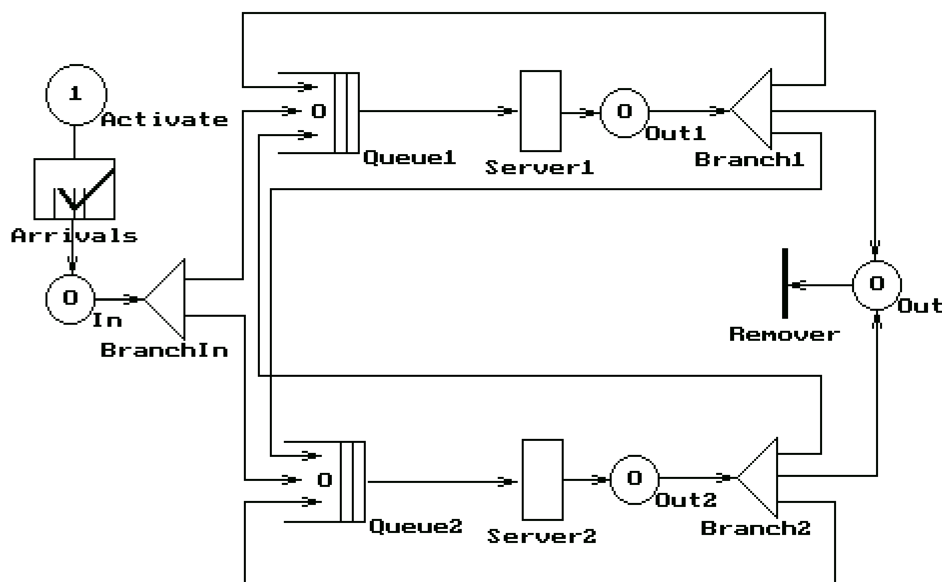
PetriSim neumožňuje přiřadit značkám prioritu nebo je klasifikovat do tříd. Na obr.17 je model obsluhy se dvěma prioritami zákazníků. Ve skutečnosti jedna fronta je modelována dvěma frontami pro dvě různé priority s oddělenými generátory, což lze rozšířit na libovolný počet. Obsluhy jsou také dvě, ale pomocí semaforu *2_Off* je zajištěno, že vždy probíhá pouze jedna. Inhibiční hrana zajistí, že spodní obsluha pro nižší prioritu může začít pouze, je-li horní fronta prázdná. Po vyjmutí inhibiční hrany by se jednalo v podstatě o obsluhu zákazníků dvou různých tříd stejné priority, kdy by bylo možné definovat různou dobu obsluhy podle třídy zákazníka.



Obr.17 Model systému se dvěma prioritami zákazníků

2.5.2 Sítě hromadné obsluhy

Vytvoření sítě propojením výše uvedených základních uzlů je evidentní, průchod zákazníků lze modelovat již dříve popsaným větvícím přechodem. Na obr.18 je jednoduchá síť se dvěma frontami, která byla pro porovnání modelována také pomocí Extendu, Arenou a simulátorem sítí v JavaScriptu při použití balíčku JSSim popsaném v příští kapitole. V tab.1 je porovnána rychlost těchto čtyř simulátorů. Intervaly mezi příchody a délky obsluh byly exponenciální, aby bylo možné porovnat výsledky simulace s teoretickým Jacksonovým modelem. Shoda byla dobrá, což není překvapující. Délka experimentu byla zvolena tak, že bylo obslouženo celkem asi 20000 zákazníků. Simulace byly provedeny na počítači s procesorem PIII, 800MHz, model v JavaScriptu byl otevřen prohlížečem Internet Explorer 5. Z hodnot v tab.1 vyplývá, že rychlost PetriSimu se díky rychlosti Turbo Pascalu blíží rychlosti profesionálních simulátorů.



Obr.18 Model sítě hromadné obsluhy

Simulátor	Arena	Extend	JSSim	PetriSim
Čas [s]	5	10	25	13

Tab.1 Porovnání rychlosti simulátorů

PetriSim se osvědčil jako nástroj pro práci s Petriho a časovými sítěmi pro potřeby výuky a výzkumu. Je volně k dispozici na webu [24] včetně zdrojových textů a on-line dokumentace s celou řadou příkladů.

3 JSSIM

JSSim (JavaScript Simulator) je souborem deklarací v jazyce JavaScript, které tvoří prostředí pro diskrétní simulaci orientovanou na události. JSSim vznikl postupným zobecněním funkcí použitých při vytváření diskrétních simulačních modelů zabudovaných do dokumentů napsaných v jazyce HTML.

3.1 MOTIVACE

World Wide Web sítě Internet (dále web) je soubor navzájem provázaných dokumentů zvaných webovské stránky, pro jejichž vytváření a zobrazování platí jednotná pravidla. Stránky jsou psány většinou v jazyce HTML (HyperText Markup Language) [3] a pro přístup k nim platí pravidla reprezentovaná protokolem HTTP (HyperText Transfer Protocol). Potřeba fungujících vazeb a komunikace si tak vynutila normalizaci v celosvětovém měřítku, což nebylo dříve u izolovaných počítačů z konkurenčních důvodů možné. Web je tak platformou, kde určitá webovská stránka by měla být zobrazena stejným způsobem různými prohlížecími programy na různých počítačích a pod různými operačnímy systémy. Skutečnost se tomuto ideálu blíží. Rozdíly v zobrazení dokumentů různými prohlížeči nejsou velké a jsou způsobeny zejména velmi rychlým vývojem, který předbíhá normalizaci.

