Nástroje Diskrétní Simulace

Habilitační práce

Vysoké učení technické v Brně

Fakulta informačních technologií

Jaroslav Sklenář Duben 2003

OBSAH

B Práce s vazbou na JSSim

[B1] Sklenář, J.: *Object Oriented Programming in JavaScript.* Sborník konference "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.35-42.

[B2] Sklenář, J.: *Simulator of Queueing Networks.* Sborník konference "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.127-135.

[B3] Sklenář, J.: *Client Side Web Simulation Engine.* Sborník konference "27th Conference of the ASU Model Oriented Programming and Simulation", Rättvik, 2001, p.1-13.

[B4] Sklenář, J.: *Discrete Event Simulation in JavaScript.* Sborník konference "28th Conference of the ASU The Simulation Languages", Brno, 2002, p.115-121.

Úvod

Předkládaná habilitační práce obsahuje soubor mých vybraných publikací spolu s úvodním textem, který je zařazuje do širšího kontextu vycházejícího z historie diskrétní počítačové simulace (dále jen simulace), která je předmětem mého odborného zaměření. V úvodu také odkazuji na některé své práce které souvisejí se simulací, ale nezapadají do dvou oblastí kterými se dále zabývám podrobněji v kapitolách 1 a 2. Za těmito kapitolami následuje seznam mých publikací a některých nepublikovaných prací, tříděný podle odborných oblastí a času. Následuje seznam použitých odkazů na práce jiných autorů, které rozlišuji rozdílným formátem. Vybrané práce jsou zařazeny v přílohách.

Existuje mnoho neformálních i formálních definic simulace. Jednou z prvních a velmi výstižných je definice Dahlova (Dahl 1966): "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. Simulace je proto téměř stejně stará jako samočinné počítače. 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 – čas. Řízení časové osy je relativně náročný programátorský problém (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. Jedním z prvních byl jazyk Simscript vytvořený firmou Rand Corp. začátkem 60. let (Markowitz 1963). Právě tento jazyk byl základem jazyka Simscript T200 který jsem implementoval v rámci výzkumného projektu "Modelování a Simulace" na katedře matematiky FE VUT v 70. letech - [9], [10]. Jazyk Simscript T200 byl obohacen o možnost automatizovaných repetičních výpočtů, což v původním jazyce možné nebylo - [3]. Kompilátor a "run-time" rutiny byly napsány v jazyce symbolických adres APS. Přesto, že implementace simulačního jazyka na počítači s vnitřní pamětí 128KB dnes vypadá směšně, v tomto jazyce byla vytvořena řada malých a středně složitých demonstračních modelů a také několik praktických simulačních studií. Později jsem používal jazyk Simscript II implementovaný ve Výzkumném ústavu matematických strojů na počítačích řady JSEP. To umožnilo realizaci celé řady simulačních studií komunikačních systémů a částí počítačových sítí v rámci spolupráce mezi Katedrou telekomunikací FE VUT a Výzkumným ústavem spojovací techniky v Praze a Ústavem aplikované kybernetiky v Bratislavě. Výsledky byly shrnuty v pracech [20] až [24], [48] a [52]. V 80. letech se pak podařilo získat implementaci jazyka Simula pro počítače řady JSEP. Pro praktickou simulaci tak bylo možné použít objektově orientované programování v

[1] Skutečnost, že se v této práci 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.

systémové třídě Simulation tohoto jazyka. Bylo to bohužel již v době, kdy střediskové počítače dožívaly a začala simulace na osobních počítačích.
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]. Práce byly zaměřeny jednak na analýzu výrazových prostředků již existujících a také na vývoj nástrojů nových. Do první kategorie zapadá využití tehdy módní Obecné teorie systémů. Typickým příkladem je dnes již klasická kniha (Zeigler 1976). Touto problematikou jsme se také zabývali v rámci výše zmíněného výzkumného projektu na Katedře matematiky FE VUT. Teoretické výsledky byly publikovány v pracech [1], [2], praktickými aplikacemi se pak zabývají práce [4], [5], [25], [26], [50] a [57]. Do oblasti metodiky simulace lze také zařadit práci formalizující náhodné zobrazení - [6] a využití fuzzy čísel při simulaci s nepřesnými vstupními daty - [7], [8]. Formální výrazové prostředky nejsou používány pouze při simulaci. Vznikla celá řada prostředků pro popis technických systémů a programů. Jedním z nich je jazyk SDL (Specification and Description Language), určený k popisu telekomunikačních systémů, zejména elektronických ústředen s programovým řízením. Jazykem SDL se zabývají práce [54], [55] a [56].
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, 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) – (Forrester 1961) 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 - (Kelton 2002) nebo Extend - (Imagine That, Inc.), který lze použít i pro simulaci spojitou, kde nevadí jeho primitivní integrační metody. 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 z hlediska výuky. Např. v rámci předmětu "Modelling and Simulation", který učím v prvním ročníku oboru "Statistics and Operations Research" na maltské univerzitě, studenti zvládnou během jednoho semestru Extend a Powersim do té míry, že jsou schopni vypracovat projekt, který obsahuje netriviální simulační model. 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

[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 (Pidd 1998) používají např. termín "Visual Interactive Modelling System" (VIMS). Všeobecně akceptovaný termín neznám.

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 např. Extend obsahuje jazyk zvaný Modl (podobný jazyku C), kterým lze naprogramovat 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. Podobně v prostředí Arena lze programovat uživatelské bloky v simulačním jazyce Siman. Další nevýhodou je statika topologie. Jen velmi těžko se vytváří modely, kde bloky vznikají a zanikají, což se samozřejmě netýká transakcí. 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 - jako např. Extend - práci s hierarchickými bloky, které jsou uvnitř sítěmi a to na více úrovních. Často je pak výhodnější model naprogramovat. Bylo by jistě zajímavé zjistit, jaké procento simulačních modelů je ještě dnes výhodné programovat. Mnoho se jich totiž programuje v podstatě ze setrvačnosti, nebo z důvodu nedostupnosti některého z grafických prostředí.
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. 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 JSSim popsaný v kapitole 2.
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.

1 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. Tato kapitola je úvodem a komentářem k pracem zařazeným v příloze A.

1.1 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). Jedna z několika ekvivalentních definic je uvedena v práci [A2], kde je použit původní název Marked Petri Net. 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. Úvod do této teorie obsahují např. monografie (Peterson 1981) a (Starke 1980), rozšířeními se zabývají např. články (Diaz 1982), (Murata 1989) nebo články ve sborníku (Reisig 1998a), 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 stránkách Petri Nets World na Aarhuzské Universitě v Dánsku (*http://www.daimi.au.dk/PetriNets/*). V češtině je k dispozici práce (Češka 1994). Existují také desítky programů které lze označit jako prostředí pro práci s Petriho sítěmi. Stránky Petri Nets World obsahují databázi těchto programů která obsahuje i PetriSim.

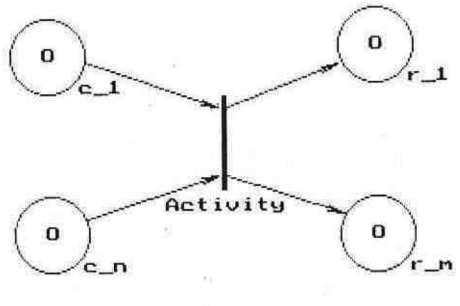
Aplikace Petriho sítí - viz např. sborník (Reisig 1998b) - lze rozdělit do dvou kategorií: analytické a simulační. Analytické aplikace založené na analytických schopnostech Petriho sítí postupují následovně: zkoumaný systém je vyjádřen Petriho sítí, ta je analyzována a podle výsledků pak lze usuzovat 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. Použití Petriho sítí k analýze výrobního systému (konkrétně stáčírny nápojů na Maltě) je věnována práce [40]. 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 analyzovat, nebo dokázat správnost nějakého protokolu.

1.2 Petriho sítě a simulace

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). Čas lze definovat ve vztahu k místům, přechodům, nebo i hranám. Viz popis např. v knize (Desrochers 1995). Sítě v PetriSimu jsou založeny na zpoždění v přechodech, což je nejčastější přístup. Mechanismus je tento: provedení (firing) přechodu trvá určitou dobu. Při zahájení provedení jsou značky vyjmuty ze vstupních míst, při ukončení provedení jsou značky umístěny do výstupních míst. Trvání provedení definuje uživatel, může být konstantní, závislé na stavu sítě nebo náhodné[7]. Trvání provedení přechodu lze použít k řízení časové osy u všech tří základních principů časování disktrétních simulačních modelů.

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. 1, který byl stejně jako všechny sítě v této práci nakreslen grafickým editorem PetriSimu. Posun v čase je realizován trváním provedení.



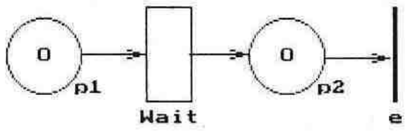
Obr.1: Model podmíněné aktivity Petriho sítí.

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.

[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. knihy (Haverkort 1998) a (Lindemann 1998).

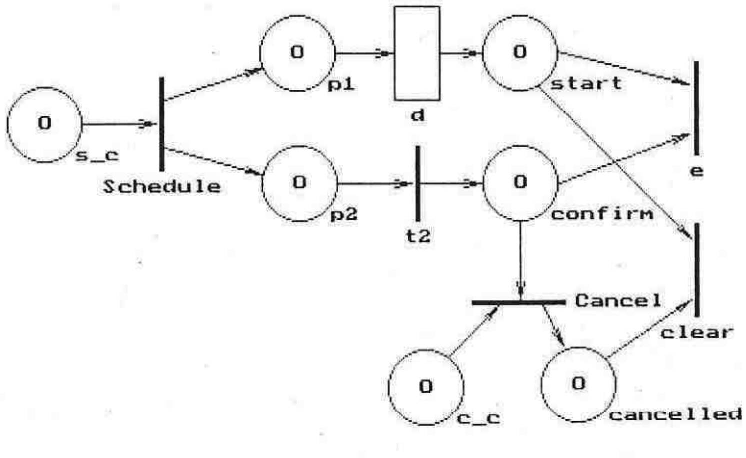
Události

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* $(t_1\text{-}t)$" 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. 2, kde provedení přechodu *Wait* trvá *d*. Přestože kreslení Petriho sítí není zcela normalizované, časované přechody se většinou odlišují od okamžitých přechodů tak, jak je to znázorněno na obr.2. Aktivace události = přechodu *e* pak následuje okamžitě.



Obr.2: Plánování události Petriho sítí.

Zrušení naplánované události "*cancel e*" je již složitější, částečné řešení je uvedeno na obr. 3, kde přechod *Schedule* zahájí plánování. Jediný časovaný 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.

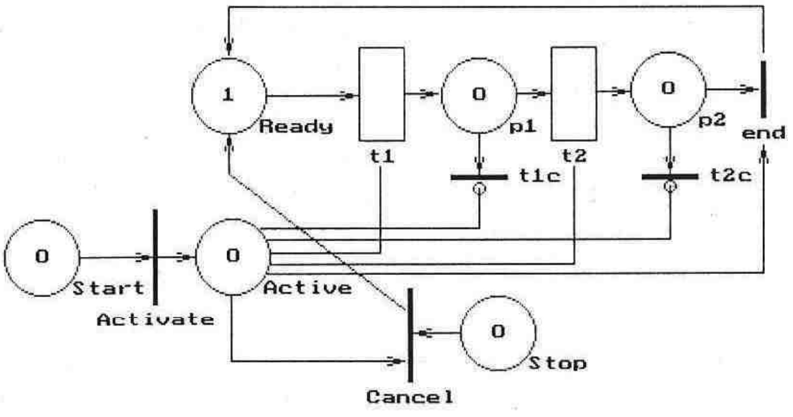


Obr.3: Plánování události s možností zrušení.

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.

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 časovaným přechodem je evidentní – viz např. přechod *Wait* na obr. 2.
Activate P je nejjednodušší forma příkazu, který aktivuje proces *P*. Řešení je naznačeno na obr. 4, 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. Síť na obr. 4 používá všechny tři typy hran, které lze v PetriSimu vytvořit. Tzv. testovací hrana (testing arc), kreslená jako čára bez šipky nebo kroužku, testuje zda místo obsahuje daný počet značek. Pokud ano, a pokud jsou splněny ostatní podmínky provedení přechodu, pak je provedení přechodu zahájeno bez vyjmutí značek. Testovací hrany na obr. 4 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.



Obr.4: Aktivace a ukončení procesu.

P.cancel(R) je provedení metody *cancel* procesu *P*, která deaktivuje (přerušuje provádění) nějakého jiného procesu *R*. K přerušení procesu je na obr. 4 použit přechod *Cancel*, který také uvede proces do počátečního stavu. Vzhledem k nepřerušitelnosti provádění přechodu se opět jedná o řešení částečné. Proces na obr. 4 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 tzv. inhibiční hrany (inhibitor arc) zakončené kroužkem. Ty testují, zda je místo prázdné. Pokud ano, přechod může být proveden. Přechod *t1c* resp. *t2c* tak vyjme značku z místa *p1* resp. *p2*, pokud je místo *Active* prázdné. V PetriSimu mohou být všechny tři typy hran násobné. U inhibičních hran to znamená, že přechod může být proveden, pokud vstupní místo obsahuje méně značek, než je násobnost hrany. Jedná se tedy o negaci podmínky testované normální hranou, která je ukončena šipkou.

1.3 Původní textové prostředí – [A1]

Předchůdce PetriSimu, nazvaný pracovně PETSIM, byl naprogramován v jazyce Logos (Weinberger 1991), 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 jsem neměl přístup k jinému objektovému jazyku. Po příchodu Object Pascalu v rámci Turbo Pascalu verze 5.5 jsem pak používal tento jazyk, PetriSim ve dnešní verzi 4 je naprogramován v Turbo Pascalu resp. Borland Pascalu 7.
Článek [A1] nejdříve ukazuje jak reprezentovat znalost o sítích jako takových, místech a přechodech pomocí tříd jazyka Logos. Atributy a metody těchto tříd jsou uvedeny na str. 389-392. Následuje popis prostředí pro práci s Petriho sítěmi. Pomocí mechanismu dědění (inheritance) pak byly definovány tzv. barvené sítě (Coloured Petri Nets)[8] – str. 394 a časové sítě – str. 395, se krerými už bylo možné provádět jednoduchou simulaci. Význam článku [A1] vidím v tom, že ukazuje využití technik objektově orientovaného programování k vytvoření a obohacení jistého uživatelského prostředí. Bylo to v době, kdy používání těchto technik bylo s výjimkou Simuly teprve v začátcích.
Poznamenejme, že jazyk Logos měl jednu vlastnost, kterou klasické kompilované objektové jazyky jako např. Simula nebo Java nemají – možnost modifikovat (individualizovat) instance tříd na úrovni kódu. V klasických objektových 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 (XP) a při ukončení provedení (YP) přechodu – viz str. 395 dole. Tyto metody pak bylo možné definovat pro každý přechod zvlášť podle potřeby simulačního modelu spolu s metodou DUR, která vracela prodlevu provedení.
Vzhledem k textové komunikaci a dalším omezením použitého Loga nepřicházelo v úvahu použití PETSIMU k praktické simulaci. Byl použit pouze k výuce Petriho sítí. Přílohou článku [A1] je uživatelský manuál, který ukazuje jednoduchost práce.

1.4 Metodika simulace časovými sítěmi – [A2]

Článek [A2] shrnuje zkušenosti získané používáním PetriSimu verze 2. Na příkladu sítě hromadné obsluhy tvořené třemi uzly – viz obr. 1 je ukázán celý postup. Síť hromadné obsluhy je nejdříve vyjádřena časovou sítí – viz. obr. 2, ke které je pak doplněn kód v Pascalu, jehož podstatná část je zařazena v příloze. Jsou vyjmenovány procedury, které doplňuje uživatel (str. 62) a standardní procedury PetriSimu (str. 63), které podporují uživatelské programování. Konkrétně se jedná o generování prodlevy

[8] Poznamenejme, že se zde jedná o původní jednoduché barvené sítě definované v knize (Peterson 1981). Pojmem barvené sítě 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 *http://www.daimi.aau.dk/CPnets/*.

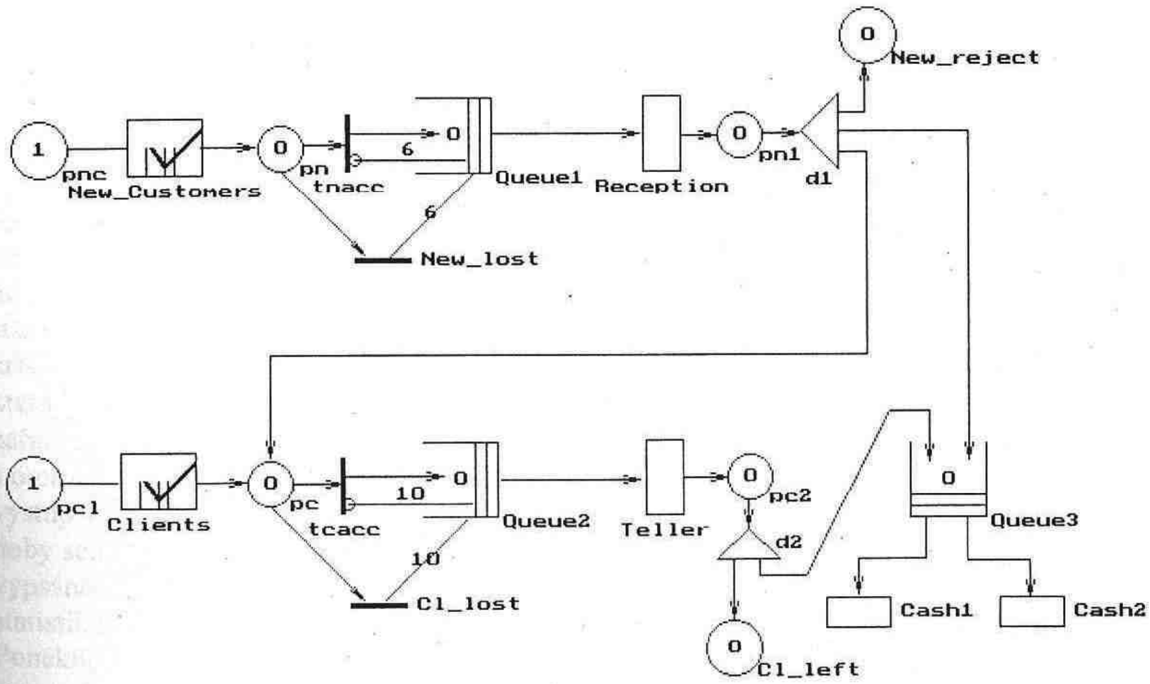
při realizaci přechodu, což je základní nástroj práce s časem, získání označení místa a času modelu a možnost modifikace označení. Poslední dvě procedury zobrazují síť a ukončují experiment. Následuje vysvětlení, jak pracovat se statisticky monitorovanými objekty, s náhodnými čísly a s uživatelskými 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 – byl také popsán v článku [11]. 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 už je automaticky poskytována statistika o době setrvání značek v místech, 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.

1.5 Grafické prostředí PetriSimu – [A3]

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. Viz obr. 2, který je kopií obrazovky těsně před spuštěním editoru. Editor si vybírá uživatel sám, standardně je používán editor *edit.com*, který je součástí Windows. Obr. 2 je modelem paketové sítě tvořené čtyřmi vstupními linkami, vnitřním zpracováním s omezenou kapacitou paměti a pěti výstupními linkami. Samotný model a jeho vytvoření je popsáno v článku [35].