Webovské stránky nejsou pouze pasivní texty a grafika. Stránky mohou obsahovat kód s cílem umožnit jejich dynamické chování a umožnit jejich využití ke sběru dat, což zahrnuje mimo jiné verifikaci dat na straně uživatele. Došlo tak poprvé v historii ke skutečné normalizaci programovacích jazyků[10], z nichž nejdůležitější je Java [6] a JavaScript [7]. Přes vnější podobu danou syntaxí odvozenou z jazyka C se jedná o dva různé jazyky. Oba umožnují vytváření libovolně rozsáhlých programů, oba obsahují všechny prostředky běžné u vyšších programovacích jazyků. Z jistého pohledu jsou však zcela odlišné. Java je kompilovaný[11], objektově-orientovaný jazyk s přísnou kontrolou typu proměnných. JavaScript je naopak interpretovaný jazyk, jehož proměnné nejsou deklarovány, a lze jim kdykoliv přiřadit hodnoty libovolného typu včetně funkcí. Mezi Javou a JavaScriptem existuje celá řada rozdílů, které jsou přehledně shrnuty v on-line dokumentaci[12]. Z dalšího popisu bude zřejmé, že podstatný je rozdíl ve vazbě na HTML kód, který definuje hostitelskou stránku. Java ve stránce existuje ve formě tzv. appletu, který lze ze stránky aktivovat, ale dále je na ní nezávislý. Např. všechny ovládací prvky si applet vytváří ve zvláštním okně. Naproti tomu JavaScript je do HTML kódu integrován. Tzv. značky (tag) jazyka HTML mohou obsahovat úseky v JavaScriptu, kterými programátor definuje co se má stát při různých událostech, jako je např. pohyb myší nad textem, stisknutí tlačítka, apod. K práci s JavaScriptem není nutné mít žádné překladače, protože kód je přímo interpretován prohlížečem. Programy lze psát libovolným textovým editorem, editory jazyka HTML standardně umožňují psaní příkazů v JavaScriptu. Jsou rovněž zdarma k dispozici ladicí programy pro nejčastěji užívané prohlížeče (Internet Explorer, Netscape Communicator).

---

[10] *Snaha o normalizaci programovacích jazyků existovala vždy, nikdy jí však nebylo skutečně dosaženo. Nejblíže byla pravděpodobně Simula, ovšem za cenu „zmrazení" v počátcích vývoje, což je jedním z důvodů malého rozšíření tohoto jazyka.*

[11] *Produktem kompilace Javy je tzv. „byte code", který je při provádění programu většinou interpretován. Z pohledu uživatele je však Java kompilovaným jazykem.*

[12] *Viz [29] kde je řada dalších odkazů.*

Tyto skutečnosti jsou důvodem, že podle některých autorů je 80% aplikací na webu psáno v JavaScriptu, [6]. JavaScript byl původně určen zejména ke zpracování dat na straně uživatele (client side) před jejich odesláním do serveru, ve kterém je stránka uložena. Toto zpracování ovšem může být libovolně složité, jsou k dispozici všechny standardní prostředky k numerickým a nenumerickým operacím. HTML, ve kterém je JavaScript integrován, řeší problém rozhraní mezi programem a uživatelem. Na rozdíl od Java appletu se všechny ovládací prvky snadno a rychle zapíší přímo v HTML, což platí pro vstup dat, ovládání programu a výstup výsledků. Tyto HTML objekty lze pak přímo používat v JavaScriptu v podstatě jako proměnné. Nabízí se tak možnost implementovat v JavaScriptu + HTML aplikace určené k řešení různých problémů včetně implementace jednoduchých a středně složitých simulačních modelů. Samozřejmě takto nelze realizovat rozsáhlé simulační studie.

K ověření tohoto předpokladu jsem napsal v JavaScriptu několik simulačních modelů systémů hromadné obsluhy, které byly zpřístupněny na webu. Výsledky byly značně povzbudivé ze dvou důvodů. Modely předně vzbudily značnou pozornost, dostal jsem řadu reakcí z celého světa. Dále se pak ukázalo, že interpretační programy dnešních prohlížečů jsou dostatečně rychlé k tomu, aby bylo možné úspěšně simulovat i netriviální středně složité modely. Problémem není ani kapacita vnitřní paměti, která je dnes obrovská a stále rychle roste. JavaScript má stejně jako Java garbage collector, není proto nutné se příliš starat o alokaci paměti. Jediným problémem je zákaz přístupu na disk. Částečným řešením je použití tzv. cookies, které však mají malou kapacitu a ne každý uživatel je povoluje vytvářet. Simulační modely na webu jsou dostupné doslova každému, kdo je připojen na Internet. Lze je použít (a bylo jich použito) k řešení praktických problémů. Jsou také velmi výhodné pro potřeby výuky. Obecně jsem přesvědčen, že v řadě aplikací je zbytečně zatěžován server. Typický počítač na straně uživatele (client) je dnes vybaven velmi rychlým procesorem a pamětí velké kapacity. Lze mu proto svěřit veškeré zpracování dat, které nevyžaduje častý přístup do centrální databáze nebo častou komunikaci s jinými klienty. Aplikace popisované v této kapitole se řídí touto filosofií a lze je proto z tohoto pohledu označit jako aplikace na straně uživatele (client side applications). Následující kapitoly se zabývají problémy, které bylo nutno řešit s cílem vytvořit objektově-orientovaný nástroj umožňující relativně snadné psaní simulačních modelů. Při psaní modelů postupně vznikl soubor deklarací, které jsou využitelné obecně. Lze je proto považovat za nástroj k diskrétní simulaci v JavaScriptu, který jsem pracovně nazval JSSim. V Javě existuje simulačních knihoven několik[13], pro JavaScript jiný simulační nástroj neznám.

3.2 OBJEKTOVĚ-ORIENTOVANÉ PROGRAMOVÁNÍ V JAVASCRIPTU

Jazyk JavaScript je definován v již zmíněné normě ECMA 262. Existuje ve dvou formách. Tzv. Server Side JavaScript je kompilován do formy zvané bytecode a interpretován serverem před odesláním stránky. Client Side JavaScript je interpretován prohlížeči na straně uživatele při zobrazení stránky a později při akcích uživatele, které JavaScript aktivují. Pro jakékoliv operace prováděné na straně uživatele je proto nutné použít Client Side JavaScript, na který se dále omezím. JavaScript obsahuje prostředky, které jsou dnes standardní ve všech vyšších jazycích (strukturované příkazy, funkce, základní datové typy včetně strukturovaných). Není to však ve své základní podobě objektově-orientovaný jazyk v klasickém slova smyslu založený na třídách a jejich instancích. Jeho autoři tvrdí, že je to objektový jazyk založený na tzv. prototypech (prototype-based language), viz např. [7]. Po bližším seznámení s prototypy JavaScriptu bylo zřejmé, že objektový model JavaScriptu lze snadno používat tak, že alespoň z hlediska uživatele se jedná o klasický model orientovaný na třídy (class-based). Článek [B1] ukazuje, jak lze

[13] *Např. JavaSim, vytvořený na University of Newcastle upon Tyne [30], je souborem balíčků (packages) v Javě, který umožňuje procesově orientovanou diskrétní simulaci. Jiný nástroj pro procesovou simulaci v Javě je simjava vytvořená na University of Edinburgh [31].*

v JavaScriptu vyjádřit třídy a jejich vlastnosti a metody[14] a jak lze pomocí konstruktérů (constructor) vytvářet instance a jejich vlastnosti a metody. Jazyk JavaScript má jednu zajímavou vlastnost, kterou je možnost modifikace instancí na úrovni kódu. Je to dáno tím, že v JavaScriptu je text funkce považován za hodnotu, na kterou lze následně použít operátor volání ( ). Instance vytvořené podle určitého vzoru tak mohou mít různé metody stejného jména. Modifikace může pokračovat tak, že některým instancím lze přidat vlastnosti a metody, které jiné instance nemají. Samozřejmě je pak otázkou, zda lze u takových instancí ještě hovořit o jejich třídách. Považuji za výhodný spíše opačný postup: úmyslně nevyužívat všechny možnosti, které JavaScript s neexistující kontrolou typů nabízí, a programovat tak v podstatě stejně, jako v klasických objektově-orientovaných jazycích. Tento přístup znamená používat pouze metody uložené v prototypech, které jsou shodné pro všechny instance vytvořené daným konstruktérem. Funkce použitá jako konstruktér, která vytváří hodnotové vlastnosti instancí, pak nahrazuje deklaraci třídy v kompilovaných jazycích.

Druhým základním principem objektově-orientovaného programování je mechanismus dědění (inheritance) vyjádřený stromovou strukturou, ve které má každá třída (s výjimkou nejvyšší) svou nadtřídu[15] (superclass) a sama může být nadtřídou libovolného počtu podtříd (subclass)[16]. K implementaci dědění nabízí JavaScript řešení založené na nahrazení prototypu instancí nadtřídy. Toto řešení považuji za nevýhodné ze dvou důvodů. Prototyp má v tomto případě také hodnotové vlastnosti instance nadtřídy, které nelze instancemi modifikovat a které jsou v případě vytvoření vlastností stejného jména konstruktérem podtřídy neviditelné. Výsledkem je tedy pouze zbytečně alokovaná paměť, což není podstatné. Druhý důvod je závažný. Prototypy mají vlastnost nazvanou *constructor*, kterou lze testovat typ instancí. Pokud přepíšeme standardní prototyp odkazem na instanci nadtřídy, dojde samozřejmě k porušení hodnoty vlastnosti *constructor*. Alternativou je explicitní programované dědění založené na kopírování metod z prototypu nadtřídy. Na obr.19 je funkce *inherit*, která kopíruje metody (ne hodnotové vlastnosti) a příklad jejího použití, po kterém následuje vytvoření další metody podtřídy.

```
function inherit(from,to) {
   for(var p in from.prototype)
    if (typeof from.prototype[p] == "function")
      to.prototype[p] = from.prototype[p]
};
inherit(point,colorpoint);
colorpoint.prototype.changecolor = function(c) {
   this.color = c
};
```

Obr.19 Programované dědění metod v JavaScriptu

K dědění hodnotových vlastností, které jsou vytvářeny konstruktérem, lze použít techniku ukázanou na obr.20. Samotné vytvoření vlastností provede zvláštní funkce, kterou volá konstruktér dané třídy a obdobné funkce podtříd, v čemž lze pokračovat na libovolném počtu úrovní. Jinak by totiž bylo nutné vytvoření vlastností v podtřídě zopakovat, což přináší nebezpečí inkonsistence.

---

[14] Vlastnosti (properties) a metody tříd existují pro třídu jako takovou, ne pro jednotlivé instance. Některé objektově-orientované jazyky, jako např. Simula, vlastnosti a metody tříd nemají. Všechny objektově-orientované jazyky umožňují definovat vlastnosti a metody instancí, což je první základní princip objektově-orientovaného programování (encapsulation).

[15] Umožňuje-li jazyk vícenásobné dědění (multiple inheritance), pak může být nadtříd více.

[16] Poznamenejme, že ne všechny objektově-orientované jazyky používají pojem „class“. Např. Object Pascal nebo simulační jazyk Modsim používají název „object“. Princip je však shodný.

Takto lze přidat vlastnost nadtřídy s tím, že změna se automaticky promítne do všech podtříd, stejně jako u klasických objektově-orientovaných jazyků. Obě techniky (dědění metod a vlastností) umožňují vícenásobné dědění (multiple inheritance).

```
function pointProperties(xcoor,ycoor,obj) {
   obj.x = xcoor;
   obj.y = ycoor;
};
function point(initx,inity) {                // constructor
   pointProperties(initx,inity,this)
};
function colorpointProperties(xcoor,ycoor,icolor,obj) {
   pointProperties(xcoor,ycoor,obj);
   obj.color = icolor;
};
function colorpoint(initx,inity,initcolor) { // constructor
   colorpointProperties(initx,inity,initcolor,this)
};
```

Obr.20 Dědění hodnotových vlastností v JavaScriptu

Třetím základním principem objektově-orientovaného programování je tzv. polymorphismus, který v podstatě znamená možnost modifikace chování zkompilovaného kódu v závislosti na konkrétním typu vytvořených instancí. Např. aktivace metody *queue.getfirst()* vrátí první prvek fronty, je-li v proměnné *queue* instance fronty typu FIFO, resp. poslední, je-li instance typu LIFO. Tohoto efektu je u objektově-orientovaných jazyků orientovaných na třídy dosaženo pomocí složitého mechanismu nazvaného „late binding“[17]. V JavaScriptu tento problém neexistuje. Interpretovaný JavaScript bez kontroly typů je polymorfický z principu. *x.m()* lze provést kdykoliv za předpokladu, že proměnná *x* odkazuje na objekt, který má metodu *m*.