1.6 Použití mechanismu dědění – [A4]

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. Diagramy na str. 176 naznačují, jak byly s minimálními zásahy do stávajícího kódu implementovány inhibiční hrany a rozšíření časových sítí s cílem simulovat sítě hromadné obsluhy[9]. Většinou se jedná pouze o kreslení míst a přechodů různými názornými ikonami – viz str. 177. Pouze v jednom případě byl definován nový typ přechodu, který se liší funkcí. Jde o větvení pracovně nazvané "branching" – viz ikony na str. 178. Tento přechod pracuje tak, že po ukončení provedení přechodu je značka umístěna pouze do jednoho výstupního místa. Toto místo je vybráno uživatelským kódem, pokud ne, je vybráno náhodně, všechna výstupní místa se stejnou pravděpodobností. Použití těchto přechodů k náhodnému nebo deterministickému větvení po ukončení obsluhy je evidentní. Před zavedením těchto přechodů bylo nutné programovat větvení pomocí nebezpečné modifikace označení výstupních míst. V poslední kapitole je popsán simulační model banky, jejíž činnost byla vyjádřena sítí hromadné obsluhy se třemi frontami, z nichž dvě mají omezenou kapacitu, čili může dojít ke ztrátě zákazníků. Ve dvou případech následuje po obsluze náhodné větvení – viz obr. 1 článku [A4]. Na obr. 5 je ukázáno použití zobecněných testovacích hran, které v době napsání článku [A4] ještě nebyly k dispozici.



Obr.5: Model banky časovou sítí.

[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 *http://ls4-www.informatik.uni-dortmund.de/QPN/.*

Např. fronta reprezentovaná místem *Queue1* má kapacitu 6 zákazníků. Inhibiční hrana umožňuje provedení přechodu *tnacc* (zařazení do fronty) pokud je ve frontě zákazníků méně. Naopak testovací hrana umožňuje provedení přechodu *New_lost* (ztráta zákazníka) pokud je ve frontě již 6 zákazníků. Bez testovacích hran bylo nutné ke každé frontě doplnit místo, jehož počáteční značení je kapacitou fronty – viz obr. 1 článku [A4]. Počet ztracených zákazníků je rovný počtu provedení přechodu *New_lost*, který lze snadno získat po ukončení simulace.
Model banky není složitý, ale také není triviální. Přesto veškerý kód, který bylo nutno napsat (mimo vyhodnocení simulace), je tvořen asi 13 řádky, jak je uvedeno v příloze 2 článku [A4], kde je také vidět programování větvícího přechodu nastavením jeho atributu *Branch*. Příloha 3 shrnuje výsledky dvou simulačních experimentů, jejichž cílem bylo nalézt optimální počet pracovišť. Numerické parametry a další detaily jsou uvedeny v článku, experiment délky 10000 minut trvá asi 3 sekundy (PII, 300MHz).
Popsaný mechanismus dědění je otevřený, z diagramů je zřejmé, že PetriSim vždy pracuje s rozhraním třídy *User*. Před tuto třídu lze zařadit libovolnou posloupnost nadtříd, které dále obohacují vlastnosti časových sítí. Z uživatelského hlediska se jedná o specializaci v globálním smyslu pro všechna místa a všechny přechody. Lokální specializace je pak dosaženo pomocí úseků kódu pro jednotlivé přechody. Jako ukázka tohoto postupu byly vytvořeny sítě, kde místa jsou zobrazena jako text s různou barvou podle toho, zda místo je nebo není prázdné. Viz uvítání na domovské stránce PetriSimu na *http://staff.um.edu.mt/jskl1/petrisim/index.html*.

1.7 Simulace sítí hromadné obsluhy – [A5]

Sítě hromadné obsluhy jsou velmi často simulovány, protože analytické metody většinou buďto neexistují, nebo jsou založeny na nesplněných předpokladech. 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. Nejdříve je simulován systém G/G/1 – viz obr. 1, kde je vidět že k jeho vytvoření stačí doslova 3 řádky kódu. Dva na generování prodlevy mezi příchody zákazníků a trvání obsluhy, třetí je aktivací procedury která vypisuje standardní výsledky simulace systému G/G/1 do zadaného souboru. Uživatel musí také zadat jméno místa a jméno přechodu, protože ze struktury sítě nelze odvodit, která dvojice místo-přechod tento systém tvoří. Zobecnění na vícekanálovou obsluhu zařazením více obsluh = přechodů je evidentní. Následuje model systému G/G/1/K s omezenou kapacitou tak, jak použit výše na obr. 5. Obr. 2 článku ukazuje standarní výstup PetriSimu. 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.
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. 3 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, četnost příchodů je $(M - n)\lambda$. Obr. 3 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. 3 je vidět, jak je střední doba rozdělení odvozena od označení místa *Population*, které obsahuje zákazníky mimo obsluhu. Následující odstavec se zabývá modelováním systémů 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. Příchod náhodné dávky je ukázán na str. 406.
PetriSim neumožňuje přiřadit značkám prioritu, nebo je dělit do tříd. Na obr. 4 je model obsluhy se dvěma prioritami zákazníků. Ve skutečnosti jedna fronta je modelována dvěma frontami pro dvě různé priority (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. Vyjmutím inhibiční hrany by se jednalo o obsluhu zákazníků dvou různých tříd stejné priority (ne zcela přesně, protože výběr obsluh by byl náhodný).
Vytvoření sítě propojením výše uvedených základních uzlů je evidentní, náhodné větvení je realizováno již dříve popsaným větvícím přechodem. Na obr. 5 je jednoduchá síť se dvěma frontami, která je pro porovnání na obr. 6 modelována pomocí Extendu. Stejná síť byla také modelována Arenou a simulátorem sítí v JavaScriptu popsaném v příští kapitole. V tabulce je porovnána rychlost těchto čtyř simulátorů.
PetriSim se osvědčil jako nástroj pro práci s Petriho a časovými sítěmi pro potřeby výuky a výzkumu. Je k dispozici na *http://staff.um.edu.mt/jskl1/petrisim/index.html.*

2 JSSim

JSSim (JavaScript Simulator), jehož jsem autorem, je souborem deklarací v jazyce JavaScript, které tvoří prostředí pro diskrétní simulaci orientovanou na události. JSSim vznikl postupným zobecněním funkcí použitých při vytváření simulačních modelů zabudovaných do dokumentů napsaných v jazyce HTML. Tato kapitola je úvodem a komentářem k pracem zařazeným v příloze B.

2.1 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) (Darnell 1998) 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 (Eckel 1998) a JavaScript (Flanagan 1998). 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).
Tyto skutečnosti jsou důvodem, že podle některých autorů je 80% aplikací na webu psáno v JavaScriptu - (Eckel 1998). 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

[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 *http://developer.netscape.com/docs/manuals/js/client/jsguide/index.htm.*

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). Dále komentované práce byly vybrány tak, aby bylo zřejmé, jaké problémy 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.

2.2 Objektově orientované programování v JavaScriptu – [B1]

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). Detailní popis tohoto objektového modelu zde neuvádím, protože je popsán v článku [B1] – viz kap. 2, 3 a 4. Po bližším seznámení 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). Kap. 5 článku [B1] ukazuje, jak lze 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 – viz kap. 1.3. Je to dáno tím, že v JavaScriptu je text funkce považován za hodnotu na kterou lze následně použít operátor volání "( )" – viz vytvoření metody *shift* na str. 37. 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

[13] Např. *JavaSim*, vytvořený na University of Newcastle upon Tyne (*http://javasim.ncl.ac.uk/*) 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 (*http://www.dcs.ed.ac.uk/home/hase/simjava/*).

[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 objektového programování (encapsulation).

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 objektových jazycích orientovaných na třídy. Tento přístup znamená používat pouze metody uložené v prototypech - viz kód na str. 38, které jsou shodné pro všechny instance vytvořené daným konstruktérem. Funkce použitá jako konstruktér, která vytváří hodnotové vlastnosti instancí, tak nahrazuje deklaraci třídy v kompilovaných jazycích.
Druhým základním principem objektového 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 – viz str. 39. 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, po kterém může následovat vytvoření dodatečných metod podtřídy. Na str. 40 je funkce *inherit*, která kopíruje metody (ne hodnotové vlastnosti) a příklad jejího použití. K dědění hodnotových vlastností, které jsou vytvářeny konstruktérem, lze použít techniku ukázanou na str. 39. 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. Takto lze přidat vlastnost nadtřídy s tím, že změna se automaticky promítne do všech podtříd stejně jako u klasických objektových jazyků. Obě techniky (dědění metod a vlastností) umožňují vícenásobné dědění (multiple inheritance). V příloze článku [B1] je příklad vytvoření třídy pro FIFO fronty, která dědí z obecné fronty *GenQueue*, která implementuje základní vlastnosti všech front, a z fronty *StatQueue*, která implementuje statistické sledování délky. Vlastní metody pak implementují operace dané FIFO mechanismem[17].
Třetím základním principem objektového 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 objektových jazyků orientovaných na třídy dosaženo pomocí složitého mechanismu nazvaného "late binding"[18]. 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*.

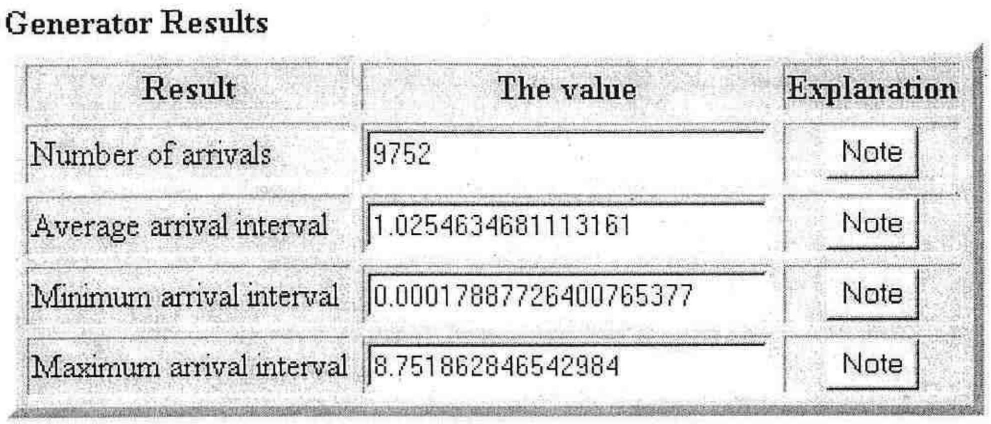
[15] Umožňuje-li jazyk vícenásobné dědění (multiple inheritance) pak může být nadtříd více.
[16] Poznamenejme, že ne všechny objektové jazyky používají pojem "class". Např. Object Pascal nebo simulační jazyk Modsim používají název "object". Princip je však shodný.
[17] V příloze článku [B1] je původní implementace front, kde hostitelskou datovou strukturou bylo pole. JSSim má nyní fronty implementované klasicky řetězeným seznamem objektů.
[18] Jedná se totiž v podstatě o porušení principu typovaných proměnných.

Článek [B1] končí popisem techniky, která nemá obdobu v jiných objektových jazycích. Programované dědění totiž umožňuje kopírovat pouze některé vybrané metody a vytvářet tak "podtřídy" které jsou zjednodušením "nadtřídy". Na str. 41 je zobecněná funkce *inherit*, která kopíruje pouze vyjmenované metody. Této techniky jsem použil při 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.

2.3 Simulátor sítí hromadné obsluhy – [B2]

Existuje pouze jediný způsob, jak ověřit funkčnost přímého zabudování simulačního modelu do webovské stránky. Napsal jsem proto několik modelů, které jsou k dispozici na *http://staff.um.edu.mt/jskl1/simweb/*. 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 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].
Po úvodu s motivací a předpoklady použití modelu následuje nástin řešení. JavaScript přímo nepodporuje nakreslení sítě myší tak, jak to umožňují již zmíněné interaktivní simulační prostředky jako je Extend 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 v kap. 4, deklarace generátoru je zařazena v příloze 1. 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

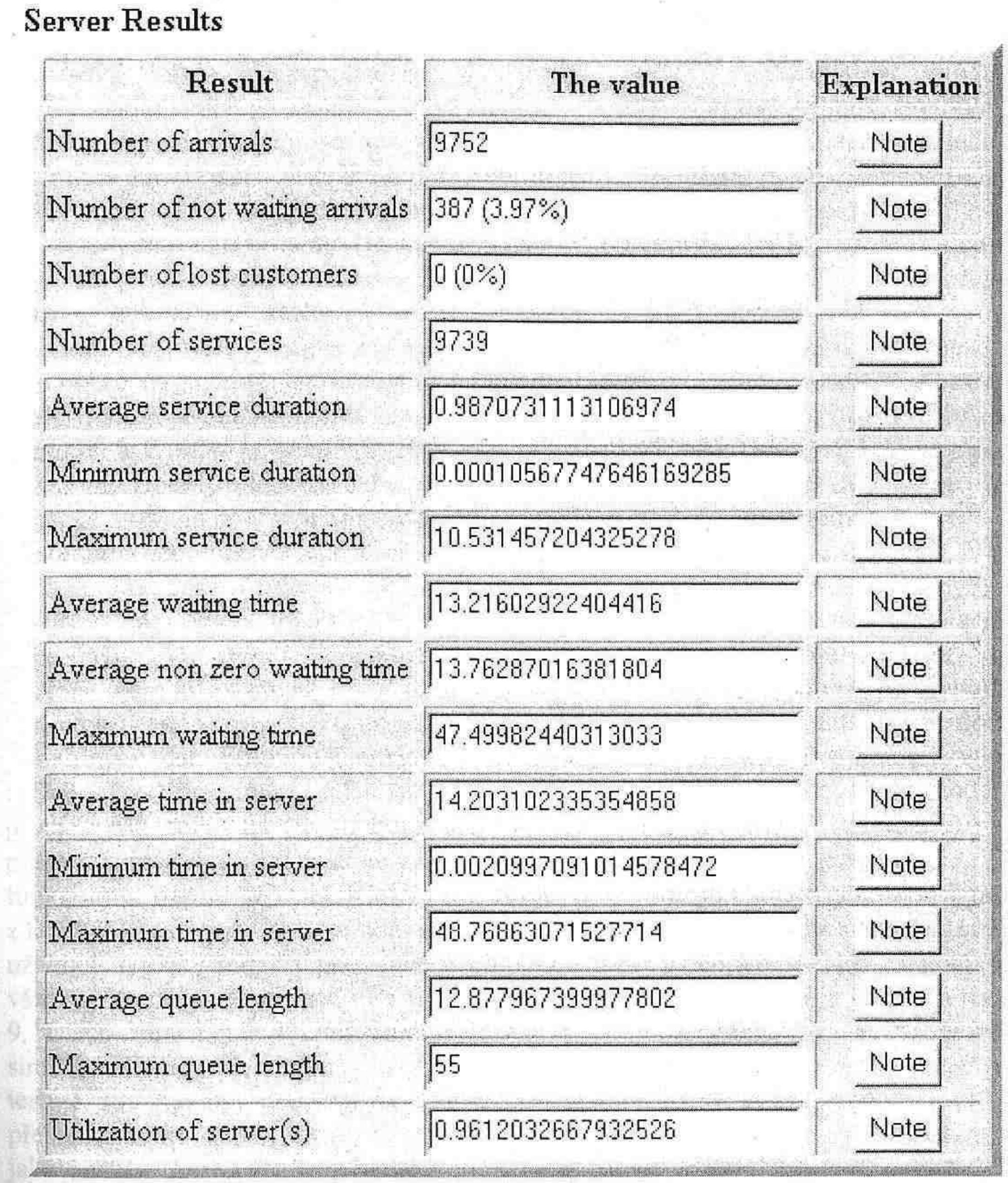
kapacitou a organizací fronty (FIFO nebo LIFO), počtem shodných obslužných kanálů, náhodným rozdělením trvání obsluhy a stejně jako generátor náhodným směrováním. Pro odchod zákazníka ze sítě je rezervováno číslo cílové stanice 0. Pro generátory a stanice poskytuje model výsledky ukázané na obr. 6 a obr. 7 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.6: Výsledky simulace generátoru.

Po popisu tříd síťových objektů následuje stručný popis řídícího programu, kterému je věnována příští kapitola. Následuje příklad simulace jednoduché sítě, která je podobná modelu banky modelované Petriho sítí v článku [A4]. Doby obsluh však byly zadané tabulkami distribučních funkcí – viz přílohu 3. Jsou porovnány doby výpočtu pro různé prohlížeče a různé délky experimentu na specifikovaném počítači s jedinou aktivní úlohou. Je zřejmé, že sítě střední velikosti lze simulovat v rozumném čase desítek sekund nebo několika minut.
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.6: Výsledky simulace obsluhy.

2.4 Řídící program a podpůrné prostředky – [B3]

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[19]. 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. Detaily jsou uvedeny v článku [B3]. Byla definována třída *heap*, jejíž instance nazvaná po vzoru Simuly *SQS* (Sequencing Set) je kalendářem událostí[20].
Následující kapitola shrnuje funkce, které řídicí program poskytuje uživateli. Vedle plánování události a zrušení naplánované události je to inicializace řídicího programu, po které následuje inicializace modelu uživatelem a nastartování simulačního experimentu. Je rovněž k dispozici konstruktér *evnotice* záznamu události, který vytvoří jedinou vlastnost, kterou je čas výskytu události. Uživatel po vytvoření záznamu typicky vytvoří vlastnosti další, např. pro rozlišení typů událostí a uložení libovolných dat. Přílohou článku [B3] je úplný kód jednoduchého modelu včetně HTML. Použití záznamu události je zřejmé např. ve funkci *arrival*, která modeluje příchod zákazníka na str. 9. Použití uživatelem vytvořené vlastnosti *eventtype* pro přepínání mezi typy událostí je zřejmé ve funkci *eventroutine* na téže stránce. Tuto funkci musí napsat uživatel. Řídicí program ji opakovaně po vyjmutí prvního záznamu z kalendáře aktivuje, záznam události je předán jako argument. Další akce programuje uživatel, typicky půjde v prvé řadě o přepínání mezi jednotlivými typy událostí, čili všechny modely mohou mít tuto funkci prakticky shodnou, tak jak je uvedena na str. 9. Další funkce pak budou implementovat jednotlivé události jak je to běžné u simulačních jazyků tohoto typu. Uživatel musí napsat další dvě funkce. *finish_run* testuje zda má být ukončen experiment. Verze na str. 9 pouze testuje, jestli čas překročil délku experimentu. Lze naprogramovat jiné složitější podmínky ukončení jako např. obsloužení určitého počtu zákazníků apod. Uživatel musí dále naprogramovat samotné spuštění simulace a vyhodnocení experimentu po jeho ukončení. Funkce *simulation* na str. 11 postupně inicializuje řídicí program, inicializuje proměnné modelu, plánuje příchod prvního zákazníka a první poruchu a

[19] 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 modely v Pascalu na *http://staff.um.edu.mt/jskl1/models2.zip*.
[20] 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 (Fries 2001), který tvrdí, že seznam je nakonec vždy alespoň stejně rychlý jako stromy.

poté předá řízení řídicímu programu. Po návratu je experiment vyhodnocen funkcí *evaluation*. Podobnost s programováním v Simscriptu není náhodná. Funkce *simulation* je aktivovaná stisknutím tlačítka *Run*, jak je patrné z HTML kódu na str. 12 uprostřed. Kód v příloze článku [B3] je ukázkovým modelem z manuálu řídicího programu: *http://staff.um.edu.mt/jskl1/simweb/engine/engman.html*.
Simulační jazyky nebo knihovny poskytují mimo řízení časování modelu celou řadu dalších funkcí. Jejich výčtem začíná další kapitola článku [B3]. Podrobněji je naznačena implementace front a statisticky monitorovaných proměnných. K jejich implementaci je použita technika Simscriptu zvaná "left monitoring" navržená již ve článku (McNeley 1968)[21] 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. Při vyhodnocení experimentu pak lze přímo volat metody, které vrací průměr, směrodatnou odchylku, apod. – viz použití metody *average* funkcí *evaluation* na str. 10. JSSim obsahuje definici dvou tříd pro statisticky monitorované proměnné v čase, kdy průměr je vypočítán z časového integrálu (délka fronty) a pro proměnné jejichž statistika závisí pouze na přiřazených hodnotách (doba čekání ve frontě). Stručným popisem příkladu simulace systému hromadné obsluhy s poruchami článek končí.