Článek [B1] končí popisem techniky, která nemá obdobu v jiných objektově-orientovaných jazycích. Programované dědění totiž umožňuje kopírovat pouze některé vybrané metody a vytvářet tak „podtřídy“, které jsou zjednodušením „nadtřídy“. Této techniky jsem použil např. k definici třídy, jejíž instance reprezentují diskrétní náhodné proměnné. Tato třída je zjednodušením třídy pro generování obecných náhodných proměnných, která již byla dříve k dispozici. Samozřejmě je nutné zaručit, aby vybrané metody byly soběstačné. Tohoto efektu lze u hodnotových vlastností dosáhnout tak, že vytvoření vlastností rozdělíme do několika funkcí a při zjednodušení zařadíme pouze ty, které jsou potřebné. Vhodným názvem pro tuto techniku by mohlo být zjednodušení (simplification). Jedná se vlastně o opak dědění, při kterém nadtřídu obohacujeme. Obohacení v jistém směru však může následovat i po počátečním zjednodušení.

3.3 SIMULÁTOR SÍTÍ HROMADNÉ OBSLUHY

Existuje pouze jediný způsob, jak ověřit funkčnost přímého zabudování simulačního modelu do webovské stránky. Napsal jsem proto několik modelů, které jsou k dispozici na [25]. Jedná se o tři modely systémů s jednou frontou a obecný simulátor sítí. První je model systému M/M/1, který je určen k ověření teoretického modelu, jehož úplné odvození je zahrnuto. Další dva modely pak umožňují simulovat v podstatě libovolné systémy G/G/c s jednou frontou, která může mít omezenou kapacitu, populace může být také omezená, příchod a obsluha mohou být po dávkách. Intervaly mezi příchody a trvání obsluh mohou mít obecné rozdělení dané tabulkou distribuční funkce. Práce s těmito modely je velmi rychlá a pohodlná, což mi potvrdila řada jejich uživatelů. Některé podněty jsem také použil k vylepšení a rozšíření modelů. Dále jsem se proto zaměřil na

---

[17] *Jedná se totiž v podstatě o porušení principu typovaných proměnných.*

model sítě, jehož vytvoření je obtížnější, který je náročnější na procesor a paměť počítače, ale který je také ze známých důvodů mnohem atraktivnější pro řešení praktických problémů. Základní myšlenky a výsledky jsou shrnuty ve článku [B2].

JavaScript přímo nepodporuje nakreslení sítě myší tak, jak to umožňují již zmíněné interaktivní simulační prostředky jako je např. 0Extend nebo Arena. Popis sítě je proto textový. Nepovažuji to za podstatné omezení, zejména pokud se jedná o síť, kde se zákazník po obsluze přemísťuje náhodně do více možných uzlů. Pokud je taková síť rozsáhlá, diagram se brzy stane nepřehledným a v podstatě zbytečným. Ponechal jsem možnost zobrazit diagram, po stisknutí tlačítka se zadá umístění souboru v počítači uživatele. Modelovaná síť je tvořena dvěma typy uzlů: generátory zákazníků a obslužnými stanicemi, obou může být v modelu prakticky neomezený počet. Protože každý uzel je reprezentován řadou parametrů a mnoha výsledky po ukončení simulace, je vždy zobrazen jeden generátor a jedna obslužná stanice. Přepínání je pohodlné pomocí výběru (značka SELECT), který po otevření zobrazí čísla existujících generátorů a stanic, jejichž počty lze průběžně měnit. Vždy jsou také zobrazeny výsledky, které se vztahují k celé síti, jako je např. celkový počet ztracených zákazníků nebo průměrná doba setrvání v síti.

Dvěma typům síťových uzlů odpovídá deklarace dvou tříd, které jsou popsány ve článku [B2]. *Generátor* je zadán náhodným rozdělením intervalů mezi příchody zákazníků a jejich náhodným směrováním, které je modelované náhodným číslem cílové obslužné stanice. *Obslužná stanice* je zadána kapacitou a organizací fronty (FIFO nebo LIFO), počtem shodných obslužných kanálů, náhodným rozdělením trvání obsluhy a stejně jako generátor náhodným směrováním. Pro odchod zákazníka ze sítě je rezervováno číslo cílové stanice 0. Pro generátory a stanice poskytuje model výsledky ukázané na obr.21 a obr.22 pro síť M/M/1 vytvořenou automaticky při otevření modelu: jeden generátor, jedna stanice, intervaly i obsluha mají střední hodnotu 1. Délka experimentu byla 10000 časových jednotek, jejichž interpretace je daná uživatelem. Tlačítka Note zobrazují přesné definice výsledků (v tzv. „alert" okně prohlížeče).

Generator Results

Result	The value	Explanation
Number of arrivals	9752	Note
Average arrival interval	1.0254634681113161	Note
Minimum arrival interval	0.00017887726400765377	Note
Maximum arrival interval	8.751862846542984	Note

Obr.21 Výsledky simulace generátoru

Simulátor umožňuje práci se sítěmi prakticky neomezené velikosti a libovolné topologie. Uživatel nejdříve zadá počty generátorů a obslužných stanic, které pak postupně zobrazuje a zadává jejich parametry. Jednotlivá náhodná rozdělení, celé uzly a celou síť lze uložit do tzv. cookies. To je velmi výhodné, pokud se v síti vyskytuje více shodných nebo podobných uzlů nebo rozdělení. Podmínkou ovšem je, aby uživatel cookies povolil v nastavení prohlížeče. Problémem je také omezená kapacita cookies, která neumožňuje ukládání rozsáhlých sítí. Model proto obsahuje jednoduchou správu cookies - jejich zobrazení a vymazání z paměti.