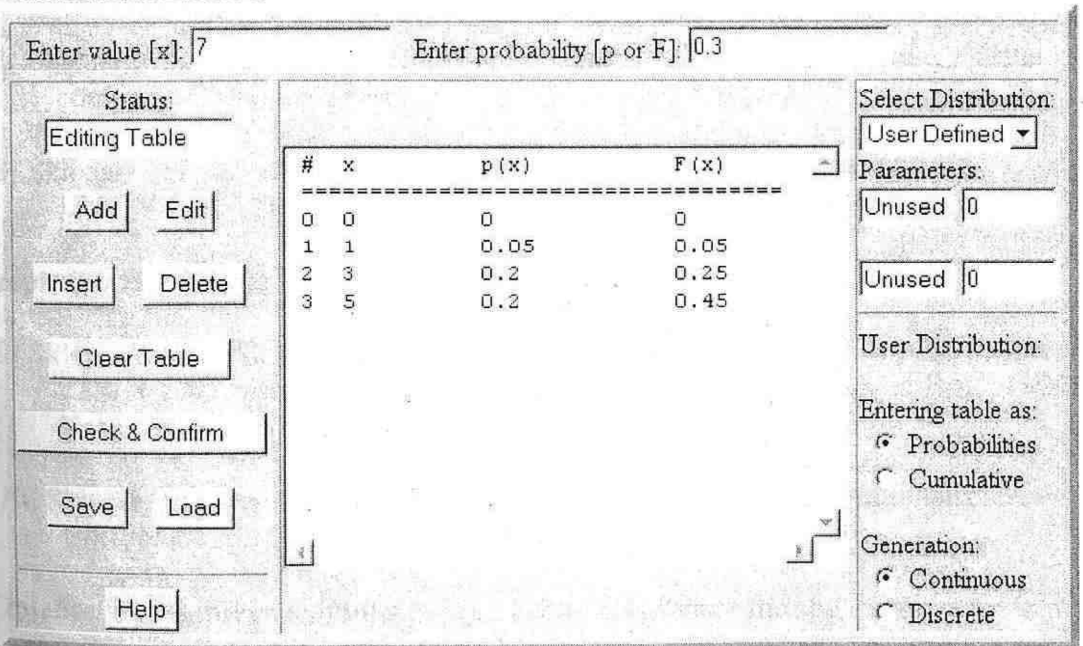
2.5 Diskrétní simulace v JavaScriptu a vazba na HTML – [B4]

Článek [B4] byl zařazen proto, že se ve své druhé části zabývá využitím úzké vazby mezi kódy v JavaScriptu a v HTML s cílem zjednodušit co nejvíce výtváření dokumentů obsahujících simulační modely. V první části jsou shrnuty základní údaje o řídicím programu, jsou uvedeny příklady funkcí, které píše uživatel. Po shrnutí podpůrných prostředků simulace je v kap. 4 naznačeno, jak konkrétně realizovat vazbu mezi popisem struktury dokumentu v HTML a JavaScriptem, který implementuje simulační model a jeho ovládání uživatelem.
První příklad ukazuje techniku validace vstupu nezáporné celočíselné hodnoty. Správná hodnota je uložena do globální proměnné *numofcserf*, která obsahuje počet obsloužených zákazníků, po kterém je ukončen experiment (netestováno při nule). Jakákoliv chyba je ohlášena, textové pole je pak přepsáno původní hodnotou. Tlačítko *Help* zobrazí přesný význam hodnoty zadávaného parametru, což je důsledně dodržováno u všech vstupů. Všechny modely rovněž umožňují zobrazit ve zvláštním okně nápovědu, která je návodem, jak model používat včetně všech předpokladů. Žádná další uživatelská dokumentace není třeba.
Následující příklad ukazuje přímou vazbu mezi instancí třídy a HTML objekty dokumentu. Metoda *update* instance *intstat* statisticky monitorované veličiny je aktivována při každém generování prodlevy mezi příchody zákazníků – viz funkci *simulation* na str. 118, která plánuje příchod prvního zákazníka. Na str. 119 je HTML kód, který zobrazuje část tabulky se statistikou intervalů mezi příchody, na str. 120 je výřez okna prohlížeče (v tomto případě Internet Exploreru 6, ale rozdíly mezi

[21] 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.

prohlížeči jsou nepatrné). Aktualizace hodnot po ukončení simulace je provedena jedinou aktivací metody *scrupdate*, jejíž aktivace a implementace je na str. 120. 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. Na str. 120 je také metoda *winupdate*, která dynamicky vygeneruje do zadaného okna HTML kód, který zobrazí statistické hodnoty jako standardní text. Všechny modely umožňují generovat výsledky v textové formě ve zvláštním okně. Lze je pak kopírovat do jiných dokumentů, např. do MS Wordu. Text na konci str. 120 byl takto zkopírován z okna prohlížeče. 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. Na obr. 8 je 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.

Distribution of Intervals



Enter value [x]: 7
Enter probability [p or F]: 0.3
Status: Editing Table
Select Distribution: User Defined
Parameters: Unused 0
Unused 0
Entering table as: (• Probabilities) Cumulative
Generation: (• Continuous) Discrete

```
#  x     p(x)         F(x)
===================================
0  0     0            0
1  1     0.05         0.05
2  3     0.2          0.25
3  5     0.2          0.45
```

Obr.8: Vstup parametrů náhodné proměnné.

Popisované simulační modely jsou také zahrnuty do projektu tutORial organizace INFORS (International Federation of Operational Research Societies) koordinovaného universitou v Melbourne, viz *http://www.ifors.ms.unimelb.edu.au/tutorial/*.

Seznam publikací a některých nepublikovaných prací

SIMULACE

Metodologie a teorie diskrétní simulace

[1] Sklenar, J.: *Application of General Systems Theory in Computer Networks Simulation.* Proceedings of the conference "Computer Networks'80", Havířov, 1980, p.139-146.

[2] Sklenar, J: *Apparatus for Formal Description of Discrete Systems.* International Journal of General Systems, New York, 1981, Vol. 7, No. 4, p.225-234.

[3] Sklenar, J.: *Repetition Computations in the SIMSCRIPT Language.* Proceedings of the "International Conference on Modelling and Simulation", Valle de Chevreuse, France, 1982, p.3-4.

[4] Sklenar, J.: *Use of Systems Theory in Computer Networks Simulation.* Informačné Systémy, 1982, No. 1, p.47-58.

[5] Sklenar, J.: *Methodology of Discrete Systems Simulation.* Proceedings of the "International Conference Simulation of Systems", Praha, 1983, p.61-66.

[6] Sklenar, J.: *Definition of a Random Mapping by Means of the Probability Theory.* Proceedings of the Technical University of Brno, 1987, Vol B-118, p.39-49.

[7] Sklenar, J.: *Use of Fuzzy Numbers in Simulation of Systems.* Proceedings of the conference "Simulation of Systems", Opava, 1987, p.38-41.

[8] Sklenar, J.: *Fuzzy Numbers.* Proceedings of the Technical University of Brno, 1988, Vol. B-121, p.267-273.

Implementace jazyka Simscript T200

[9] Sklenarova, K., Sklenar, J.: *The SIMSCRIPT Language and its Compiler for the TESLA 200 Computers.* Proceedings of the Technical University of Brno, 1975, Vol.B-56, p.237-243.

[10] Sklenar, J.: *Discrete Simulation Language SIMSCRIPT T200.* Automatizace, 1981, No. 4, p.108-110.

Objektově orientovaná simulace, prostředky a aplikace simulace na internetu

[11] Sklenar, J: *A Turbo Pascal Unit SimSet.* ASU Newsletter, 1995, Vol. 23, No. 1, p.1-17.

[12] Sklenar, J.: *Use of Internet Facilities in Education.* Proceedings of the "IFIP Conference Human-Computer Interaction INTERACT'97", Sydney, 1997, abstract of the poster, p.569-570.

[13] Sklenar, J.: *Prototype Oriented Simulation in JavaScript.* Proceedings of the "25th Conference of the ASU System Modelling Using Object Oriented Simulation and Analysis", Balaton, Hungary, 1999, p.61-71.

[14] Sklenar, J.: *Object Oriented Programming in JavaScript.* Proceedings of the "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.35-42.

[15] Sklenar, J.: *Simulator of Queueing Networks.* Proceedings of the "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.127-135.

[16] Sklenar, J.: *Interactive Simulators in JavaScript.* Proceedings of the "15th European Simulation Multiconference Modelling and Simulation ESM 2001", Praha, 2001, p.247-254.

[17] Sklenar, J.: *Client Side Web Simulation Engine.* Proceedings of the "27th Conference of the ASU Model Oriented Programming and Simulation", Rättvik, Sweden, 2001, p.1-13.

[18] Sklenar, J: *Teaching Nested Simulation.* ASU Newsletter, 2002, Vol. 27, No. 2, p.43-49.

[19] Sklenar, J.: *Discrete Event Simulation in JavaScript.* Proceedings of the "28th Conference of the ASU The Simulation Languages", Brno, Czech Republic, 2002, p.115-121.

Výsledky simulačních studií

[20] Sklenar, J.: *Simulation of Data Transmission Networks.* Proceedings of the conference "Remote Data Transmission'79", Karlovy Vary, 1979, p.168-170.

[21] Sklenar, J.: *Standard Elements of Simulation Models of Data Transmission Systems.* Proceedings of the conference "Remote Data Transmission'85", Karlovy Vary, 1985, p.109-111.

[22] Sklenarova, K., Sklenar, J.: *Simulation of Data Transmission Systems.* Proceedings of the conference "Remote Data Transmission'87", Karlovy Vary, 1987, p.72-74.

[23] Sklenar, J.: *Methodology of Simulation of Data Transmission Systems.* Proceedings of the conference "Simulation of Systems", Kopřivnice, 1988, p.71-75.

[24] Sklenar, J.: *Simulation Study of Two Versions of a Data Network.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.15-21.

[25] Sklenar, J.: *Formal Description of a Circuit Switching Network.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.23-29.

[26] Sklenar, J.: *Formal Description of a Packet Switching Network.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.31-36.

[27] Borg, A., Cordina, G., Sklenar, J.: *A Sectoral Model of the Patterns of Water Consumption in Malta.* Proceedings of the "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.137-147.

[28] Borg, A., Cordina, G., Sklenar, J.: *A Model of the Macroeconomic Dynamics of a Small, Open Economy: An Application to Malta.* Proceedings of the "26th Conference of the ASU Object Oriented Modeling and Simulation", Malta, 2000, p.149-156.

[29] Sklenar, J: *Predicting Water Demand in a Small Mediterranean Country.* International Journal of Computing Anticipatory Systems, 2002, Vol. 11, p.387-397.

APLIKACE PETRIHO SÍTÍ

[30] Dostal, O., Sklenar, J.: *Petri Net as a Tool for Verification of Communication Protocols.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.117-123.

[31] Sklenar, J.: *Graph Models of Dynamic Systems.* Proceedings of "The First Malta Conference on Graphs and Combinatorics", Malta, 1990, abstract of paper, p.27.

[32] Sklenar, J.: *Environment for Modelling of Petri Net Based Networks in the LOGOS Language.* Proceedings of the conference "EUROLOGO'91", Parma, 1991, p.385-400.

[33] Sklenar, J., Sklenarova, K.: *Network Paradigm Modelling in the LOGOS Language.* Proceedings of the conference "EUROLOGO'93", Athens, 1993, p.289-296.

[34] Sklenar, J.: *PetriSim - Environment for Simulation of Petri Networks.* Proceedings of the "20th Conference of the ASU Object Oriented Modelling and Simulation", Praha, 1994, p.214-221.

[35] Sklenar, J.: *Time Net Based Model of a Packet Switching Node.* Proceedings of the "AMSE Conference Communications, Signals and Systems CSS'96", Brno, 1996, p.469-472.

[36] Sklenar, J.: *Discrete Simulation and Time Networks.* Proceedings of the "22nd Conference of the ASU Object Oriented Modelling and Simulation", Clermont-Ferrand, 1996, p.57-67.

[37] Sklenar, J.: *Event Driven Visual Programming in PetriSim Environment.* Proceedings of the "23rd Conference of the ASU Object Oriented Modelling and Simulation", Stará Lesná, Slovakia, 1997, p.170-179.

[38] Sklenar, J.: *Using Inheritance to Implement High Level Petri Networks.* Proceedings of the "24th Conference of the ASU Object Oriented Modelling and Simulation", Salzau, Germany, 1998, p.173-182.

[39] Sklenar, J.: *Simulation of Queueing Networks in PetriSim.* Proceedings of the "16th European Simulation Multiconference Modelling and Simulation 2002", Darmstadt, 2002, p.403-407.

[40] Caruana, E., Ceska, M., Sklenar, J.: *Petri Nets and their Use in Modelling of Manufacturing Systems.* Proceedings of the "XXIVth International Autumn Colloquium ASIS 2002 - Advance Simulation of System", Ostrava, MARQ, 2002, p.17-28.

TELEKOMUNIKAČNÍ TECHNIKA

[41] Sklenar, J.: *Microcomputers and Data Transmission.* Proceedings of the conference "Computers and their Use in Electrotechnics", Technical University of Brno, 1979, p.33-36.

[42] Sklenar, J.: *Mathematical Models of Amplitude Modulations.* Elektrotechnický Časopis, 1981, Vol. 32, No. 4, p.290-299.

[43] Sklenar, J.: *Microcomputer Implementation of an Encoder-Decoder of the Fire Code.* Slaboproudý Obzor, 1981, Vol. 42, No. 7, p.319-323.

[44] Sklenar, J.: *Software Implementation of Encoders-Decoders of Error Recovery Codes.* Proceedings of the conference "Modern Programming'82", Pezinok, 1982, p.69-87.

[45] Filka, M., Kapoun, V., Nemec, K., Sklenar, J.: *Diagnostic Methods in Telecommunication Networks.* Slaboproudý Obzor, 1985, Vol. 46, No. 1, p.46-48.

[46] Ertinger, Z., Sklenar, J.: *Data Transmission in Channels with High Error Rate.* Slaboproudý Obzor, 1985, Vol. 46, No. 1, p.33-38.

[47] Herman, I., Sklenar, J.: *Program Implementation of an Encoder-Decoder of Error Correcting Codes.* Proceedings of the Technical University of Brno, 1985, Vol.B-64, p.75-78.

[48] Sklenarova, K., Sklenar, J.: *Simulation of Telecommunication Systems.* Proceedings of the Technical University of Brno, 1986, Vol. B-112, p.23-39.

[49] Sklenar, J.: *Application of Microcomputers for Implementation of Telephone Modems.* Proceedings of the Technical University of Brno, 1988, Vol. B-121, p.233-238.

[50] Dostal, O., Sklenar, J.: *Formal Specification of Communication Protocols.* Proceedings of the Technical University of Brno, 1988, Vol. B-121, p.211-216.

[51] Brabec, Z., Ertinger, Z., Herman, I., Sklenar. J.: *Software of Complex Telecommunication Systems.* Proceedings of the Technical University of Brno, 1988, Vol. B-122, p.107-110.

[52] Ertinger, Z., Lhotak, M., Sklenarova, K., Sklenar, J.: *Discrete Simulation in Telecommunications.* Proceedings of the Technical University of Brno, 1988, Vol. B-122, p.111-114.

[53] Sklenar, J.: *Microprocessors 80286 and 80386.* Proceedings of the conference "MICROSYSTEM'88", Bratislava, 1988, p.72-73.

[54] Sklenar, J.: *Formal Description Techniques for Telecommunication Systems.* Proceedings of the conference "COMPUTER NETWORKS'88", Bratislava, 1988, p.88-89.

[55] Herman, I., Sklenar, J.: *Introduction to the SDL Language.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.57-67.

[56] Brabec, Z., Herman, I., Sklenar, J.: *The SDL Language and its Application in Telecommunications.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.69-77.

[57] Dostal, O., Sklenar, J.: *Verification of Communication Protocols.* Proceedings of the Technical University of Brno, 1989, Vol. A-43, p.125-130.

[58] Popela, P., Sklenar, J.: *Signal Identification by Nonlinear Optimization.* Proceedings of the "8th IEEE International Conference on Electronics, Circuits and Systems ICECS 2001", Malta, 2001, Vol. III, p.1151-1154.

[59] Popela, P., Sklenar, J.: *Nonlinear programming applied to harmonic analysis.* Proceedings of the "7th International Conference on Soft Computing MENDEL 2001", Brno, 2001, p.240-245.

Odkazy

Češka, M. (1994) *Petriho sítě*, Akademické nakladatelství CERM Brno.

Dahl, O.J. (1966) *Discrete Event Simulation Languages*, Norsk Regnesentral.

Darnell, R. (1998) *HTML 4 Unleashed. Professional Reference Edition.* Sams.net Publishing.

Desrochers, A.A., Al-Jaar, R.Y. (1995) *Applications of Petri Nets in Manufacturing Systems*, IEEE PRESS.

Diaz, M. (1983) *Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models*, Computer Networks, No 6, p.419-441.

Eckel, B. (1998) *Thinking in Java.* Prentice Hall. Inc.

Flanagan, D. (1998) *JavaScript - The Definitive Guide.* O'Reilly & Associates, Inc.

Forrester, J.W. (1961) *Industrial Dynamics*, The MIT Press.

Fries, G. (2001) 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.

Haverkort, B.R. (1998) *Performance of Computer Communication Systems. A Model.Based Approach*, John Wiley & Sons.

Kelton, W.D., Sadowski, R.P., Sadowski, D.A. (2002) *Simulation with Arena*, McGraw-Hill.

Lindemann C. (1998) *Performance Modelling with Deterministic and Stochastic Petri Nets*, John Wiley & Sons.

Markowitz, H.M., Hausner, B., Karr, H.W. (1963) *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall.

McNeley, J.L. (1968) *COMPOUND DECLARATIONS*, Proceedings of the IFIP Working Conference on Simulation Languages, Oslo, North-Holland Publishing Company, p.292-304.

Murata, T. (1989) *Petri Nets: Properties, Analysis and Applications*, Proceedings of the IEEE, Vol. 77, No 4, April 1989, p.541-580.

Peterson, J.L. (1981) *Petri Net Theory and Modelling of Systems*, Prentice-Hall.

Pidd, J. (1998) *Computer Simulation in Management Science*, John Wiley & Sons.

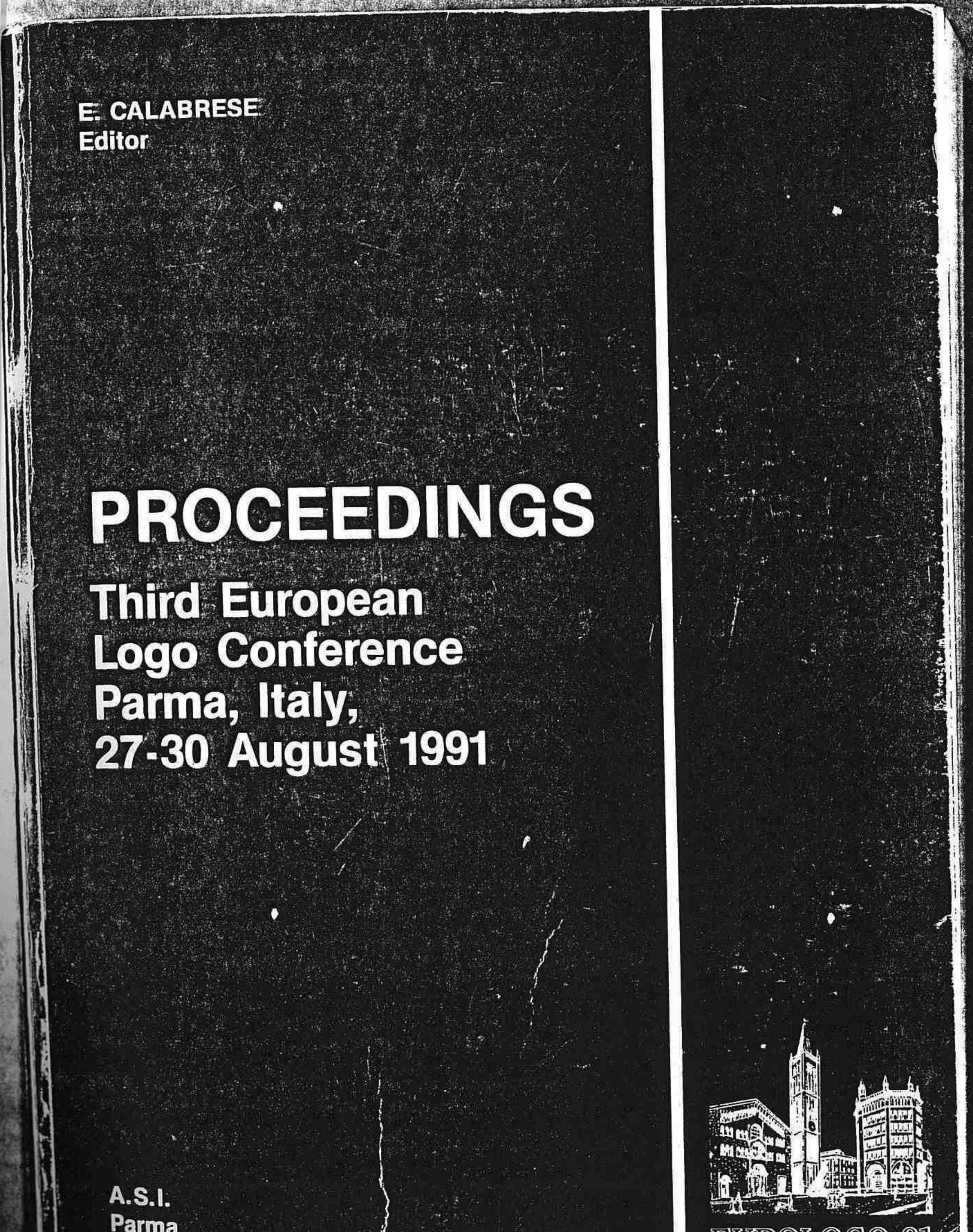
Reisig, W., Rozenberg, G. (Eds.) (1998) *Lectures on Petri Nets I: Basic Models*, Springer-Verlag.