Server Results

Result	The value	Explanation
Number of arrivals	9752	Note
Number of not waiting arrivals	387 (3.97%)	Note
Number of lost customers	0 (0%)	Note
Number of services	9739	Note
Average service duration	0.9870731113106974	Note
Minimum service duration	0.00010567747646169285	Note
Maximum service duration	10.531457204325278	Note
Average waiting time	13.21602922404416	Note
Average non zero waiting time	13.76287016381804	Note
Maximum waiting time	47.49982440313033	Note
Average time in server	14.203102335354858	Note
Minimum time in server	0.0020997091014578472	Note
Maximum time in server	48.76863071527714	Note
Average queue length	12.877967399977802	Note
Maximum queue length	55	Note
Utilization of server(s)	0.9612032667932526	Note

Obr.22 Výsledky simulace obsluhy

3.4 ŘÍDÍCÍ PROGRAM A PODPŮRNÉ PROSTŘEDKY

Rychlost simulace závisí do značné míry na rychlosti řídicího programu (v anglické literatuře nazývaného Simulation Engine), který udržuje seznam záznamů událostí seřazený podle času výskytu. Tento program plní tři základní funkce: zařazení záznamu na správné místo při plánování události, opakované vybírání záznamu s nejmenší hodnotou času výskytu a aktivaci příslušné události a vyjmutí záznamu, pokud má být dříve naplánovaná událost zrušena. Tyto tři funkce jsou v podstatě shodné u obou dnes nejpoužívanějších principů řízení času: simulace orientované na časově bezrozměrné události a simulace založené na paralelních procesech. Ve druhém případě reprezentují záznamy segmenty procesů a nikoliv nezávislé události. Při vytvoření obecně použitelného řídicího programu je proto nutné učinit v prvé řadě zásadní rozhodnutí, který z těchto

dvou způsobů vidění dynamiky v čase použít[18]. Procesová simulace je nejblíže realitě a je dnes defacto standardem při vytváření nových simulačních nástrojů. Události mají oproti procesům dvě výhody. Snadněji se implementují a snadněji se učí. Oba tyto faktory považuji v případě simulačního nástroje v JavaScriptu za rozhodující. Není určen odborníkům pro rozsáhlé simulační studie, typicky se bude jednat o malé nebo střední relativně jednoduché modely. Byl proto zvolen tento přístup (Discrete Event Simulation - DES), který z hlediska uživatele umožňuje dvě plánovací operace: *naplánování události v daném čase* a *zrušení naplánované události*.

Základní informace o implementaci řídicího programu a dalších podpůrných funkcích jsou obsahem článku [B3]. Pro kalendář událostí jsem zvolil datovou strukturu zvanou „heap", která je binárním stromem optimálního tvaru. Obě základní operace *zařazení záznamu* a *vyjmutí prvního záznamu* jsou provedeny v čase $O(\log_2 n)$, kde n je počet záznamů. Byla definována třída *heap*, jejíž instance nazvaná po vzoru Simuly *SQS* (Sequencing Set) je standardně kalendářem událostí[19]. Pro psaní diskrétních simulačních modelů obsahuje JSSim tyto funkce:

evnotice() je konstruktér záznamu události. Vrací objekt, jehož jediná vlastnost slouží k uložení času výskytu události. Další vlastnosti může přidat uživatel. Typicky půjde o typ události určený k výběru události.

initialize_run() vyprázdní kalendář po případném předchozím experimentu a vynuluje čas. Jde o inicializaci simulačního experimentu, po ní typicky následuje inicializace samotného modelu.

schedule(e, t) plánuje událost danou záznamem události *e* v čase *t*.

cancel(e) ruší výskyt události *e*.

simulation_run() je vlastní nastartování simulačního experimentu, který končí vyprázdněním kalendáře nebo může být ukončen uživatelem, viz níže uvedenou funkci *finish_run*.

Součástí uživatelské části modelu musí být tyto funkce:

finish_run() vrací „true", je-li splněna podmínka k ukončení experimentu. Lze testovat čas, stav modelu jako např. počet obsloužených zákazníků, apod.

eventroutine(e) je funkce aktivovaná řídicím programem po vyjmutí záznamu *e* z kalendáře. Typicky zde dochází k výběru události podle jejího typu.

Je rovněž nutné naprogramovat *zahájení* a *vyhodnocení* simulačního experimentu.

Jako ukázka programování v JSSimu je na obr.23 uvedena funkce modelující událost příchodu zákazníka. Po vytvoření zákazníka následuje test, zda lze zahájit obsluhu. To je možné za předpokladu, že je server volný (prázdný odkaz na obsluhovaného zákazníka *served*) a že není v poruše (*serverOK* true). *Waittime* je statistický objekt, který slouží ke sběru hodnot délky čekání a k jejich statistickému vyhodnocení. Jeho metoda *update* je volána při každé nové zjištěné hodnotě, která je v případě okamžitého zahájení obsluhy 0. Následuje vytvoření záznamu události

---

[18] *Existuje ještě třetí princip založený na aktivitách. Pro každou aktivitu je definována podmínka jejího spuštění, řídicí program zvyšuje čas po krocích, ve kterých testuje podmínky provedení aktivit. Nízká účinnost je evidentní, přesto lze tento postup vzhledem k jeho jednoduchosti doporučit tehdy, kdy je nutné naprogramovat simulační model v jazyce, který podporu řízení času nemá. Viz moje modely v Pascalu, které lze stáhnout z [32].*

[19] *Použití stromové struktury s logaritmickým časem trvání operací je nepochybně teoreticky nejrychlejší řešení. Praxe však teorii odpovídat nemusí. Problémem je režie daná relativní složitostí operací na stromech ve srovnání s jednoduchými operacemi na řetězeném seznamu. Při malém průměrném počtu záznamů v kalendáři tak může vést použití seznamu ke zvýšení rychlosti. Tuto zkušenost mi potvrdil autor Lund Simuly [9], který tvrdí, že seznam je nakonec vždy alespoň stejně rychlý jako stromy. Uživatel JSSimu má proto možnost výběru mezi stromem a seznamem.*

ukončení obsluhy a její naplánování. Nelze-li zahájit obsluhu, zákazník je zařazen do fronty. Podobnost s programováním v Simscriptu není náhodná.

```
function arrival() {        // customer arrival
   arrivals++;              // update counter of arrivals
   var cust = new customer(time);   // creating the customer
   var ev = null;
   if ((served==null)&&(serverOK)) {// can start service ?
     served = cust;         // yes, store served customer
     waittime.update(0);  // update statistics
     ev = new evnotice(); // create event notice
     ev.eventtype = 2;      // set event type (2 = end of service)
     served.event = ev;   // store served customer event
     schedule(ev,time + normal(meanservice,serviceStd));
   } else {
     queue.enqueue(cust); // no, enqueue the customer
   };
   ev = new evnotice();   // schedule next arrival
   ev.eventtype = 1;      // set event type (1 = arrival)
   schedule(ev,time + exponential(meaninterval));
};
```