Reisig, W., Rozenberg,. G. (Eds.) (1998) *Lectures on Petri Nets II: Applications*, Springer-Verlag.

Starke, P.H. (1980) *Petri Netze*, VEB Deutscher Verlag der Wissenschaften, Berlin.

Weinberger, J. (1991) *Logo, Simulation and Object-Oriented Programming*, Sborník konference EUROLOGO 91, 1991, Parma, p. 219-225.

Zeigler, B.P. (1976) *Theory of Modelling and Simulation*, John Wiley & Sons.



E. CALABRESE
Editor

PROCEEDINGS

Third European
Logo Conference
Parma, Italy,
27-30 August 1991

A.S.I.
Parma
1991

EUROLOGO 91

ENVIRONMENT FOR MODELLING OF PETRI NET BASED NETWORKS IN THE LOGOS LANGUAGE

Jaroslav Sklenar

University of Malta
Department of Computer Studies
Msida
MALTA

Abstract

The paper deals with the modelling of Petri net based networks. For the modelling the Logos language has been used. The Logos language is the Logo extension intended to support teaching of Object Oriented Simulation. Facilities similar to those of the system class Simulation of the Simula language are available. The modelling environment able to create, modify and execute any number of Petri nets has been designed. Using the concept of subclasses the basic system has been modified to enable modelling of both Petri and Coloured and both Petri and Time networks.

1. Introduction

In computer simulation one of the most important steps is the definition of the system simulated. For this purpose various informal and formal means are used. Use of some formal apparatus decreases the probability of errors and ambiguities in the system definition. The apparatus used depends on the nature of the system simulated.

For description of systems with parallel processes very often Petri networks (P-nets) and their modifications are used. Use of network models enables a quite natural expression of parallelism. The problem is the expression of dynamics because the original P-nets don't use time at all. That's why the so called time nets (T-nets) were developed to incorporate time to network models.

For practical use of network models the supporting software system is necessary.

2. Petri networks

The purpose of this section is an informal introduction of an idea of P-nets. The marked P-net is a 5-tuple (P,T,I,O,M) where P and T are non empty finite sets of places and transitions respectively. P-net is an oriented graph. The input function I defines for every transition the set of input places. The output function O defines for every transition the set of output places. For every input/output place the number of input/output arcs connecting this place with the transition is defined. Places can contain any number of the so called tokens. Initial marking M defines the numbers of tokens in all places before the start of the execution of the P-net. A transition can be enabled or disabled. A transition is enabled if all its input places contain at least so many tokens as is the number of oriented arcs from the place to the transition. Transition which is not enabled is disabled. An enabled transition can be fired. During firing every arc starting at an input place takes one token from this place and every arc starting at the transition adds one token to its output place.

From modelling point of view places represent some conditions which must be true before some activity (transition firing) can start. Firing changes the markings of both input and output places. This can enable other transitions to be fired. Transitions can be fired in parallel because there can be more enabled transitions at a time. Thus P-nets are very suitable for modelling large systems with parallel processes especially means of inter process communication and synchronization. P-nets have been intensively studied. They have also some analytical capabilities. For details see [1] where many other references can be found.

The disadvantage is that P-net as a formal apparatus for system description is very primitive. That's why there are many modifications of the original P-nets. See for example [2]. The Coloured nets generalise the idea of marking. The T-nets incorporate time to network models. There are several modifications of T-nets but typically they introduce the duration of firing of a transition.

3. Environment for network modelling

Requirements which should be satisfied by a software system able to create models based on various modifications of P-nets are:

a) Creation of networks with various numbers of places and transitions. Some properties of places and transitions will be common, some will be modified (individualized). It should be possible to modify both network marking and topology. Interactive network execution is required.

b) Both places and transitions have some attributes and can perform some actions (procedure attributes).

c) For T-nets the activity is expressed as a performance of parallel processes (process is a sequence of firings of a transition).

d) The system should be open for other types of networks because there is a continued development in this field. An introduction of a new network type should be based on an existing system (e.g. system for original P-nets) and only new special features should be added.

e) A means (language) used must meet all those requirements directly by its own constructs. Writing translators or language preprocessor is too time consuming and not necessary.

The Object Oriented Programming (OOP) copes with all the problems mentioned above in a very natural and elegant way. Its main principles are as follows:

1) System dynamics is understood as a common performance of objects (actors). Each object has its value and procedure attributes and has its life. In Object Oriented Simulation (OOS) the life has a dimension in time.

2) It is possible to define prototypes (classes) of similar objects and to generate any number of instances according to the pattern. A class represents a general knowledge, which is individualized by attributes of generated objects.
3) The inheritance principle enables a hierarchical classification of classes. A subclass inherits all attributes from its parent class. New attributes may be added, attributes inherited may be changed. Thus a subclass represents a more detailed knowledge based on the more general inherited knowledge. Each subclass may have its subclasses etc. Procedure attributes (actions) of the parent class can be considered as a problem oriented language within a subclass. Thus new constructs can be added without writing any language processors.
4) There are facilities for communication and synchronisation of lives of objects which have a time dimension in case of OOS.

The use of the OOP approach to solve problems mentioned above is evident. Knowledge of places and transitions can be represented by two classes with appropriate attributes storing marking and information about network topology. Actions can represent activities involved in transition firing. The most important is the use of inheritance. All network concepts have some common features. Those can be inherited. New things and ideas can be introduced by adding new attributes or by modifying existing ones.

3.1 The Logos language

The Logos language is an extension of the Logo language. Its purpose is especially a support of education of OOP and OOS. The Logos language enables a use of all Logo facilities and moreover it offers a set of facilities useful for creation of simulation models of discrete systems. From Logo point of view the new constructs are procedure calls (all names start by the "l" character). New simulation oriented statements are similar to those used in the Simula's system class Simulation. The Logo interpreter enables an interactive work which is very advantageous for education where the relatively limited speed doesn't cause problems. There will be another paper at this conference dealing with the Logos language in more detail.

4. Knowledge of P-nets

The knowledge of P-nets is represented by 3 Logos classes:

`$`	—	class representing a P-net,
`$P`	—	class representing a place of a P-net,
`$T`	—	class representing a transition of a P-net.

4.1 Knowledge of a P-net

The following example shows a conversation which declares the class $ with 14 attributes (without body).

```
?!CLASS "$
INSERT !ATTR OF $
NOP NOT AP AT EX FP FT I F S DP DT DPN DTN
INSERT !BODY OF $        (Enter pressed)
REPORT ON $ :
ATTRIBUTES OF $ :
[$.NOP $.NOT $.AP $.AT $.EX $.FP $.FT $.I $.F $.S $.DP
     $.DT $.DPN $.DTN]
BODY OF $ :
[& 1 !TERM]                (this is an empty body)
```

In the Logos language it is not necessary to distinguish between value and procedure attributes. It is possible to change a value attribute to a procedure one and vice versa. The class $ has the following value attributes:

`NOP`	—	number of places,
`NOT`	—	number of transitions,
`AP`	—	the actual place (place to work with at a time),
`AT`	—	the actual transition,
`EX`	—	this attribute distinguishes whether the object has been created (1) or not (0),
`FP`	—	number of the first place of the object (it is possible to create and execute several P-nets),
`FT`	—	number of the first transition of the object.

The class $ has the following procedure attributes:

```
I   — initialization (creation) of the object,
F   — fixing (modification) of the object,
S   — simulation (execution) of the object.
```

The following is the listing of the procedure `$.I` whose parameter is the object (P-net) number. Notes in curl brackets are not a part of the program:

```
TO $.I :N
(LOCAL "P "I "J "K "FP "FT)                              {Local variables}
MAKE "P (WORD "$ :N)                  {P contains the object name eg. $1}
IF THING (WORD :P ".EX)>0                  {Test whether $n.EX=0 or not}
     [PR ( LIST "PETRI "NET :P "EXISTS "YOU "CAN "CHANGE "IT
          "( WORD :P ".F ") )
     PR ( LIST "( "IT "HAS THING (WORD :P ".NOP) "PLACES "AND
          THING (WORD :P ".NOT) "TRANSITIONS ") )
     PR " STOP]                    {Information about an existing object}
TYPE ( LIST "ENTER "NUMBER "OF "PLACES "OF "THE "PETRI "NET
     :P ":)                                                     {Prompt}
MAKE WORD :P ".NOP RW                        {Reading attribute $n.NOP}
TYPE ( LIST "ENTER "NUMBER "OF "TRANSITIONS "OF "THE "PETRI
     "NET :P ": )
MAKE WORD :P ".NOT RW                        {Reading attribute $n.NOT}
MAKE "I THING (WORD :P ".NOP)                        {Number of places}
MAKE "J THING (WORD :P ".NOT)                   {Number of transitions}
MAKE "FP :$P.! + 1                   {Standard attribute $P.! contains}
               {the number of the last object (here place) generated}
MAKE WORD :P ".FP :FP                           {Store the value to $n.FP}
MAKE "FT :$T.! + 1                       {dtto for the first transition}
MAKE WORD :P ".FT :FT
REPEAT :I [MAKE "K !NEW "$P]                            {Create places}
              {The Logos statement !NEW creates a new object}
REPEAT :J [MAKE "K !NEW "$T]                       {Create transitions}
MAKE "K THING (WORD :P ".FT)                         {First transition}
```

```
REPEAT :J [ MAKE (WORD "$T :K ".EMP) 1 MAKE "K :K+1 ]
                    {Make all transitions empty — see attribute $Tn.EMP}
MAKE WORD :P ".EX 1                           {Make the object existing}
MAKE WORD :P ".AP :FP                      {Actual place is the 1st place}
MAKE WORD :P ".AT :FT                     {dtto for the actual transition}
PR ( LIST "PETRI "NET :P "HAS :I "PLACES "AND :J "TRANSITIONS
     )
PR(LIST "( "FIRST "PLACE :FP ", "FIRST "TRANSITION :FT "))
END
```

The procedures `$.F` and `$.S` are much more complicated, their description is beyond the scope of this paper. Appendix A contains the description of user commands interpreted by these procedures.

4.2 Knowledge of a P-net's place

The class `$P` has only one value attribute M which contains the marking of the place. Then it has 2 procedure attributes both with 2 parameters:

`REM n m` —	removing m tokens from the place n,
`ADD n m` —	adding m tokens to the place n.

These procedures enable individualization of P-net places. Every place has its own copy of the procedure attribute. So it is possible to change (edit) those procedures after P-net creation. Procedures can be used to collect statistic data, to display messages about current or special markings etc.

4.3 Knowledge of a P-net's transition

The class $T has the following value attributes:

`IP` —	list of input places,
`OP` —	list of output places,
`IPN` —	list of weights of input places (numbers of arcs from the place to this transition),
`OPN` —	list of weights of output places,
`EMP` —	this attribute distinguishes whether the transition is empty (1) or whether it has been already connected to the net (0).

The class $T has the following procedure attributes both with 2 parameters:

`TEST n t` — test whether the transition t of the net n is enabled,
`FIRE n t` — firing the transition t of the net n.

These procedures perform activities described in the section 2. They can be modified to introduce special features to P-net based models. One typical example of such a modification is an implementation of the so called predicate — see also [2]. Predicate represents a more complicated condition to enable transition firing. Within the procedure `$Tt.TEST` (name of the attribute TEST of the transition t) it is possible to evaluate any predicate (logical function of markings of input places of t).

The modification of the procedure `$Tt.FIRE` enables implementation of any special effects connected with the firing of the transition t.

4.4 Environment for P-nets

The environment for work with P-net based models uses the simulation facilities of the Logos language even if the models don't exist in time. The time is introduced in T-nets — see later. In Logos every simulation is in fact an execution of the body of the object `!MAIN1` of the standard class `!MAIN`. The computation starts by the command `!SIMULATION` followed by entering the program name and the attributes and the body of the class `!MAIN`. The following is the beginning of the conversation creating a program `PNET` in debugging mode. User attributes of `!MAIN` are not used, entering of the body is not shown because it is displayed as a part of the Logos report. The layout has been edited because of readability.

```
?!SIMULATION
INSERT THE NAME OF YOUR SIMULATION PROGRAM.
IT MUST START WITH A LETTER.
NOT MORE THAN FOUR CHARACTERS RECOMMENDED.
PNET
DEBUG?
TRUE
INSERT !ATTR OF !MAIN        (Enter pressed)
INSERT !BODY OF !MAIN
[!CLASS "$ !CLASS "$P --- etc. see report ---]
REPORT ON !MAIN :
ATTRIBUTES OF !MAIN : []
BODY OF !MAIN :
[& 1 !CLASS "$ !CLASS "$P !CLASS "$T]
[& 2 !RESAVEIMAGE]
[& 3 PRINT [* * * ENTER COMMAND ( C|F|F N|S|S N|Q )] MAKE "$1
    RL]
[& 4 IF ( FIRST :$1 ) = "C [!NEXT 9 ] ]
[& 5 IF ( FIRST :$1 ) = "F [!NEXT 11] ]
[& 6 IF ( FIRST :$1 ) = "S [!NEXT 13] ]
[& 7 IF ( FIRST :$1 ) = "Q [MAKE "!MAIN1.!LC 3 THROW
    "TOPLEVEL] ]
[& 8 PRINT [UNKNOWN COMMAND] PRINT " !NEXT 3 ]
[& 9 MAKE "$1 !NEW "$ MAKE (WORD :$1 ".EX) 0 PRINT ( LIST
    "PETRI "NET :$1 "CREATED ) ]
[& 10 RUN ( LIST ( WORD :$1 ".I ) :$.! ) !NEXT 3 ]
[& 11 MAKE "$1 BUTFIRST :$1 IF EMPTYP :$1 [MAKE "$1 :$.!]
    [MAKE "$1 FIRST :$1] ]
[& 12 RUN ( LIST ( WORD "$ :$1 ".F ) :$1 ) !NEXT 3 ]
[& 13 MAKE "$1 BUTFIRST :$1 IF EMPTYP :$1 [MAKE "$1 :$.!]
    [MAKE "$1 FIRST :$1] ]
[& 14 RUN ( LIST ( WORD "$ :$1 ".S ) :$1 ) !NEXT 3 ]
[& 15 !TERM]
```

Notes:

1) The body of the object !MAIN1 is a list of segments. The beginnings "`& n`" of segments are automatically added by Logos. The segment is a sequence of Logos statements. The statement `!NEXT` n is an unconditional jump to the segment number n.
2) The first segment contains declarations of classes mentioned above. The 2nd segment contains the statement `!RESAVEIMAGE` which saves the whole program (Logos plus all variables and procedures) on disk. It is performed as the 1st statement after loading. So it is possible to load the

program by the Logo statement LOAD, to edit it and after starting by the Logos statement !GO all changes will be saved on disk.

3) Segment 3 prompts the user and reads the command. Then the 1st letter is tested and the control is passed to the appropriate segment. The standard attribute !MAIN1.!LC contains the number of the segment to be executed as the next one. In the command "Q" it is assigned the value 3 and the control is passed to the Logo interpreter. So the next execution of the program starts by the prompt.
4) The command "C" (segment 9) creates a new object $n of the class $ and starts the procedure $n.I to initialize it.
5) The commands "F" and "S" (segments 11 and 13) test whether the network number has been entered. If not the number of the last network generated is taken. Then the procedure $n.F or $n.S respectively is called.
6) After execution of all commands but "Q" the control is passed back to the prompt. The last segment with the Logos statement !TERM finishing the execution of the !MAIN1's body is automatically added by Logos.

Appendix A contains a concise user guide of the Petri nets simulator called PETSIM.

5. Knowledge of Coloured nets

Coloured nets generalize the idea of marking. The finite non empty set of colours is defined. A place can contain tokens of different colours. So the marking of a place is a list whose elements are numbers of tokens of different colours. Similarly the weight of an arc from an input place to the transition is a list. Its elements are numbers of tokens of different colours required by this arc to enable the transition. These numbers of tokens are removed from the place during the transition firing. Similarly the output arcs contain the lists with numbers of tokens to be added to output places.

The knowledge of coloured nets is represented by subclasses of the classes described in section 4. The following subclasses have been defined:

$_C	—	subclass of the class $ representing a coloured net. It has one new attribute NOC which contains the number of colours,
$P_C	—	subclass of the class $P representing a place of a coloured net,

$T_C — subclass of the class $T representing a transition of a coloured net.

With the exception of NOC all subclasses have the same attributes as their parent classes. It is possible because Logo variables may contain both single values and lists. Procedure attributes have been modified according to the rules mentioned above.

The environment for work with coloured nets is very similar to this for P-nets. The first segment of the body of the object of the standard class !MAIN contains the declarations of both parent classes and subclasses. The Logos statement !SUBCLASS "$ "C declares the subclass $_C of the class $. It copies and renames all attributes of the parent class. For example the $.NOP is renamed to $_C.NOP etc. Then the user is prompted to change by editing the attributes. Because the parent classes are also present it is possible to create, modify and simulate any number of both Petri and Coloured nets at a time. The body of the object !MAIN1 has been slightly modified to enable working with both types of networks.

Thus the OOP approach of the Logos language has enabled the creation of a new environment with the minimum effort.

6. Knowledge of T-nets

Using the same method as for the Coloured nets the environment for work with the so called modified T-nets has been created. The T-nets introduce the time to network models. To test whether the transition is enabled or not the predicate is evaluated. Markings of places are generalized. The most important difference is the use of the body of the class $T_T called $T_T.!BODY. This is the class representing the knowledge of a transition of a T-net. The following is the listing of the $T_T.!BODY.

```
[& 1 IF ( RUN ( LIST WORD !CURR ".PRED ) ) = 1 [ !NEXT 2 ] [
     !NEXT 5 ]]
[& 2 RUN ( LIST WORD !CURR ".XP ) ]
[& 3 !HOLD RUN ( LIST WORD !CURR ".DUR ) ]
[& 4 RUN ( LIST WORD !CURR ".YP ) !NEXT 1 ]
[& 5 !QUE ( WORD "$_T :$1 ".QU ) ]
[& 6 !OUT !CURR ( WORD "$_T :$1 ".QU ) !NEXT 1 ]
[& 7 !TERM ]
```

Notes:

1) The 1st segment tests whether the transition is enabled. The procedure `$T_Tn.PRED` evaluates the predicate enabling the transition n. The Logos function `!CURR` returns the name of the so called current object (object whose body is executed). If the transition is enabled the `$T_Tn.PRED` returns "1".
2) The 2nd segment performs the procedure `$T_Tn.XP` which represents the activity connected with the beginning of the firing of the transition.
3) The 3rd segment represents the duration of the firing. The Logos procedure `!HOLD` suspends the execution of the object's body for the time specified. This time is computed by the procedure `$T_Tn.DUR`.
4) The 4th segment calls the procedure `$T_Tn.YP` which represents the activity connected with the end of the firing of the transition.
5) In the 5th segment the Logos procedure `!QUE` puts the current object to the specified queue. `QU` is the new value attribute of the class `$_Tn`. It is a queue of the disabled transitions. Every place of the T-net has a new attribute which is a list of transitions which have this place among their input places. After changing the marking of any place the transitions from this list waiting in the queue `QU` are resumed. Their bodies continue at the segment 6.
6) The Logos procedure `!OUT` in segment 6 removes the transition from `QU`. The control is passed to the segment 1 where the predicate is evaluated.

The body of `!MAIN1` has also been slightly modified especially the "`S`" command. It calls the `!HOLD` procedure to define the duration of the simulation run. After its completion the procedure `$_Tn.EVAL` which evaluates the simulation results is called. For T-nets it is supposed that almost all procedure attributes of objects will be modified according to the simulation requirements. The environment enables work with both Petri and T-nets. It is consistent with both environments described above because the same user commands are used. The user's work on a simulation model consists in the definition of the network topology and especially in programming (editing) the procedure attributes of all objects including `!MAIN1`. It is possible to create and work with several networks simultaneously.

7. Conclusion

The design of the three environments has proven the usefulness of the object oriented approach supported by the Logos language. The purpose of the software packages is to support education of the following topics:

- Object Oriented Simulation of discrete systems,
- network modelling using Petri net based formalisms,
- application of principles of OOP.

REFERENCES

1) Peterson, J.L.: Petri Net Theory and Modelling of Systems. Prentice-Hall Inc., Englewood Cliffs, N.J. (1981)

2) Diaz, M.: Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models. Computer Networks, no. 6, 419-441 (1982)

Appendix A: Petri net simulator user guide.

Starting system:

`LOGO`	Starting Logo interpreter from MS-DOS.
`LOAD "PETSIM`	Loading an empty system.
or	
`LOAD "Any_name`	Loading existing network(s) — see `SAVE`.
`!GO`	Starting P-net interpreter. Prompt: `* * * ENTER COMMAND (C|F|F N|S|S N|Q)`

PETSIM commands:

`C`	Creating a new P-net. After prompt enter number of places and transitions. PETSIM displays the numbers of the first place and the first transition.
`F [n]`	Fix (modify) the P-net n (last created by default). Prompt: `ENTER COMMAND ( H = HELP )`

Subcommands:

`H`	Displaying menu.
`T [n]`	Create the connection of the transition n (first or next by default). After prompt enter lists of: - input places - weights of input places - output places - weights of output places
`P [n]`	Set marking of the place n (first or next by default). After prompt enter the marking as an integer constant.
`M`	Set markings of all places. After prompt enter list of markings.

D Display P-net information about:

`D`	all places and transitions,
`D T`	all transitions,
`D T n`	transition number n,
`D P`	all places,
`D P n`	place number n.

Q Quit the command F (go to the upper level).
It is recommended to save the network(s) to disk using the Logo SAVE command.

`S [n]` Simulate the P-net n (last created by default).
Prompt:

```
SET MODE | DISPLAY | QUIT ( 0|1|2|3|D|Q )
```

Options:

0 Stop before every transition firing.
Prompt:

```
PRESS 'S' TO SET MODE OR ANY KEY TO CONTINUE.
```

1 Stop if there are 2 or more enabled transitions.
(Select transition to be fired).

2 Random selection of a transition if there are more than one enabled. Displaying messages about enabled and fired transitions.

3 Random selection of a transition, no display on the screen.
(Use this option to search for deadlocks).

Notes:

1) If all transitions are disabled the system displays the message: `DEADLOCK - NO TRANSITION ENABLED` then it displays the markings of places.
2) For options 2 and 3 it is possible to stop the run by pressing any key but 'S'. By pressing 'S' the run is stopped and it is possible to change the option. Otherwise the user is prompted to press any key to continue.

`D`	Displaying markings of places.
`Q`	Quit the command S (go to the upper level).

`Q` Quit the PETSIM (go back to the Logo interpreter).

Finishing work:

`SAVE "Any_name`	Saving the whole system with all created networks in the current state to disk.
`.DOS`	Go back to MS-DOS.

ASU

The Association of SIMULA Users

Object-Oriented Modelling and Simulation

Proceedings
of the
22nd Conference of the ASU

15th-17th July 1996

University Blaise Pascal
Clermont-Ferrand
France

Discrete Simulation and Time Networks

Jaroslav Sklenar
Department of Computer Information Systems
University of Malta, Msida, Malta
e-mail : jskl@unimt.mt

Abstract

The common problem of discrete simulation is complexity of most of the activities involved. That's why there is a permanent effort to find a way how to simplify the process or at least to give a clear and commonly acceptable guide. The so called Paradigm Oriented Simulation is based on certain apparatus (abstraction, paradigm) which is used as a tool for the description of the system simulated, as a base for creation of the simulation model and as a support of other tasks like model verification, experimentation, documentation, etc. A commonly accepted tool for simulation of discrete systems are Petri networks and their modifications. To support Petri net based simulation the package PetriSim written in the Turbo Pascal language has been created. It contains a graphical editor of Petri networks and it also supports development of user defined models based on Petri networks. Version 2 of PetriSim supports simulation based on the so called Time networks. There are several modifications of Time networks, but all of them are based on the original Petri networks and all of them introduce time to network models. PetriSim is based on the fact, that firing of a transition can take a certain time. This time (firing duration) is generated by the user at the moment when the firing starts. So there may be more transitions in the "firing on" status with various times of the firing completion. The package thus contains a usual (*SQS*) structure which orders these future events by their occurrence in time. Time networks based simulation starts by creating a Time network whose purpose is timing and synchronisation of all activities in the model. Then the user writes a Turbo Pascal unit whose procedures contain all user defined details like model initialisation, observation and statistics, experiment evaluation, and debugging operations. All these actions are supported by objects, procedures and functions, that simplify programming of all typical activities. The paper explains the basic ideas and shows how a simulation model of a simple queuing network has been created. The PetriSim environment makes model creation relatively simple and straightforward even for a not very experienced Pascal programmer.

1. Introduction

Computer simulation of discrete systems belongs to most difficult programming tasks. The reason of it is that we must add one more dimension - time, because in the model it is necessary to keep the same sequence of activities as in the real system. That's why programming simulation models in general 3GL is considered too difficult. Of course there are efficient simulation languages like Simscript, Modsim, class Simulation of Simula and others that offer tools for all typical simulation tasks. The problem is learning these languages. For people who do not intend to write simulation models regularly and for students who are supposed to learn basic ideas of simulation during one semester course this is an unacceptable initial investment. That's why there is a permanent effort to simplify the simulation process. There exist for example many special purpose simulators (like Comnet, Network, Simfactory, and basically all simulators of continuous systems), in which only input data are requested.

Another approach is the so called *Paradigm Oriented Simulation*. It is based on certain paradigm that is used as a tool for the description of the model and other activities involved in a simulation study. The tool used depends on the nature of the system simulated. A commonly accepted tool for description of discrete systems are Petri networks and their modifications. Petri networks (P-nets) enable a very natural and lucid way how to express cooperation and communication of parallel processes. This is very useful when modelling computer and communication systems but there exist also many other application areas. Detailed description of P-nets and their modifications is out of the scope of this paper. All important definitions and results can be found in [1], [2] deals especially with modifications intended to increase the modelling power of network models. Because of the paper consistency the next chapter explains basic ideas of Petri and Time networks.

2. Basic Ideas of Petri Nets

The marked Petri Net is a 5-tuple (P, T, I, O, M) where:
P and T are non empty finite sets of places and transitions respectively ($P \cap T = \emptyset$).
I is the so called Input function: I: $P \times T \rightarrow \mathbf{N}$, where $\mathbf{N}$ is the set of non negative integer numbers. The value $I(p,t)$ is the number of (directed) arcs from the place p to the transition t.
O is the so called Output function: O: $T \times P \rightarrow \mathbf{N}$, where the value $O(t,p)$ is the number of arcs from the transition t to the place p. So the 4-tuple (P,T,I,O) is a bipartite (bichromatic) directed multigraph whose arcs connect nodes of 2 distinct sets (P and T). When expressed graphically places are drawn as circles and transitions are drawn as short thick lines - see the Figure 2.
M is the initial marking of places: M: $P \rightarrow \mathbf{N}$, where the value $M(p)$ is the number of the so called tokens that are located in the place p.
A transition is enabled if all places contain at least so many tokens as is the number of arcs from the place to the transition. A transition which is not enabled is disabled.
An enabled transition may be fired. During firing every arc whose endpoint is the transition removes one token from its starting (input) place and every arc starting at the transition adds one token to its ending (output) place.

From the modelling point of view places represent certain conditions that must be true for certain activity (transition firing) to start. Firing changes the marking of both input and output places. This may enable or disable other transitions, etc. All enabled transitions may be fired in parallel. Petri networks have both analytical and descriptive (modelling) capabilities. For more details see [1]. Because the descriptive capabilities of the original Petri model are rather poor, there are many extensions to the basic P-nets to improve modelling power of network models. A good overview can be found in [2]. The so called Time networks (T-nets) introduce time, because in the original P-nets time does not exist at all. There are several approaches how to cope with time, PetriSim is based on the most common "firing delay" approach. Firing takes certain time (that of course may be zero). When a firing starts, the tokens are removed from input places (this represents the fact, that certain activity has started). When the firing ends, the tokens are added to output places (certain activity has terminated). Analytical properties of this abstraction are irrelevant, because PetriSim uses T-nets as a descriptive tool only.

3. PetriSim Facilities

To explain the basic ideas of *T-net based Paradigm Simulation* this chapter describes how a particular simulation model has been created using PetriSim facilities. The emphasis is the user's point of view. Implementation aspects has been briefly dealt with in [3]. PetriSim implementation is based on object oriented capabilities of the Turbo Pascal 7 language. All screen objects are object instances, most PetriSim functions are represented by object methods. This approach makes any PetriSim amendments and modifications relatively easy, that has been proven during implementation of the version 2. Objects involved in PetriSim implementation are not directly accessible to the user. There are some units in PetriSim environment, that are intended as a support of typical simulation tasks (random numbers, statistics, linked lists). Here the user is supposed to use the objects of these units either directly or to declare other specialised objects with inherited capabilities. The following main steps are involved in building a simulation model using the *T-net based Paradigm Simulation*:

- Informal description of the system simulated,
- Creating a T-net model,
- Creating a user model on the T-net skeleton,
- Experimentation with the user model.

PetriSim supports the last three steps.

3.1 Informal Description of the System Simulated

The system simulated (see Figure 1) is a queuing network made of three queues and three servers. All customers are at first served by Server1. If the Server1 is busy, they wait in the queue Queue1. Then the customer proceeds either to the server Server2 (with the probability P_1) or to the server Server3 (with the probability $1-P_1$). Customers waiting for these servers form the queues Queue2 and Queue3 respectively. Population of customers and sizes of queues are not limited, queues are orderly FIFO queues. Interval between two adjacent arrivals and duration of the three services are random variables with experimentally obtained distribution (distribution function tables are available). For this system the usual queuing statistics (times spent at the servers, queue lengths, and total time spent in system) is to be found together with utilisation of servers. This queuing system can not be analysed by analytical models of queuing theory because of general distribution of arrival intervals and service duration. (Queuing theory results for queuing networks are very limited anyway and even those that exist are very complicated and time consuming to use). So the only feasible way how to get the results is creating a simulation model and experimenting with it to gather all data necessary to compute the results.

To create a network model an informal description (the first step of all simulation studies) should be made in such a way to support identification of parallel processes and their interactions. Instead of parallel processes it is also possible to state clearly necessary conditions and results of all activities.

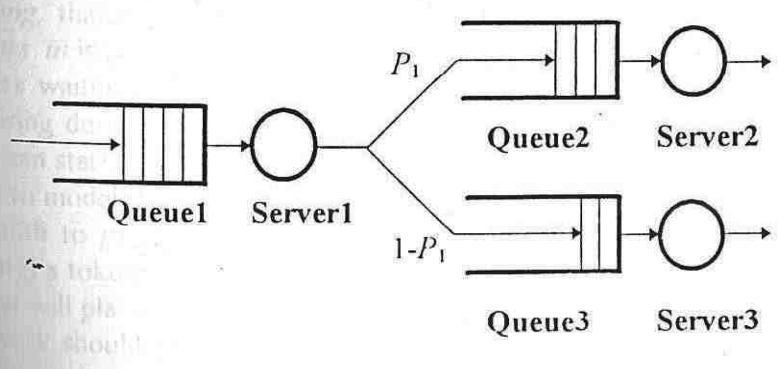


Figure 1: The queuing system being simulated

3.2 Creating the T-net model

A T-net model is intended to represent the basic relationships in the model, especially all synchronisation and communication mechanisms, and unlike P-net model also the model timing. There is no commonly accepted guide how to convert real world processes to the 'T-net language'. Nevertheless certain network constructs are obvious and may be learned from other network models. T-net creation starts by expressing the conditions in terms of presence of tokens in certain places. Then the activities must be expressed as transitions together with arcs from input places and arcs to output places. One possible T-net abstraction of the queuing system in the Figure 1 is the T-net in the Figure 2. Note that graphically a T-net looks like a P-net. The difference is visible only during a user simulation experiment when the network can be displayed in current status (the transitions in firing state are displayed in different way together with the firing completion time).

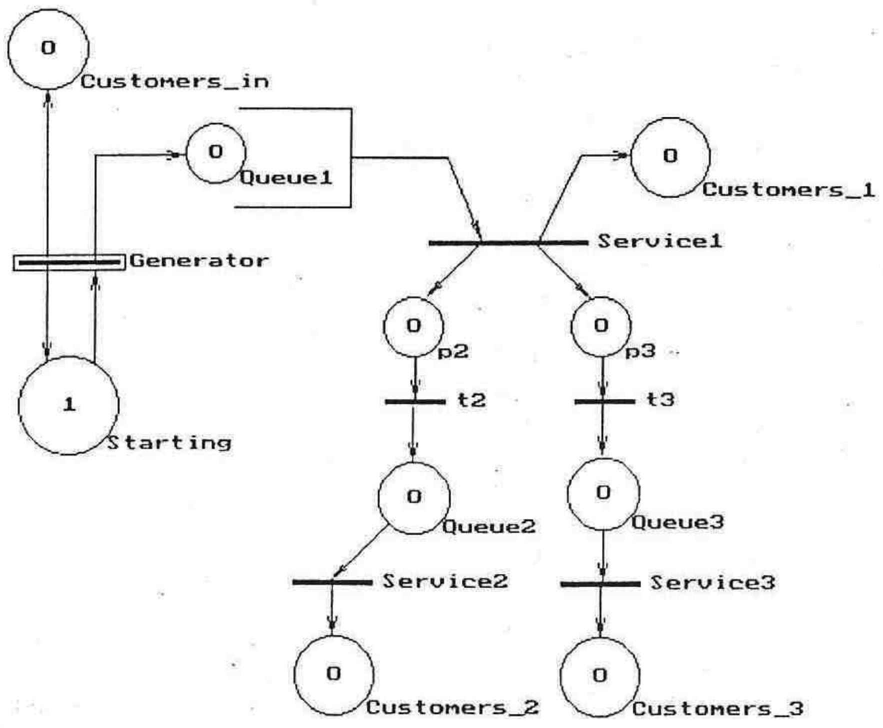


Figure 2 : T-net abstraction of the queuing system

Arrival of customers is represented by the transition *Generator* whose firing duration is the interval between two adjacent arrivals. The token in the place *Starting* (the only input place of *Generator*) starts the whole process. When the firing of *Generator* ends the token is returned to *Starting*, that will start immediately next firing (interval till the next arrival). The place *Customers in* is the counter of customers that entered the system, the place *Queue1* represents customers waiting for Server1. Working of Server1 is modelled by the transition *Service1* whose firing duration is the random service time. The transition has one input place *Queue1* (service can start if there is at least one customer - token - in the queue). The places p_2 and p_3 are used to model the random switching of customers. When firing of *Service1* ends, a token is placed both to p_2 and p_3 and a user procedure is activated. This procedure then removes (randomly) a token from either the place p_2 or p_3. So only one of the transitions t_2 or t_3 will be fired, that will place a token to one of the places *Queue2* or *Queue3*. This user interaction with the network should of course be used only if necessary with great caution, but it works well and T-net is kept relatively simple. There is a purely T-net way how to model random switching, but it needs several more places and transitions. Obviously the transitions *Service2* and *Service3* represent the other two servers, the places *Customers_1*, *Customers_2*, and *Customers_3* are counters of customers served by the three servers.

In PetriSim environment a T-net can is built in incremental way by repeated additions of various objects to an initially empty net. It is always possible to add any number of new places and new transitions and to create new arcs between them. Arcs can be deleted and they may be given practically any shape including free drawing - see the arc from *Queue1* to *Service1*. The T-net can always be stored in a disk. Because the file name can be defined by the user it is possible to keep more versions of the same network model. PetriSim contains some demonstration examples that explain basic ideas and typical network constructs.

PetriSim supports several simulation modes that do not work with time (so actually a P-net model is simulated). These modes differ in the degree of user control over firing of transitions. If there are more enabled transitions the user can either select the transition to be fired or the selection can be done randomly by PetriSim (equal probabilities). Some modes are based on user action before firing a transition. These step modes enable detailed studying and analysis of network behaviour. There are also two automatic modes with and without delay between firings. The fast mode is supposed to be used when searching for deadlocks. P-net simulation checks basic logical relationships in the model, especially all kinds of communication and cooperation procedures and mechanisms. If there are any user interactions with the model, they must be in these P-net modes performed manually, so practically only step modes can be used. Also all cases when the network behaviour depends on firing duration must be simulated by careful user's selection of the transition to be fired.

3.3 Creating a user model on the T-net skeleton

After experimentation with the P-net a user model is created to obtain quantitative results and to enable simulation in time. T-net models are able to do the model timing, but to get quantitative results they are as primitive as the original P-nets. The only quantitative data available are various counters represented by numbers of tokens in certain places. In our model this is used to collect data on lengths of queues and to count customers. The basic problem is the fact, that tokens do not contain any data. In our model we have to measure the time spent at servers and in the system. When working with tokens only, we know times when services end, but we do not know when services (or waiting) have started. This is a fundamental problem that may be solved in two ways:

a) There are networks (P-net extensions) that *generalise the notion of tokens* so that tokens may be practically any data structures. This does not solve another problem: keeping tokens in structures. For example in our model we want the tokens to wait for service in FIFO queues. The original P-net model of course does not define any structure on tokens in a place (they form a set). So it is obvious, that any individualisation of tokens must be based on many additional assumptions that of course acts against the simplicity and clear, well defined working of network models.

b) Another approach is *separation of network and user data*. This approach is used in PetriSim. Network is kept as primitive as possible (the only difference between T-nets and P-nets is the firing duration). All user data is kept (and observed) in user data structures. All operations on user data structures are of course activated and synchronised by T-net generated events (starting and finishing firing of certain transitions). So for example in our model the place *Queue1* has an associated queue (a linked list similar to SimSet's Head) that contains objects (descendants of a linked list item similar to SimSet's Link) that represent customers. Having this it is possible to keep any user data on customers (in our model ~~we~~ just record the arrival time and the time when the first service has finished). Inserting customers to the queue is synchronised by finishing firing of *Generator,* removing is synchronised by starting firing of *Service1* (during the service the object is referenced by a pointer associated with *Service1*). There are another two queues associated with the places *Queue2* and *Queue3* respectively. Other user data is used for statistics and model parameters. Separation of user and network data keeps the network model clear and simple and the user is not limited in complexity of user data structures. The disadvantage of this approach is certain duplicity (customers are represented both by tokens in places and by objects in queues). This duplicity of course does not cause memory problems (all tokens in a place are represented just by the number of tokens - one long integer in four bytes).

The user part of the model is made of procedures and data written in the Pascal unit *USER*. For each version of *USER* the program *PETRISIM.PAS* is recompiled, so the program *PETRISIM.EXE* is in fact one particular user model (that of course may be used to create and work with any number of other nets, but can not perform their user experiments). In the implementation part of the unit *USER* it is possible to declare any number of procedures, functions, variables and any other objects. These are the user defined interface procedures of the unit *USER*.