Obr.23 Událost příchod zákazníka

Simulační jazyky nebo knihovny poskytují mimo řízení časování modelu celou řadu dalších funkcí. Jejich implementací se zabývá článek [B3], kde je podrobněji naznačena implementace front a statisticky monitorovaných proměnných. K jejich implementaci je použita technika Simscriptu zvaná „left monitoring“ navržená již ve článku [14][20] pod názvem „store association“. Princip je ten, že výskyt monitorované proměnné na levé straně přiřazovacího příkazu aktivuje rutinu, které je předána hodnota výrazu. Lze tak průběžně aktualizovat statistiku aniž je programátor jakkoliv zatěžován. V JavaScriptu (a v Pascalu) je nutné nahradit přiřazovací příkaz aktivací přiřazovací metody, která hodnotu uloží a zajistí vše potřebné. Viz aktualizaci statistiky doby čekání na obr.23. Při vyhodnocení experimentu pak lze přímo volat metody, které vrací průměr, směrodatnou odchylku, apod. Viz jejich použití na obr.24 a obr.25. JSSim obsahuje definici dvou tříd pro statisticky monitorované proměnné v čase, kdy momenty jsou počítány z časových integrálů (délka fronty) a pro proměnné, jejichž statistika závisí pouze na přiřazených hodnotách (doba čekání ve frontě). JSSim dále obsahuje třídy a funkce pro generování náhodných proměnných a další pomocné funkce pro validaci dat, práci s cookies, apod.

3.5 DISKRÉTNÍ SIMULACE V JAVASCRIPTU A VAZBA NA HTML

Článek [B4] se zabývá využitím úzké vazby mezi programy v JavaScriptu a v HTML s cílem zjednodušit co nejvíce výtváření dokumentů obsahujících simulační modely. Je naznačeno, jak konkrétně realizovat vazbu mezi popisem struktury dokumentu v HTML a JavaScriptem, který implementuje simulační model a jeho ovládání uživatelem.

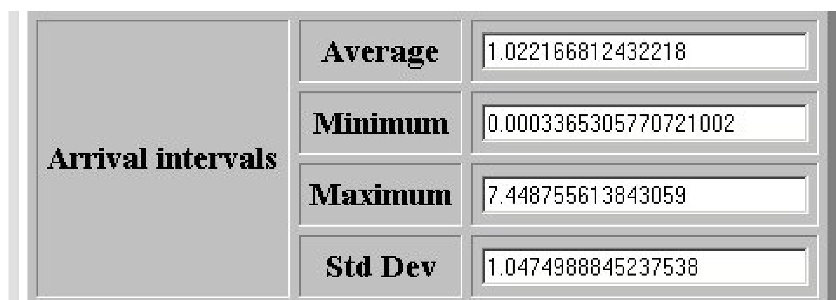
[20] *Mimochodem se jedná o sborník konference, kde byla poprvé představena Simula 67 a s ní, aniž si toho byli autoři vědomi, i objektově-orientované programování. Ve sborníku je rovněž záznam diskuse po přednesení McNeleyova příspěvku, kde Ole-Johan Dahl navrhuje řešení pomocí tříd, které jsem nezávisle později zvolil.*

Příklad na obr.24 ukazuje přímou vazbu mezi instancí třídy JavaScriptu a HTML objekty dokumentu. V tomto případě jde o instanci *intstat* statisticky monitorované veličiny určené ke sběru a vyhodnocení statistiky intervalů mezi příchody zákazníků. Na obr.24 je HTML kód, který zobrazuje část tabulky se statistikou intervalů mezi příchody a výřez okna prohlížeče (v tomto případě Internet Exploreru 6, ale rozdíly mezi prohlížeči jsou nepatrné). Aktualizace hodnot na obrazovce po ukončení simulace je provedena aktivací metody *scrupdate*, viz obr.24, příkazem *intstat.scrupdate("document.form1.intstat")*. Uvedená implementace je umožněna funkcí *eval* JavaScriptu, jejímž parametrem je zdrojový kód, který funkce provede. První aktivace funkce *eval* tak provede příkaz *document.form1.intstatav = average()*, který zobrazí průměrnou hodnotu intervalu v textovém poli *intstatav*. Je na uživateli, aby textová pole HTML byla správně nazvána. Toho lze snadno docílit definováním standardních úseků v HTML upravovaných podle potřeby operací „Replace All“, kterou mají prakticky všechny textové editory. Lze také použít nějaký makro pre-procesor, který by vygeneroval HTML kód automaticky.

```
JavaScript:
   tally.prototype.scrupdate = function(dname) {with(this){
     eval(dname + "av.value = average()");
     eval(dname + "mi.value = min");
     eval(dname + "ma.value = max");
     eval(dname + "sd.value = stdDev()");
   }};
```

```
HTML:
   <TR>
   <TH ROWSPAN=4> Arrival intervals </TH>
   <TH> Average </TH>
   <TD><INPUT TYPE="text" NAME="intstatav" SIZE=25></TD>
   <TR><TH> Minimum </TH>
   <TD><INPUT TYPE="text" NAME="intstatmi" SIZE=25></TD></TR>
   <TR><TH> Maximum </TH>
   <TD><INPUT TYPE="text" NAME="intstatma" SIZE=25></TD></TR>
   <TR><TH> Std Dev </TH>
   <TD><INPUT TYPE="text" NAME="intstatsd" SIZE=25></TD></TR></TR>
```

```
Screen:
```



Obr.24 Příklad vazby HTML - JavaScript

Na obr.25 je metoda *winupdate*, která dynamicky vygeneruje do zadaného okna HTML kód, který zobrazí statistické hodnoty. Podobné metody využívají modely ke generování výsledků v textové formě ve zvláštním okně. Lze je pak kopírovat do jiných dokumentů, např. do MS Wordu. Text na obr.25 byl takto zkopírován z okna prohlížeče.

```
JavaScript:
    tally.prototype.winupdate = function(stitle,w) {with(this){
      w.writeln(stitle + " statistics:" + "<BR><UL>");
      w.writeln("<LI> Average: " + average());
      w.writeln("<LI> Minimum: " + min);
      w.writeln("<LI> Maximum: " + max);
      w.writeln("<LI> Std Dev: " + stdDev() + "</UL>");
    }};
```

```
Screen:
  Arrival intervals statistics:
```