USERINIT = User model initialisation called once before starting the experiment. This procedure is supposed to set up the initial model status that involves typically these:
- initialisation of user data like storing zeros to counters, etc.,
- creating and preparing all random, statistical, and SimSet like objects,
- reading model parameters (from keyboard or file),
- preparing animation.

*USERFIRE(**Var** s : String)* = The user's activity when starting firing a transition. The parameter *s* is the name of the transition to be fired. This and the next procedure represent the user's part of model behaviour that includes collecting statistical data, animating the model, etc.

*USERENDFIRE(**Var** s : String)* = The user's activity when finishing firing a transition. The parameter *s* is the name of the transition. This and the previous procedure represent the user's part of model behaviour.

USEREVAL = Evaluation of the experiment called once when the experiment terminates. This procedure is used to compute the results and to display them and/or store in file. Then the control is passed back to PetriSim.

USERDEAD = The user's activity in case of deadlock. *Deadlock* is a T-net status with no enabled transition. Normally a deadlock is caused by an error in synchronisation or communication algorithm. This procedure may display a user's message and perform any post-mortem analysis. Then the control is passed back to PetriSim.

To support writing procedures of the unit *USER*, there are certain facilities in PetriSim units that enable the user to generate firing delay, to get the model status and time, to change marking of places, to display the T-net in current status, that is very important for debugging, and to terminate the experiment:

FIREDELAY(dt : Real) is a procedure that generates a firing delay *dt*. This procedure is called by the procedure *USERFIRE* when the firing starts. A notice is created and inserted to *SQS*, so the firing will end after *dt* time units.

MARKINGOF(PName : String) : LongInt is the function that returns marking of the place *PName*. If the place is not found, the function returns -1. This function is used typically to collect statistical data.

PNTIME : Real is the function that returns the model time, that can not be directly changed.

*CHANGEMARKING(PName: String; M: LongInt; **Var** OK: Boolean)* is a procedure that changes marking of the place *PName* to *M*. *OK* contains false if the place has not been found.

SHOWNET is a procedure that displays the T-net in current status. Transitions in "firing on" status are displayed in different way together with firing completion time. This enables a detailed analysis of model behaviour.

TERMINATE is the procedure that terminates the user simulation experiment immediately after return from the procedure *USERFIRE* or *USERENDFIRE* to PetriSim.

3.4 Experimentation with the user model

User experiment is started from PetriSim for the selected T-net (there may be many nets in RAM at a time, but only one of them has its simulation experiment). The user selects the so called *User experiment* mode and then activates the menu option *Start simulation*. PetriSim at first calls the procedure *USERINIT* and then at proper times the procedures *USERFIRE* and *USERENDFIRE*. If there are more enabled transitions at a time, the one to be fired is selected randomly (equal probabilities). Anyway all enabled transitions start firing before the model time is updated to the next event time. In case of deadlock the procedure *USERDEAD* is called. If there is no deadlock, the experiment can be stopped only by the user by calling the procedure *TERMINATE* either in *USERFIRE* or *USERENDFIRE*. In this case the *USEREVAL* procedure is called and the control is passed to PetriSim. Then the experiment can be started again. Breakpoints are not implemented, because together with tokens it would be necessary to store time, the whole *SQS*, and all user data. Because PetriSim is intended as an educational tool, very long experiments are unlikely. After finishing the experiment the user sees the T-net in final status. There are menu options *Record marking* and *Initiate*, that enable storing and restoring network status. When loading a T-net from disk, the initial status is recorded automatically.

4. Statistical objects

The unit *STATIST* declares objects *RAccum* and *RTally* whose values are automatically statistically observed. They in fact represent observed real variables. Because Pascal does not have anything like Simscript's left monitoring, it is necessary to update a value by sending a message to the object instance (calling the method *RUpdate* with new value as a parameter). The method does all necessary updating and observation. Upon creation the object is initialised that in case of *RAccum* involves storing the initial value. Then there are methods - functions that return the current value and usual statistical figures (average, minimum value, maximum value, variance, standard deviation, and for *RTally* objects number of updates).

The difference between *RAccum* and *RTally* is the fact, that unlike *RTally RAccum* takes into account time. For example when storing a new value to an *RTally* object three times, the average is computed by dividing the sum of the three values by 3. In our model *RTally* objects are used to observe times spent at servers and the total time spent in the system. Each value is simply computed as a difference between current time and the recorded beginning of the measured period. So for example the statistics of the total time spent in the system is collected by this statement:

STime.RUpdate(PNTime - S2Cust^.Arrival);

where *STime* is the *RTally* object instance, *RUpdate* is the updating method, *PNTime* is the current time, *S2Cust* is the reference to the customer object whose service ends (here at the server Server2), and *Arrival* is the attribute that contains the time when the customer entered the system.

RAccum objects compute the time integral, so for example the average is computed by dividing the value of the time integral by duration of the time interval (initialisation automatically records the starting point - usually zero, the end point is the current time). In our model *RAccum* objects are used to observe lengths of queues and utilisation of servers (each server has an associated real status variable with two possible values: 0=idle, 1=working). So the average of this variable multiplied by 100 is the utilisation in %. So for example when the length of the Queue2 is changed, the change is recorded by this statement:

Q2.RUpdate(MarkingOf('Queue2'));

where *Q2* is the *RAccum* object instance, and *RUpdate* is the updating method that is passed the number of tokens in the place *Queue2* (number of waiting customers). User does not write any other code to get statistics. In the procedure USEREVAL the statistical results are just used in statements like this one:

WriteLn('Length of Queue2: Maximum: ', Q2.RMax:7:0, ' Average :', Q2.RAverage:7:2);

5. Random objects

The unit *RANDOMS* declares the objects *RandRS*, *RandRL*, and *RandI* that represent random numbers of three types: real step (discrete values), real linear (linear interpolation between two points of the distribution function), and integer. Each random object has its own stream of random numbers, so the initialisation method is given two parameters: seed of the random generator and the distribution function as an array of records of the fixed size. There is just one more method that returns the new random value. So for example when starting firing of the transition *Service1*, the end of firing is scheduled by generating a random firing delay:

FireDelay(Service1.RLValue);

where *Service1* is the random object instance that generates duration for the first server and *RLValue* is the method that always returns a new random value (here service duration) computed by linear interpolation. To generate random values the unit *RANDOMS* uses the standard Turbo Pascal 7 random generator, but each random object has its own current seed, that is used to update the generator's seed if new value is to be generated.

6. Linked lists

In [4] there is a description of the unit *SIMSET* that declares objects *Linkage*, *Head*, and *Link* intended to be as similar with the corresponding Simula objects as possible. There are the same methods as in the Simula system class SimSet and few more because of Pascal limited capabilities as an object oriented language. Nevertheless for the user the objects of the Pascal unit *SIMSET* are used practically in the same way as the Simula objects. So for example when firing of the transition *Generator* ends, that means an actual arrival of a customer, a customer is created, the arrival time is recorded, and the customer is inserted to the first queue:

New(Cust,Init); Cust^.Arrival := PNTime; Cust^.Into(Que1);

where *Cust* is the customer object instance (descendant of *Link*), *Init* is its initialisation method, *Arrival* is the attribute to store arrival time, *Into* is the SimSet like method to insert *Cust* at the end of *Que1*, that is the first queue (of the type *Head*).

Conclusion

Several discrete models have been created to check the usability of the package. The user Pascal programs are all relatively very simple, no programming is necessary for timing and synchronisation, for collecting and computing statistical data, for generating random numbers, and for operations on linked lists. Most of the code are short snippets that activate methods of objects. Appendix 1 is the complete text of both procedures that represent the user's part of model behaviour. Other procedures are relatively simple and what is more important, they can be in all PetriSim user models very similar if not the same. The total length of the unit *USER* is about 350 lines. PetriSim can be used in several ways both in research and education:

- as a graphical editor of Petri networks in all subjects and research projects related to this topic (P-nets may be printed, captured, and processed by other graphical editors and/or inserted to word documents - like the Figure 2 of this paper),

- as a user-friendly editor and simulator of Petri networks. P-nets may be created, stored in disk, and simulated in various modes. So it is possible to study and analyse various communication and synchronisation protocols and algorithms, e.g. in subjects and research projects oriented to computer communication,

- as a general and user-friendly simulator of discrete systems. The only skills required are basic ideas of T-net principles and (not advanced) Turbo Pascal programming. So PetriSim combines the advantages of special purpose simulators (that are easy to use, but not general) and discrete simulation languages (that are general, but learning them takes a lot of time and energy).

Currently the most important disadvantage of PetriSim (and of the P-net based modelling as such) is the problem of modularity. Complex networks too big to be displayed on one screen can be simulated, but parts of them must be kept invisible (interconnection of places and transitions can still be displayed in tabular form). Better solution is splitting a complex network to more simple ones connected by special places and/or transitions. Object oriented nature of PetriSim supports this approach. Even now it is possible to work with many nets at a time (each net is in fact an object instance). This is the way to develop future versions of PetriSim.

Appendix 1

```
Procedure UserFire(Var s : String);       { Called when firing of a transition Starts }
Begin                                     { s is the transition name }
   If PNTime>MaxTime Then Terminate;  { Experiment termination }
   If s='Generator' Then  { Generating interval between arrivals }
      FireDelay(Interval.RLValue)

   Else If s='Service1' Then Begin
      S1.RUpdate(1);  { Updating Server1 status (1=working) }
      Q1.RUpdate(MarkingOf('Queue1')); { Updating Queue1 length }
      S1Cust := CustP(Que1^.First);  { Starting service of the 1st customer in Queue1 }
      S1Cust^.Out;  { Removing the first customer from Queue1 }
      FireDelay(Service1.RLValue);{ Generating the service duration }
   End{If}

   Else If s='Service2' Then Begin  { Similarly for the other two servers }
      S2.RUpdate(1);  Q2.RUpdate(MarkingOf('Queue2'));
      S2Cust := CustP(Que2^.First);  S2Cust^.Out;
      FireDelay(Service2.RLValue);
   End{If}

   Else If s='Service3' Then Begin
      S3.RUpdate(1);  Q3.RUpdate(MarkingOf('Queue3'));
      S3Cust := CustP(Que3^.First);  S3Cust^.Out;
      FireDelay(Service3.RLValue);
   End{If};

   If Confir Then  { Confir = true in the step mode }
      Step('Starting Firing of ' + s)  { Procedure Step supports the step mode }
   Else Begin  { Updating screen in fast mode }
      GoToXY(5,6);  Write('Current Time = ', PNTime:10:3);
      GoToXY(5,9);  Write('Number of customers served: ‘,
                           MarkingOf('Customers_2')+MarkingOf('Customers_3'));
   End{If};
End{UserFire};

Procedure UserEndFire(Var s : String);       { Called when firing of a transition Ends }
Var Ok : Boolean;                            { s is the transition name }
Begin
   If s='Generator' Then Begin  { This is the actual arrival }
      Q1.RUpdate(MarkingOf('Queue1'));  { Updating Queue1 length }
      New(Cust,Init);  { Creating a customer }
      Cust^.Arrival := PNTime;  { Recording its arrival time }
      Cust^.Into(Que1);  { Inserting the new customer to Queue1 }
   End{If}

   Else If s='Service1' Then Begin  { First service ends }
      S1.RUpdate(0);  { Updating Server1 status (0=idle) }
      S1Time.RUpdate(PNTime - S1Cust^.Arrival);  { Collecting time spent at Server1 }
      S1Cust^.FirstEnd := PNTime;  { Recording first end of service }
      { Random switching: }
      If Random<P1 Then Begin  { Moving to Queue2 }
         S12Cust := S1Cust;  ChangeMarking('p3',0, OK);  { Disabling t3 }
      End
```

```
    Else Begin  { Moving to Queue3 }
       S13Cust := S1Cust;  ChangeMarking('p2',0, OK);  { Disabling t2 }
    End{If};
    If Not OK  Then  Begin  { This should not happen! }
       GoToXY(20,22);  Write('Wrong name - press Enter');  ReadLn;
       Terminate
    End{If}
 End{If}

 Else  If  s='t2'  Then  Begin  { Entering Queue2 }
    Q2.RUpdate(MarkingOf('Queue2'));  { Updating length of Queue2 }
    S12Cust^.Into(Que2)  { Inserting the customer to Queue2 }
 End{If}

 Else  If  s='t3'  Then  Begin  { Entering Queue3 }
    Q3.RUpdate(MarkingOf('Queue3'));  { Updating length of Queue3 }
    S13Cust^.Into(Que3)  { Inserting the customer to Queue3 }
 End{If}

 Else  If  s='Service2'  Then  Begin  { Service2 ends - customer leaves }
    S2.RUpdate(0);  { Updating Server2 status (0=idle) }
    S2Time.RUpdate(PNTime - S2Cust^.FirstEnd);  { Collecting time spent at Server2 }
    STime.RUpdate(PNTime - S2Cust^.Arrival);  { Collecting total time spent }
    Dispose(S2Cust);  { Removing the customer from RAM }
 End{If}

 Else  If  s='Service3'  Then  Begin  { Service3 ends - customer leaves }
    S3.RUpdate(0);  { Updating Server3 status (0=idle) }
    S3Time.RUpdate(PNTime - S3Cust^.FirstEnd);  { Collecting time spent at Server3 }
    STime.RUpdate(PNTime - S3Cust^.Arrival);  { Collecting total time spent }
    Dispose(S3Cust);  { Removing the customer from RAM }
 End{If}

 If  Confir  Then  { Confir = true in the step mode }
    Step('Finishing Firing of ' + s)  { Procedure Step supports the step mode }
 End
End{UserEndFire};
```

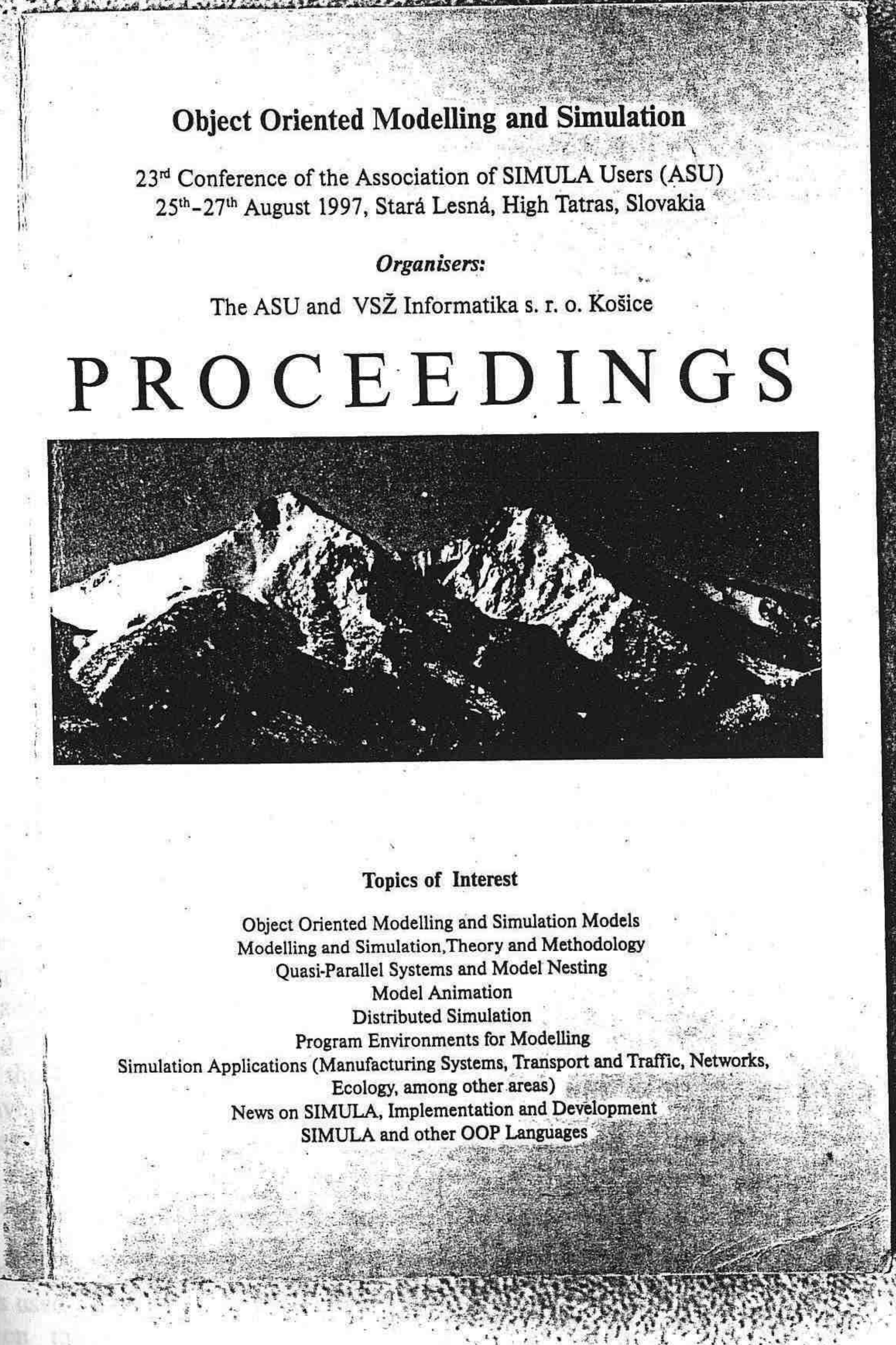
References

[1] Peterson, J.L.: *Petri Net Theory and Modelling of Systems.* Prentice Hall Inc., 1981, Englewood Cliffs, N.J.

[2] Diaz, M.: *Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models.* Computer Networks, 1982, no. 6, p. 419-441.

[3] Sklenar, J.: *PetriSim - Environment for Simulation of Petri Nets.* Proceedings of the 20th Conference of the ASU, 1994, Prague, p. 214-221.

[4] Sklenar, J.: *A Turbo Pascal unit SimSet.* ASU Newsletter, 1995, Vol. 23, No. 1, p. 1- 17.



Object Oriented Modelling and Simulation

23rd Conference of the Association of SIMULA Users (ASU)
25th-27th August 1997, Stará Lesná, High Tatras, Slovakia

Organisers:

The ASU and VSŽ Informatika s. r. o. Košice

PROCEEDINGS

Topics of Interest

Object Oriented Modelling and Simulation Models
Modelling and Simulation,Theory and Methodology
Quasi-Parallel Systems and Model Nesting
Model Animation
Distributed Simulation
Program Environments for Modelling
Simulation Applications (Manufacturing Systems, Transport and Traffic, Networks, Ecology, among other areas)
News on SIMULA, Implementation and Development
SIMULA and other OOP Languages

Event Driven Visual Programming in PetriSim Environment

Jaroslav Sklenar
Department of Computer Information Systems
University of Malta, Msida, Malta
web: http://www.cis.um.edu.mt/~jskl
e-mail: jskl@cis.um.edu.mt

Abstract

The paper describes simulation in PetriSim environment. PetriSim is a general discrete simulation tool of PC platform whose only requirement upon the user is accepting Petri and Time networks as the system description tool. The graphical editor of PetriSim offers a very fast and user friendly creation of the Petri network, that models the basic logic and relationships in the model, especially cooperation and synchronization of the activities. In the next step the user writes the user code in the Turbo Pascal 7 language (the implementation language of PetriSim) that implements user specific aspects of model behavior and all operations on data that can not be expressed by the Petri network. Also the time is introduced and so in fact the model's skeleton is a Time network based on firing duration concept (firing of transitions can take any - possibly random - time, that is generated by the user). User code is located in a Pascal unit, because Object Pascal does not have nested classes. In version 2 of PetriSim the user code was written "off-line" using the Turbo Pascal Integrated Development Environment (IDE). After creating some models it became obvious, that a closer connection between the network (graph on the screen) and the code (text editor window) will help a lot. In fact this is the basic idea of *Visual Programming* tools like for example FoxPro screen editor, Visual Basic, or Delphi Pascal. The user places an icon on the screen and can directly write a code snippet performed at some kind of activation of the icon (for example after pressing the button icon). Because activating of code snippets is associated with certain events (like pressing a button by mouse), this way of programming is also called *Event Driven Programming*. These ideas have now been implemented in PetriSim together with global modification of properties of Petri network objects using inheritance.