- `Average: 1.022166812432218`
- `Minimum: 0.0003365305770721002`
- `Maximum: 7.448755613843059`
- `Std Dev: 1.0474988845237538`

Obr.25 Příklad vazby HTML - JavaScript

Uvedená technika vazeb mezi instancemi JavaScriptu a objekty HTML umožňuje za předpokladu, že se spokojíme se standardním zadáváním vstupů a se standardním zobrazováním výsledků, prakticky automatickou generaci HTML kódu. Již nyní mi ušetřila spoustu práce. Jako další příklad je na obr.26 tabulka, která slouží k zadání parametrů náhodných proměnných, kterou používám standardně ve všech modelech. Ukazuje situaci před stlačením tlačítka *Add*, které přidá 4. položku do tabulky distribuční funkce. Význam dalších ovládacích prvků je zřejmý. Tato tabulka je generována HTML kódem, který obsahuje přibližně 5000 znaků na 190 řádcích. Automatické generování tohoto kódu dosazením několika parametrů do standardního polotovaru se ukázalo jako velmi výhodné.

Obr.26 Vstup parametrů náhodné proměnné

Simulační modely [25] lze používat přímo nebo po stažení lokálně. Modely jsou také zahrnuty do projektu tutORial organizace IFORS (International Federation of Operational Research Societies) koordinovaného univerzitou v Melbourne, viz [26]. Samotný JSSim lze stáhnout z jeho domovské stránky [33].

4 ZÁVĚR

V této práci je popsána implementace dvou programových balíčků. PetriSim je programové prostředí pro vytváření diskrétních simulačních modelů při použití grafického jazyka Petriho sítí. Po nakreslení sítě je chování modelu upřesněno napsáním většinou velmi krátkých úseků kódu, které jsou aktivovány při zahájení a při ukončení provádění časových přechodů. Lze tak po zvládnutí základů Petriho sítí a programování v Turbo Pascalu vytvářet modely, které je jinak nutné psát pomocí diskrétních simulačních jazyků, jejichž zvládnutí je zejména pro potřeby výuky nereálné. Dále je uveden výčet modelů základních systémů hromadné obsluhy s jednou frontou, které lze snadno spojovat do sítí. Přestože jsou tyto modely založeny na sítích PetriSimu, výsledky lze zobecnit na různé další typy Petriho sítí s prodlevami v přechodech.

JSSim je knihovna deklarací pro podporu diskrétní simulace orientované na události v jazyce JavaScript. Modely jsou integrované v HTML dokumentech, které jsou využity pro komunikaci modelů s uživatelem. Bylo takto vytvořeno několik modelů systémů hromadné obsluhy, které jsou po umístění na web dostupné doslova kdykoliv komukoliv. Příznivé reakce z mnoha zemí naznačují, že migrace simulačních modelů a dalších řešičů na web bude v budoucnosti pokračovat.

Oba balíčky lze volně stáhnout z jejich domovských stránek [24] a [33]. Zájemce o další detaily nebo o kopie článků tvořících habilitační práci mne může kontaktovat na domovské stránce [34].

5 LITERATURA

5.1 PŘEHLED PRACÍ TVOŘÍCÍCH HABILITAČNÍ PRÁCI

[A1] SKLENÁŘ, J. Environment for Modelling of Petri Net Based Networks in the LOGOS Language. In *Proceedings of the Third European Logo Conference EUROLOGO'91, Parma, 27-30 August 1991.* Edit. by E. Calabrese. Universita di Parma, p. 385-400.

[A2] SKLENÁŘ, J.: Discrete Simulation and Time Networks. In *Proceedings of the 22nd Conference of the ASU Object Oriented Modelling and Simulation, Clermont-Ferrand, 15-17 July 1996*, Edit. by A. Tanguy. University Blaise Pascal, p.57-67. ISSN 1102-593X.

[A3] SKLENÁŘ, J.: Event Driven Visual Programming in PetriSim Environment. In *Proceedings of the 23rd Conference of the ASU Object Oriented Modelling and Simulation, Stará Lesná, 25-27 August 1997*, VSŽ Informatika s.r.o. Košice, p.170-179. ISSN 1102-593X.

[A4] SKLENÁŘ, J.: Using Inheritance to Implement High Level Petri Networks. In *Proceedings of the 24th Conference of the ASU Object Oriented Modelling and Simulation of Environmental, Human and Technical Systems, Salzau, 24-30 September 1998*, Edit. by B. Breckling. Ecology Center, University of Kiel, p.173-182. ISSN 1102-593X.

[A5] SKLENÁŘ, J.: Simulation of Queueing Networks in PetriSim. In *Proceedings of the 16th European Simulation Multiconference Modelling and Simulation 2002, Darmstadt, 3-5 June 2002*, Edit. by K. Amborski and H. Meuth. SCS Europe, p. 403-407. ISBN 90-77039-07-4.

[B1] SKLENÁŘ, J.: Object Oriented Programming in JavaScript. In *Proceedings of the 26th ASU Conference Object Oriented Modeling and Simulation*, *Malta, 26-28 September 2000*, Edit. by J. Sklenář. University of Malta, p. 35-42. ISSN 1102-593X.

[B2] SKLENÁŘ, J.: Simulator of Queueing Networks. In *Proceedings of the 26th ASU Conference Object Oriented Modeling and Simulation*, *Malta, 26-28 September 2000*, Edit. by J. Sklenář. University of Malta, p. 127-135. ISSN 1102-593X.

[B3] SKLENÁŘ, J.: Client Side Web Simulation Engine. In *Proceedings of the 27th Conference of the ASU Model Oriented Programming and Simulation, Rättvik, Sweden, 11-16 October 2001*, ASU, p. 1-13. ISSN 1102-593X.

[B4] SKLENÁŘ, J.: Discrete Event Simulation in JavaScript. In *Proceedings of the 28th ASU Conference The Simulation Languages, Brno, 26 September-1 October 2002*, FIT VUT Brno, p. 115-121. ISSN 1102-593X.

5.2 VYBRANÁ LITERATURA

[1] ČEŠKA, M. *Petriho sítě*, Brno : Akademické nakladatelství CERM, 1994. 94 p. ISBN 80-85867-35-4.