1. Introduction

Paradigm Oriented Simulation is based on a certain concept (abstraction, paradigm) which is used as a tool for describing the system simulated, as a base for creating the simulation model and as a support of other steps like model verification, experimentation, documentation, etc. A commonly accepted tool for simulation of discrete systems are Petri networks and especially their various modifications. Because of limited scope of the paper, it is supposed, that the reader knows the very basic ideas of Petri networks, that are summarized in [4]. For more details see [1] and [2]. One of the Petri net based simulation tools is PetriSim. PetriSim has originally been created as a graphical editor and simulator of Petri networks - see [3]. After incorporating time by the so called Time networks together with the possibility of writing user code associated with events in the network, version 2 of PetriSim became a generator of discrete simulation models based on Time networks paradigm, that is the only (very general) requirement on the way models are viewed and expressed by the user - see [4]. Several models have been created to get some experience and to find PetriSim drawbacks and limitations.

For example in [5] there is a brief description of a model of medium complexity, that simulates a packet switching computer with several input and output lines, randomly generated paths of packets, and a limited number of memory buffers. The model provides all typical outputs like statistics of queues, utilization of servers (sending modules) and time spent by packets in the model. Similar results could be obtained from a rather complex program written in a general high level language (like Pascal) or much simpler and shorter, but still not a trivial program written in a discrete simulation language (like Simscript or Simulation of Simula). Using PetriSim the user has to write only few short snippets in Turbo Pascal expressing the model's behavior together with few procedures performing common things like reading system parameters, initialization of supporting objects, and experiment evaluation.

The models created have proven, that in PetriSim simulation programs can be created relatively fast without much programming, because the time control - the most difficult part of discrete simulation - is included in the underlying Time network and does not impose any requirements upon the user. The user just generates firing delays, that is supported by a unit, that contains objects whose methods return random numbers with practically any distribution. Other units support working with statistically observed variables and linked lists that model queues - see more details in [4].

2. Programming in PetriSim

The main feature of programming with PetriSim is *Event Oriented* approach. The user writes a code that is associated with events in the Time network. The events involved are starting and finishing firing of transitions. Another code is performed when the experiment starts (model initialization) and when the experiment ends (evaluation, displaying and storing results). This of course also needs some declaration of model associated data - mostly instances of objects declared in the above mentioned units. So the user code is made of a global part (associated with the model as such) and snippets related to particular transitions. In version 2 of PetriSim, the user code was written off-line in Turbo Pascal IDE. This obviously involves frequent transfers between Pascal's IDE and PetriSim. PetriSim can be started directly from the IDE, but still the frequent transfers caused inconvenience and what's worse, the user might loose the connection between code snippets and Time network's transitions. In fact to create the model it was almost necessary to print out the network graph and to use it as a reminder when writing the Pascal snippets.

To avoid these problems PetriSim has been further improved to make the relationship between network events and code snippets directly visible. Clicking a network's transition by the right mouse button (its icon is a short thick line) opens a menu window that enables direct writing of code snippets that are activated by starting or finishing firing of the transition. Other global menu options enable writing the initialization and the evaluation snippets related to the whole network. This is typical for *Visual Programming* environments based on direct and clear association between code snippets and graphical icons on the screen. PetriSim (so far) does not contain any native text editor. The user can choose her/his favorite text editor that is activated from PetriSim. This has brought another problem - the memory. In real mode Turbo Pascal programs usually use only the conventional memory (640K). If there are many memory resident programs in RAM together with PetriSim (about 250K) and a reasonably sized heap (Turbo Pascal's dynamically allocated memory), it might happen, that some memory demanding editors can not be activated. This problem can be solved in several ways. To allow future expansions of PetriSim that will be directed to network modules and hierarchy, we are using now the Protected mode offered by the Borland Pascal IDE. In this mode the whole memory is available.

3. Individualization of Object Instances.

The Object Oriented Programming (OOP) paradigm (including Super OOP) does not address the problem of *modifying object instances at code level*. A *class* of a typical OOP language declares a pattern that is used to generate instances that all have the same code (methods and possibly life) and the same attributes (and possibly parameters). (Note: because in Object Pascal object instances do not have life and parameters, from now onwards we shall consider only methods and attributes.) The only possible form of *individualization* of object instances is storing different values to attributes and to test them and behave accordingly when performing the object's methods.

There is one exception (that I know) to this common OOP limitation - the LOGOS language [6]. In LOGOS the instances have their own code generated from the common pattern, that can be further edited. So the object instances generated according to a certain class pattern can eventually behave completely differently, the only common thing that is supposed to remain are types and names of attributes - otherwise the notion of a class would have no sense any more. This approach has obvious negative consequences in the total amount of code and the host language LOGO prevents LOGOS to be used as an implementation tool of simulation packages like PetriSim.

Another case are "objects" of visual tools typical for database environments like FoxPro screen editor and similar. These objects should not be considered as object instances in OOP sense, but there exist similar features. Some properties are common - for example all *Push Buttons* look very similar, react upon the same events (pressing the mouse button) and can be modified by different labels, color, size and similar attributes. What is more important, they can have different associated code snippets that are written by the user. Typically such objects do not possess inheritance capabilities. So the notion of class is mostly not used with them at all.

So excluding LOGOS there are two extremes: not OOP "objects" of visual tools, that all have (at least from user's point of view) different code and OOP object instances, that all have the same code and that can differ only in values of attributes. The question is: *do we need anything in between* ? After getting some experience from PetriSim I am convinced, that a sort of hybrid would be very useful. First of all we definitely need all features of OOP objects. That has been proven beyond any doubt many times. But some individualization at code level would be also very useful. So far the only behavioral individualization can be achieved by testing values of attributes by code, that is the same for all instances. Let's take the transitions of PetriSim. Transitions (like other network components) are instances, that have been defined (consecutively specialized) at three inheritance levels - see later. They have many common features. They have the same type of icon, the same attributes like name, screen position, size, color, etc. and the same methods like initialization, displaying and clearing itself on the screen, testing if the transition is enabled, etc. On the other hand, from the modeling point of view, they represent various objects of the simulated system and thus they might be (and are) supposed to behave differently. For example starting firing that takes certain time can model a server of a queuing system. Starting service may involve removing the first customer from certain associated queue including updating the queue statistics, setting the server status to "working" also with updating the statistics, and storing a pointer to the customer being served. All this is application dependent. Some transitions (for example all servers) do practically the same on different objects, some do things that are completely different. Still all of them have many common things including methods.

PetriSim implements the code individualization of object instances (so far it seems to be useful only for transitions) in the following way. Because the implementation language Turbo Pascal 7 obviously does not offer any direct support for this kind of individualization, it has to be done "off-line" not using the OOP principles. This is of course only an internal, implementation point of view. The user will see the modified code snippets as integral parts of the object instances. The implementation is straightforward: When starting firing of a transition, PetriSim activates a procedure with two parameters: transition name and transition number. Transition number is generated automatically when creating a transition and can not be modified by the user. The procedure uses the transition number in a *Case* statement, that passes control directly to code snippets that have been created by the user for particular transitions. Of course a code snippet can be empty. This is the basic idea that is explained in more detail in the next chapter.

4. Code Generation

Code snippets are written in the Turbo Pascal 7 language, that offers an elegant way how to incorporate code into ready programs by the so called *include files*. There is one limitation - include files must contain complete routines. That's why the simplest

(I hope) solution is this: each object instance (transition) has two associated routines that contain the snippets activated when the firing of the transition starts and when the firing ends. From Pascal's point of view these routines are procedures without parameters with default names. For a transition with number *n* the names are *_PSn* for the starting snippet and *_PEn* for the ending snippet respectively. The procedures have no parameters because the user knows what transition the procedures belong to, the procedures are not allowed to modify anything inside PetriSim, and any relation between the transition and any kind of user data is completely the user's responsibility.

Upon user's request PetriSim generates default empty procedures that are supposed to be modified (edited) by the user. In fact a procedure like the following one is immediately after its creation displayed in the text editor's screen:

```
Procedure _PS1;  { Starting snippet of the transition t1 (Sender1) }
Begin
End;
```

Most of the snippet procedures are very short because the user's activity is typically just activation of methods of supporting objects. Snippet procedures are called by another automatically generated procedure like the following one, where the comments in (* *) are not generated:

```
Procedure Firing_Starts;   { Starting snippets }
  {$I %%S1.PAS   Procedure _PS1 }
  {$I %%S2.PAS   Procedure _PS2 }

        etc.   (* One file for each procedure called in the procedure body *)

Begin
  Case TNum of             (* TNum is the transition number - global here *)
      1: begin  {Line1}    (* Line1 is the name of the transition *)
           _PS1;
         end;
      2: begin  {Line2}
           _PS2;
         end;

      etc.

  End{Case}
End;
```

The above procedure is supposed to be called directly from the user's procedure reacting upon the starting firing event (the only difference for the ending firing event is the letter "E" instead of "S" in the above names of procedures and files). This is the generic user procedure distributed with PetriSim, that need not be modified in user models at all:

```
Procedure  FiringStarts (Var s : String;  TNum : Integer);
{ Starting firing the transition "s" with the number TNum. }
{ Including procedure Firing_Starts. Don't change next line ! }
        {$I %%S.PAS }
Begin
  { Here is the model specific behavior: }
  Firing_Starts;          { Don't change this line ! }

  If PNTime>MaxTime then Terminate;

  If Confirmation then
       Step('Starting Firing of ' + s)
  Else begin
       GoToXY(5,6);
       Write('Current Time = ', PNTime:10:3);
  End{If}
End;
```

Explanation of symbols:

FiringStarts	= user defined procedure activated when the firing starts (similar procedure *FiringEnds* is activated when the firing ends).
Firing_Starts	= automatically generated procedure that calls the starting snippets. It is stored in the file *%%S.PAS*.
PNTime	= PetriSim function that returns the current time.
MaxTime	= User real variable that is supposed to contain the duration of the experiment. The value is read during experiment initialization, but of course it is possible to modify the termination condition in any way.
Terminate	= PetriSim procedure that stops the experiment.
Confirmation	= User Boolean variable that defines the mode. If the value is true, the above procedure calls another generic procedure *Step*, that displays the network in the current status (that contains all information about the future events in the network). The value of this Boolean variable is typically selected by the user when starting the experiment. Note that for the value false the procedure just displays the current time on the screen, that shows the progress of the simulation experiment.
Step	= Procedure that displays the network in the current status (the procedure is supplied in generic form and need not be modified). Step mode is very useful and in fact represents a debugging tool that works at highly symbolic network level.

Note: Actually all visual tools have to solve the problem of individualization of the screen components, for which OOP does not offer an elegant solution. For example the authors of Delphi Pascal have solved the problem in the following way: let's say the user creates the push button *Button1* on the form *Form1* and defines the code to be performed when clicking the button by the mouse. As a result Delphi Pascal generates code whose extract is this:

```
type  TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
       private
        { Private declarations }
       public
        { Public declarations }
      end;

var   Form1: TForm1;
```

Note that the screen object *Button1* is an instance of the class *TButton*, but the associated code snippet (procedure performed on clicking) is not its method. There is no *TButton's* method *Click*, that might have been expected by an OOP enthusiast. Instead, Delphi Pascal generates a method *Button1Click* of the subclass *Tform1*, that from Pascal's point of view has nothing common with *Button1*. The only common thing is the same owner - the subclass *Tform1*. In fact the procedure is later declared as *TForm1.Button1Click*. So the association between *Button1* and *Button1Click* is done off-line by the Delphi environment, not by the OOP facilities of the Delphi host language Object Pascal. PetriSim applies a simpler approach based on include files, but like in Delphi, user modified snippets are related to the objects (events) only in user's mind, not at the host language level.

5. Specialization of Network Components

Another way how to create user specific models is based on applying inheritance, because PetriSim has been created using consistently OOP facilities of the Turbo Pascal 7 language. Basic network components like places, transitions, arcs, and network as such are all classes that can be (and are) created in several (currently three) inheritance steps, one of them being written by the user in the unit, that contains the user specific code. So there are two ways how to incorporate specific features of PetriSim user models: one based on *inheritance* (more general), another based on *individualization* of object instances achieved by connecting user code snippets with events in particular instances (transitions). These two ways complement each other to give the user a power and flexibility in creating user models. Because inheritance modifies all components of the given class, it is primarily intended as an implementation tool that will be used to incorporate various high level network models (extensions to the original Petri networks).

6. Typical Screens

Time networks based simulation in PetriSim starts by creating a Time network whose purpose is timing and synchronization of all events in the model. The rest is done by user routines that are activated by events in the Time network (starting and finishing firing of transitions) and also at the beginning and at the termination of the simulation run, that enables user initialization and user evaluation of the experiment. Figure 1 shows a Time network abstraction of a packet switching computer. The screen has been captured in the so called *screen editing mode* used to edit network properties. The user has activated a menu, that contains these options: **Unit globals** (to be just activated) opens a text editor window with the whole file *USER.PAS* (the unit with user code). Here the user declares the user data and writes the initialization and the evaluation code. **Generate** is the option that generates the code related to user snippets described in the above chapters. The options **Starting snippets and Ending snippets** open a text editor window with the include files that contain the procedures activating the starting and the ending snippets. These two options are used for debugging to modify temporarily the activities connected with network events. Editing of these procedures will be lost by next generating. The option **Help** explains the principles of creating PetriSim user models.

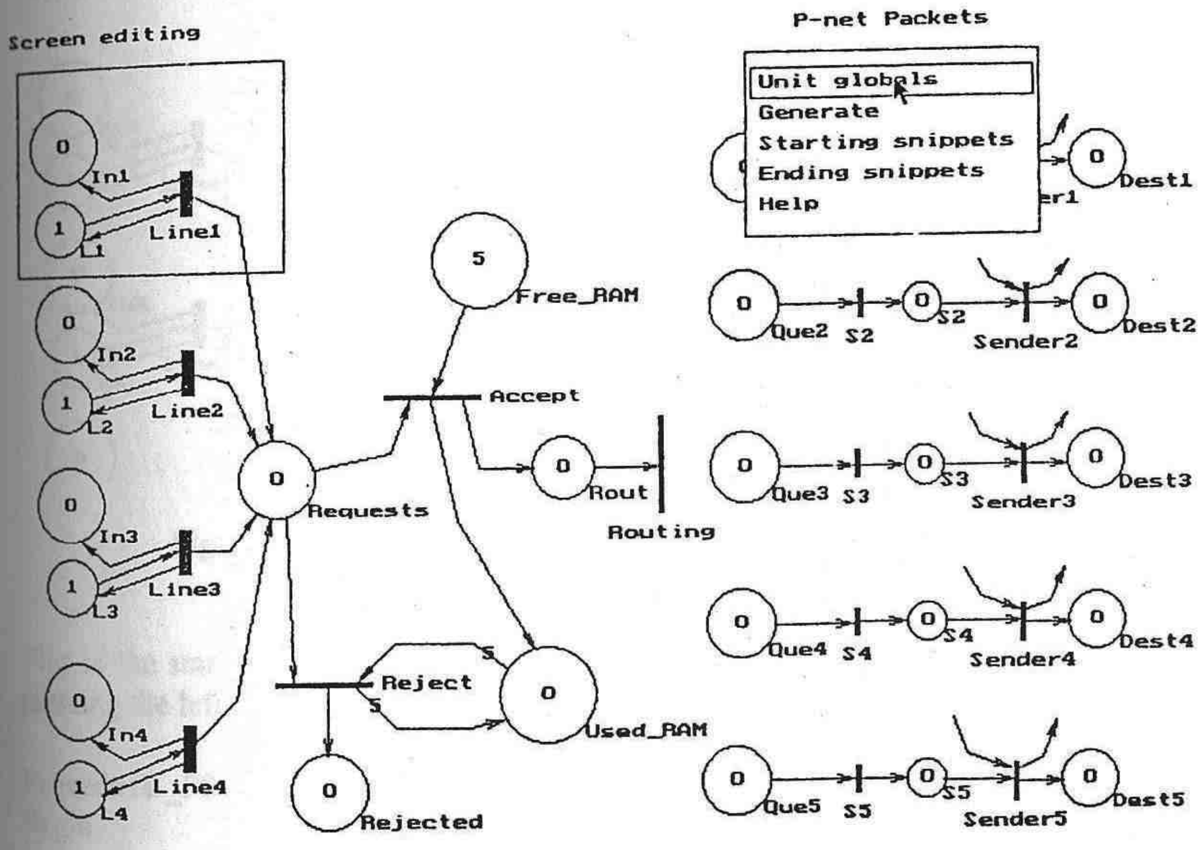


Figure 1: Time network model of a packet switching computer

The two places (*In1* and *L1*) and the transition *Line1* in the upper left corner (the rectangle is not a part of the screen) model the arrival of packets from the 1st input line. The place *In1* is just a counter used to count the packets - each firing of *Line1* increments its initially zero marking by 1. The actual arrival is represented by finishing firing of the transition *Line1*. A token is added to the place *Requests* that models packets that have arrived and will be either stored or rejected if there is no free memory buffer. Duration between two arrivals is represented by the firing delay of the transition *Line1*. The token in the place *L1* keeps the transition *Line1* firing until the experiment ends.

To get quantitative results from the model, these user defined data have been attached to the network model: each packet is represented by a record that keeps the time the packet has arrived (to compute the time spent in the system) and a randomly generated destination. Packets waiting for RAM are stored in a queue called *QI*. Figure 2 is the upper part of a screen captured after clicking the transition *Line1* by the right button. This opens a menu with obvious options intended to modify transition properties. The last two options open a text editor window with the starting and the ending snippets.

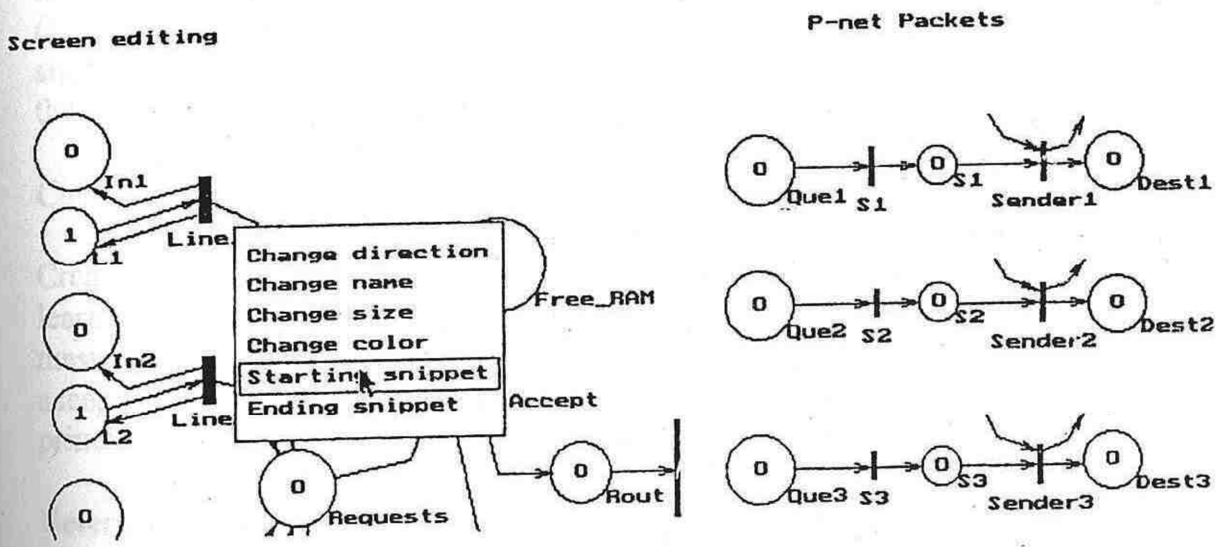


Figure 2: Creating snippets connected with the transition *Line1*

This is the starting snippet, that is shown in a text editor window immediately after pressing the left button in the situation in Figure 2:

```
Procedure _PS1;  {Starting snippet of the transition t1 (Line1)}
Begin
      FireDelay(Line1.RLValue);
End;
```

Where *FireDelay* is the PetriSim procedure used to generate firing duration. Its parameter is a real delay. *Line1* is an object instance, whose method *RLValue* returns the random value (during initialization the object is given the distribution function). In the above procedure the user writes only the procedure statement. This is the ending snippet:

```
Procedure _PE1; {Ending snippet of the transition t1 (Line1)}
Begin
        New(Packet,Init);                  { Creating a packet }
        Packet^.Arrival := PNTime;         { Its arrival time }
        Packet^.Dest := Line1D.IValue;     { Its random destination }
        Packet^.Into(QI);                  { Entering Input Queue }
End;
```

A packet (*Packet* is a *Link* object with the same methods as the *Link* of Simset) is created, the current time is stored in its attribute *Arrival*. *Dest* is its random destination (*Line1D* is a random object whose method *Ivalue* returns a random integer value - here the number of the output line). Finally the packet is inserted to the queue (*QI* is a *Head* object with the same properties as the *Head* of Simset).
After creating or editing snippets it is necessary to generate Pascal code. Then the user leaves PetriSim to recompile the whole program. Each version of PetriSim (PETRISIM.EXE) is one user model, that can be also used as a general editor and simulator of Petri networks. User code is activated only during network Simulation in the User experiment mode.