[2] DAHL, O.J. *Discrete Event Simulation Languages*, Oslo : Norsk Regnesentral, 1966.

[3] DARNELL, R. et al. *HTML 4 Unleashed,* Professional Reference Edition. Indianapolis : Sams.net Publishing, 1998. 1195 p. ISBN 1-57521-380-X.

[4] DESROCHERS, A.A.; AL-JAAR, R.Y. *Applications of Petri Nets in Manufacturing Systems*, New York : IEEE PRESS, 1995. 348 p. ISBN 0-87942-295-5.

[5] DIAZ, M. (1983) Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models, *Computer Networks,* 1983, no. 6, p.419-441.

[6] ECKEL, B. *Thinking in Java,* Prentice Hall. Inc., 1998. 1110 p. ISBN 0-13100-287-2.

[7] FLANAGAN, D. *JavaScript - The Definitive Guide,* O'Reilly & Associates Inc., 1998. 776 p. ISBN 1-56592-392-8

[8] FORRESTER, J.W. *Industrial Dynamics,* The MIT Press, 1961.

[9] FRIES, G. Neformální diskuse s Göranem Friesem z Lundské university, autorem Lund Simuly, během konference *27th Conference of the ASU Model Oriented Programming and Simulation*, Rättvik, 2001.

[10] HAVERKORT, B.R. *Performance of Computer Communication Systems*. A Model.Based Approach, John Wiley & Sons, 1998. 495 p. ISBN 0-471-97228-2.

[11] KELTON, W.D., SADOWSKI, R.P., SADOWSKI, D.A. *Simulation with Arena*, 2nd ed. McGraw-Hill, 2002. 631 p. ISBN 0-07-239270-3.

[12] LINDEMANN, C. *Performance Modelling with Deterministic and Stochastic Petri Nets*, John Wiley & Sons, 1998. 405 p. ISBN 0-471-97646-6.

[13] MARKOWITZ, H.M., HAUSNER, B., KARR, H.W. *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall, 1963.

[14] McNELEY, J.L. (1968) *COMPOUND DECLARATIONS*, In *Proceedings of the IFIP Working Conference on Simulation Languages, Oslo, May 1967,* Edit. by J-O. Dahl. IFIP. North-Holland Publishing Company, 1968. p.292-304.

[15] MURATA, T. Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE*, 1989, Vol. 77, no. 4, April 1989, p.541-580.

[16] PETERSON, J.L. *Petri Net Theory and Modelling of Systems*, Prentice-Hall, 1981.

[17] PIDD, J. *Computer Simulation in Management Science*, 4th ed. John Wiley & Sons, 1998. 279p. ISBN 0-471-97931-7.

[18] REISIG, W., ROZENBERG, G. (Eds.) *Lectures on Petri Nets I: Basic Models*, Springer-Verlag, 1998. 681 p. ISBN 3-540-65306-6.

[19] REISIG, W., ROZENBERG, G. (Eds.) *Lectures on Petri Nets II: Applications*, Springer-Verlag, 1998. 477 p. ISBN 3-540-65307-4.

[20] STARKE, P.H. *Petri Netze*, Berlin : VEB Deutscher Verlag der Wissenschaften, 1980.

[21] WEINBERGER, J. Logo, Simulation and Object-Oriented Programming. In *Proceedings of the Third European Logo Conference EUROLOGO'91, Parma, 27-30 August 1991*. Edit. by E. Calabrese. Universita di Parma, p. 219-225.

[22] ZEIGLER, B.P. *Theory of Modelling and Simulation*, Malabar Florida : Robert E. Krieger Publishing Company, 1984. 435 p. ISBN 0-89874-808-9.

[23] Petri Nets World.
http://www.daimi.au.dk/PetriNets/

[24] PetriSim Home Page
http://staff.um.edu.mt/jskl1/petrisim/index.html

[25] On-line Solvers and Simulators
http://staff.um.edu.mt/jskl1/simweb/

[26] IFORS tutORial
http://www.ifors.ms.unimelb.edu.au/tutorial/

[27] Coloured Petri Nets
http://www.daimi.au.dk/CPnets/

[28] Queueing Petri Nets
http://ls4-www.informatik.uni-dortmund.de/QPN/

[29] JavaScript Central
http://devedge.netscape.com/central/javascript/

[30] JavaSim Home Page
http://javasim.ncl.ac.uk/

[31] simjava Home Page
http://www.dcs.ed.ac.uk/home/hase/simjava/

[32] Discrete Simulation
http://staff.um.edu.mt/jskl1/simul.html#Discrete

[33] JSSim Home Page
http://staff.um.edu.mt/jskl1/simweb/jssim/

[34] Author's Home Page
http://staff.um.edu.mt/jskl1/

ABSTRACT

This paper is a modified author's habilitation dissertation. The dissertation is made of a selection of papers together with a text that comments them and puts them into a wider context of computer simulation. The dissertation is divided into two parts that describe the implementation of two discrete simulation tools created by the author.

First part deals with the discrete simulation environment called PetriSim. PetriSim is a tool that contains a graphical editor and a simulator of modified Petri nets that are close to the original Pl/Tr nets. The only additions are generalized test and inhibitor arcs, timed transitions, and a new type of branching transitions. In order to create simulation models, the user has to add code snippets performed at the beginning and at the end of transition firing. In this way it is possible to create simulation models that would otherwise be programmed in simulation languages. All that is needed when using PetriSim are the basic ideas of Petri nets and intermediate Turbo Pascal programming. Various models of single queue queueing systems that can be combined into queueing networks are presented. Though the models are based on PetriSim nets, the results can also be applied when using other types of Petri nets that use the transition firing delay paradigm.

Second part describes the implementation of the JavaScript based simulation package called JSSim that contains declarations supporting discrete event oriented simulation. The models are integrated into HTML documents that are used for the interface between the model and the user. In order to simplify creation of such models as much as possible, special attention was given to direct links between JavaScript instances and HTML screen objects. Several models of various queueing systems were implemented and placed on the web with very encouradging response.

The products can be downloaded from their home pages [24] and [33] that both contain on-line documentation. Contact the author at his home page [34] for more details and/or the copies of the papers dealt with in the dissertation.