Conclusion

Creating simulation models in PetriSim is relatively easy and still there is no need to learn a simulation language. That's why it can be used especially for education, where mastering a special simulation language during a one semester course is not acceptable. The students can learn both discrete simulation as such and also the principles of Petri network based modeling.

References

[1] Peterson, J.L.: *Petri Net Theory and Modeling of Systems.* Prentice Hall Inc., 1981, Englewood Cliffs, N.J.

[2] Diaz, M.: *Modeling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models*. Computer Networks, 1982, no. 6, p. 419-441.

[3] Sklenar, J.: *PetriSim - Environment for Simulation of Petri Nets.* Object-Oriented Modeling and Simulation, 1994, Prague, p. 214-221.

[4] Sklenar, J.: *Discrete Simulation and Time Networks.* Object-Oriented Modeling and Simulation, 1996, Clermont-Ferrand, p. 57-67.

[5] Sklenář, J.: *Time Net Based Model of a Packet Switching Node.* Communications, Signals and Systems, 1996, Brno, p. 469-472.

[6] Weinberger, J.: *Logo, Simulation and Object-Oriented Programming*. 3rd European Logo Conference, 1991, Parma, p. 219-225.

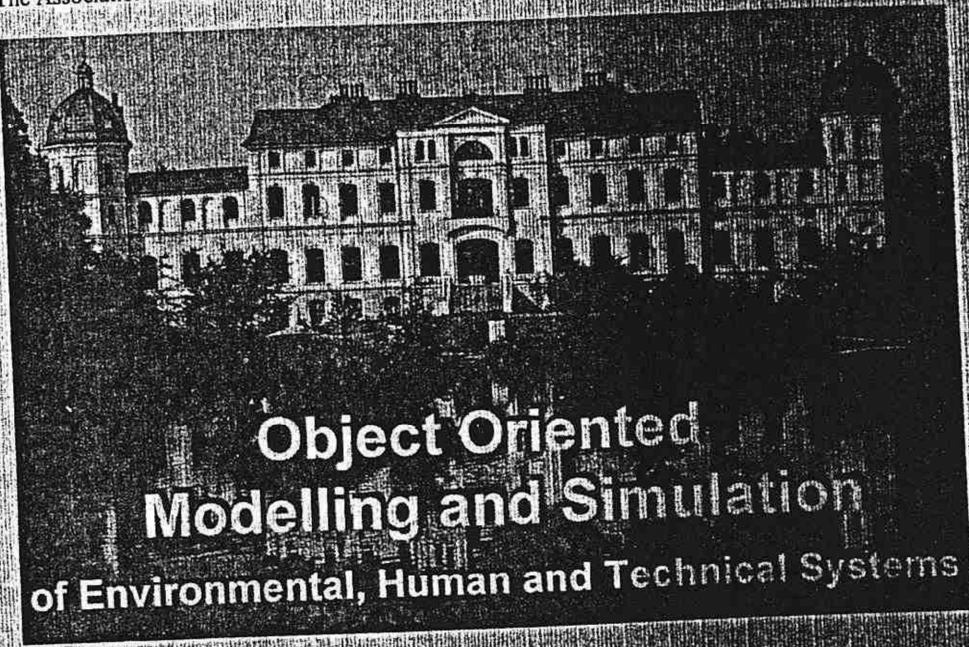
J54

ISSN 1102-593x

ASU Newsletter

Vol. 24 Supplement Oktober 1998

A Publication of:
The Association of SIMULA Users



Object Oriented Modelling and Simulation of Environmental, Human and Technical Systems

PROCEEDINGS

of the
24th Conference of the ASU 30. Sept. - 2. Oct. 1998
at SALZAU (Schleswig Holstein, Germany)

Organizers:
Ecology Center, University of Kiel
Association of SIMULA Users

Using Inheritance to Implement High Level Petri Nets

Jaroslav Sklenar, Ph.D.
Department of Statistics and Operations Research
University of Malta, Msida MSD 06, Malta
Phone: (00356) 3290 3070 Fax: (00356) 312 110
Web: http://staff.um.edu.mt/jskl1/
E-mail: jskl1@stator.um.edu.mt

Abstract

The ASU'96 paper "*Discrete Simulation and Time Networks*" has introduced the second version of the simulation package PetriSim (written in the Turbo Pascal language) that contains a graphical editor of Petri Nets and that supports creation of user defined simulation models based on the Time Nets paradigm. (Time Nets are based on the original Petri Nets and their purpose is the introduction of time to network models). The ASU'97 paper "*Event Driven Visual Programming in PetriSim Environment*" dealt with direct association between network objects (Petri Net transitions) and user code snippets, that is typical for *Visual programming*. It has outlined a possible solution to the problem of "individualization" of object instances at code level - at least from the user's point of view, because existing Object Oriented Languages don't address this problem. So in the current version of PetriSim the user can write simulation models that are originally expressed by the language of Petri Nets and later modified by adding user code snippets modeling specific features of the particular model. There is another possible way to specialize the original very general language of Petri Nets that itself has very limited modeling power. PetriSim is based on Object Oriented capabilities of the host Turbo Pascal 7 language. All network objects (places, transitions, and arcs) are object instances whose classes (objects in Object Pascal terminology) are in fact written by the user. PetriSim contains generic classes that implement the basic Petri Net behavior. The user's part of the model has to contain declaration of subclasses, whose methods are actually used by PetriSim to perform operations like testing if the transition is enabled or not and performing activities associated with occurrence (firing) of the transition. When starting work on the model, there is a generic version of the unit *User* (the user's part of the model), that contains degenerate versions of all methods, so a user's model based only on the original Petri language will be very simple. On the other hand each user can implement her/his own modification of Petri Nets by writing appropriate methods. The above two methods can be combined. User subclasses implement features common to all objects of a certain type (such as all transitions). Code snippets associated to particular objects can implement the details of individual objects. The paper shows how these ideas were used to implement nets with inhibitor arcs and nets for simulation of queuing networks.

1. Introduction

Because of limited scope of the paper, it is supposed that the reader knows the basic ideas of Petri Nets. The original Petri formalism (that is called today Place/Transition or Pl/Tr Nets) represents an analytically very powerful graphical language with very limited modeling power. There is no time, the only quantitative parameters are numbers of tokens in places. That's why there are very many various extensions - a lucid classification of Petri Nets based formalisms is [2]. Two examples of high level Petri Nets are Coloured Nets and Queueing Nets. Coloured Nets - [3] represent a general extension based on associating data with tokens that also makes it necessary to generalize conditions that enable transitions to fire and the operations involved in firing. Queueing Nets - [4] are more specialized. As the name says they are supposed to be used in modeling of queuing systems. The basic idea is a complex place - node that contains both a queue and the associated server(s). Both formalisms are supported by software tools. There is one common problem (that also applies to other models) - very complex definitions that make it very difficult to use these tools in simulation practice. The purpose of these complex definitions is to keep Petri Net properties (that are necessary if the tool is supposed to be used for system analysis and verification) together with increasing modeling power (functionality). There is another trend: attempts to combine Petri Net principles with the ideas of Object Oriented Programming (OOP). One approach based on using Smalltalk is [5] that also contains other references. PetriSim is another example of an approach how to combine Petri Net and OOP ideas.

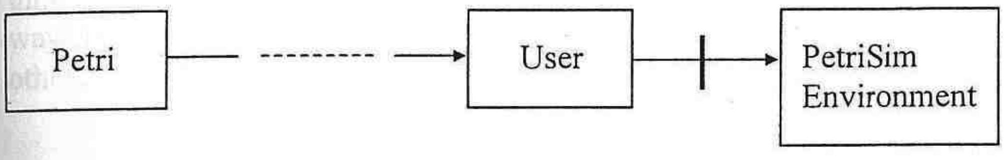
2. PetriSim Networks

PetriSim is a simulation tool that uses Petri Net based language for system description only, not for analysis and/or verification. There are three main reasons for this:

- First there are enough Petri Net tools that can be used for this purpose - see [6]..
- The second reason is based on the fact, that a "good" Petri Net (that represents a certain simulated system) is a necessary but not sufficient condition that the model is acceptable. A good Petri Net can represent a totally wrong system from the simulation point of view. For example a system that differs significantly from the part of objective reality that is being simulated.
- The third reason is practically unlimited freedom in increasing modeling power in case of not insisting on certain formal properties.

I don't think there are any commonly accepted properties that have to be kept for a formalism to be still called "High Level Petri Net". Fortunately PetriSim (so far) does not face this problem - few simple extensions have increased the modeling power very much and the network model is still very close to the Pl/Tr Nets.
Implementation of PetriSim networks is based on three classes (called "objects" in Object Pascal) that represent knowledge of ***places***, ***transitions*** and ***arcs*** (lines of any

shape connecting places and transition). The elementary knowledge on Pl/Tr Nets is stored in the Pascal unit *Petri* - see Appendix 1, where the underlined properties and methods represent the formalism, the rest is used by the graphical editor and to support simulation in various modes. There is only one extension to the basic Pl/Tr model: the firing duration. It can be used only in the so called *user simulation mode* where a user code snippet can generate a time delay between removing tokens from input places and adding tokens to output places of the transition. This is the way in which PetriSim incorporates time to the networks. The properties and methods have self-explaining names, those interested in details can download the source code from [7]. To enable extensions of the elementary Pl/Tr Nets, PetriSim uses another unit called *User*, that contains declaration of user classes **UserPlace**, **UserTransition**, **UserArc** that must have (among others) exactly the same properties and methods as those listed in the Appendix 1. So the basic inheritance hierarchy is this:

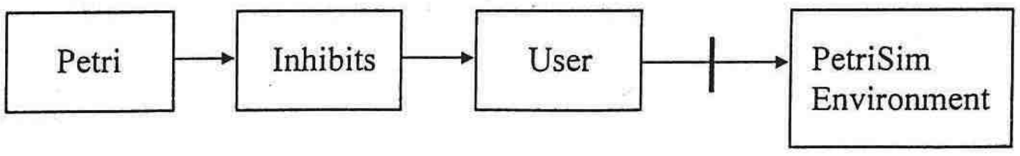


Where the dashed line means any number of inheritance steps (written typically as Pascal units) that are supposed to add a certain functionality to the original classes, but that will always keep the interface (thick line) needed by the PetriSim environment. So there are two ways how to add functionality in PetriSim:

- User code snippets that can be associated to transitions. User code can be called at the time when the firing of the particular transition starts and/or at the time when the firing of the particular transition ends. This itself represents a powerful simulation tool. For example the model described in [1] is based on this approach only (user classes in the unit *User* are empty, with only inherited properties and methods). All PetriSim simulation models need user code snippets, because this is the way to generate firing duration.
- Classes increasing functionality of all network objects. Obviously these two ways are supposed to be used together.

3. Pl/Tr Nets with Inhibitor Arcs

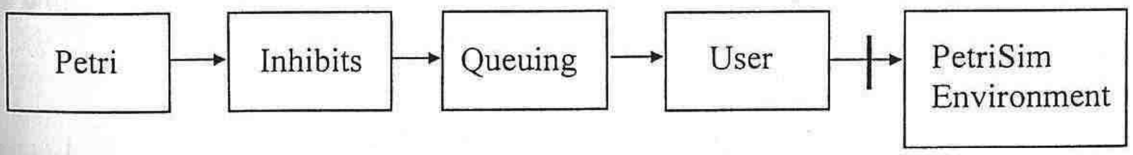
There is a well-known problem associated with Pl/Tr Nets - they are not able to test whether a place is empty or not. There are several ways how to tackle this problem - [8], the most straightforward are the so called ***inhibitor arcs***. These are input arcs to a transition (with multiplicity 1) that unlike the normal arcs end by a small circle - see for example the arc from the place *Q1Capacity* to the transition *New_lost* in Figure 1 (will be explained later). Then the necessary condition for the transition to be enabled is the fact that the place where the inhibitor arc starts is empty. A transition can have any number of normal and/or inhibitor input arcs. It has been shown that Pl/Tr Nets with inhibitor arcs (generally with the ability to test for zero) can simulate Turing machines. So we have a modeling tool that can model any system. Implementation of Inhibitor arcs in PetriSim was very simple. A unit *Inhibits* is placed between *Petri* and *User*:



It contains declaration of the classes that redeclare the methods *AShow* and *AErase* of the arc and the method *Enabled* of the transition in obvious way. Inhibitor arcs have the weight (multiplicity) zero which is the way how they are defined by the user. The unit *User* in this case declares descendants of the classes declared in *Inhibits*. The PetriSim environment does not see (and must not see) any difference. To make an inhibitor arc, the user draws a normal arc and then sets its multiplicity to zero. By the way, the multiplicity is integer, so it is possible to define practically any number of other types of arcs using the negative values.

4. Queuing Networks

Queuing networks (made of more than one queues and associated servers) are typical simulated systems because mathematical models of Queuing theory for networks are very limited. Almost all practical systems don't satisfy the strong assumptions of Jackson and BCMP networks. Queueing Nets - [4] represent one approach how to model queuing networks by High Level Petri Nets. PetriSim nets try to achieve similar functionality in much simpler and straightforward way (obviously by giving up the analytical capabilities of the nets). The classes oriented to queuing networks are declared in the unit *Queuing*:

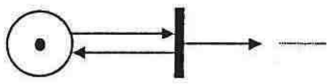


The classes of *Queuing* are subclasses of those declared in *Inhibits* because inhibitor arcs are very useful (they could have been descendents of basic classes of *Petri*). Note that again the PetriSim environment sees no difference so it creates and draws all *Queuing* icons by the same graphical editor as in both above cases. I believe that queuing networks are made of (at least) these basic objects:

- Generators of customers (with random intervals between arrivals)
- Queues (with various queuing disciplines and possibly limited capacity)

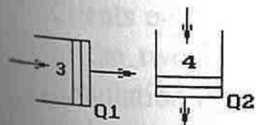
- Servers (with random service duration and the possibly to connect more servers to one queue - multichannel servers)
- Decision points (where the customers take one of alternative routs).

Because of practical reasons it is also required (to simplify user code as much as possible) that the objects automatically collect and compute basic statistical parameters like statistics of queue lengths, utilization of servers and number of generated customers. To make the PetriSim nets easy to understand the different types of objects should also have different icons. Another requirement was the maximum possible use of Petri Net graphical language, because some type of association is obvious - for example place as a queue, transition (with firing delay) as a server. To model the above parts of queuing networks the Petri Net objects can also be combined - for example a well known pair place - transition working as a generator of tokens:



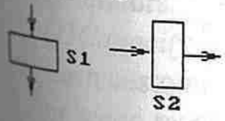
After some experimentation a certain number of objects have been defined. Because there is only one class of transitions and one class of places, different kinds of these objects are distinguished by different values of their so called *Type*, that can be changed by the user in PetriSim environment. Places and transitions are always created with type zero, which means (from the user's point of view) the original place and transition as declared in the unit *Petri*. Some types of objects differ only in the icon, some have modified behavior. So for example the method *TShow* that displays a transition tests the value of the property *TType* and displays the icon according to its value. Similarly the method *EndFire* that normally adds tokens to all output places tests the value of *TType* and in case of branching it adds token(s) to only one output place. PetriSim environment uses all methods in unified way as if there was no difference between object instances. Currently there are these kinds of objects (together with the standard place - circle and the standard transition - thick line):

Places

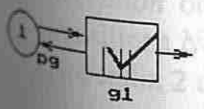


Horizontal and vertical queues

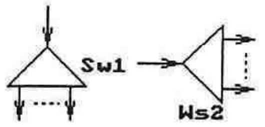
Transitions



Timed transition (with firing delay)



Generator (here shown with a place necessary to start it)



Horizontal and vertical branching.

All places have marking statistics. This is implemented by another property, that is an instance of a statistical object. At each updating of its value it computes the time integral. This is then used by its methods that return the average, the maximum and the minimum values, the variance and the standard deviation. Places have one more method *AddTokens* that together with updating marking also updates this statistical object.
Transitions keep their utilization (portion of time the firing was on) and the number of completed firings. The former is a statistical object that is set to one when firing starts and cleared to zero when firing ends. Average is then the utilization if the transition is a server. Number of firings is a counter incremented by each firing completion. It is used as a number of generated or served customers.
The only object with modified behavior is branching. It adds token(s) to one output place only. There is an integer property *Branch* whose initial value is zero. In this case the output arc/place is chosen randomly, all with the same probability. User code activated at the end of firing can change the value of this property to implement any kind of deterministic or not uniform random branching. PetriSim has a unit with routines used to generate random numbers with theoretical and general distribution. Routines of this unit are also used to generate firing delay.

5. Example Model

The system simulated expressed by a net in Figure 1 is a bank that operates in this way: New customers wait in the queue *Queue1* with limited capacity, so they may be lost. They are then served by a receptionist (transition *Reception*) and either leave (probability 30%), proceed to the teller's queue *Queue2*, that also has limited capacity (probability 60%) or proceed directly to the *Queue3* of unlimited capacity served by two cashiers (transitions *Cash1* and *Cash2*). Customers served by the teller (transition *Teller*) either leave (probability 40%) or proceed to the cashiers (probability60%). Clients enter directly the teller's queue (if not full). Random intervals between arrivals of the two types of customers and random duration of all four servers are known. Simulation is supposed to find numbers of lost and served customers, lengths of queues and utilization of servers. Most constructs in Figure 1 are self-explaining. The two generators generate new customers and clients. The places *pnc* and *pcl* just activate the generators. Limited capacity of the two queues is modeled by tokens in places *Q1Capacity* and *Q2Capacity* respectively that also include the customer being served. Each customer entering the particular queue takes one token (by starting firing of *tnacc* or *tcacc* respectively) and returns the token after being served (by finishing firing of *Reception* or *Teller* respectively). If the queue is full, the inhibitor arcs enable the transitions *New_lost* or *Cl_lost* respectively that remove the customer from the system. Appendix 2 contains all user code snippets (except experiment evaluation).

Results are in Appendix 3. They are generated by a simple procedure (made of 47 lines) that repeatedly gets the statistics of objects and displays the value(s). For example the first line of the table with statistics of queues is generated by this code:

```
PStatistics('Queue1',V,A,Max,Min,Va,S);
Writeln(R,'Queue1    ',A:14:3,Max:11:0);
```

Where *PStatistics* is a procedure (supplied with the generic version of the unit *User*) that returns the current value (*V*), the average (*A*) - here the average queue length, the maximum, the minimum, the variance, and the standard deviation of the marking of the place whose name is the first parameter. Similarly the first line of the table with utilization of servers is generated by this code:

```
TStatistics('Reception',U,N);
Writeln(R,'Receptionist ',U*100:10:2,N:18);
```

Where *TStatistics* is a procedure (supplied with the generic version of the unit *User*) that returns the utilization (*U*) and the number of firings (*N*) of the transition whose name is the first parameter. Appendix 3 contains data from two simulation runs that are both 10000 time units (minutes) long. The first columns contain data for the system in Figure 1. Note relatively big number of 90 lost clients. The second experiment tried to find the necessary queue lengths for the bank not to loose any clients. So the capacities of the two queues were increased to a very big number (10000) and the experiment was repeated. The results are in the second columns. They show that the system is stable, the only problem might be the teller's queue (maximum length 30). So the simulation study would suggest adding one more teller.

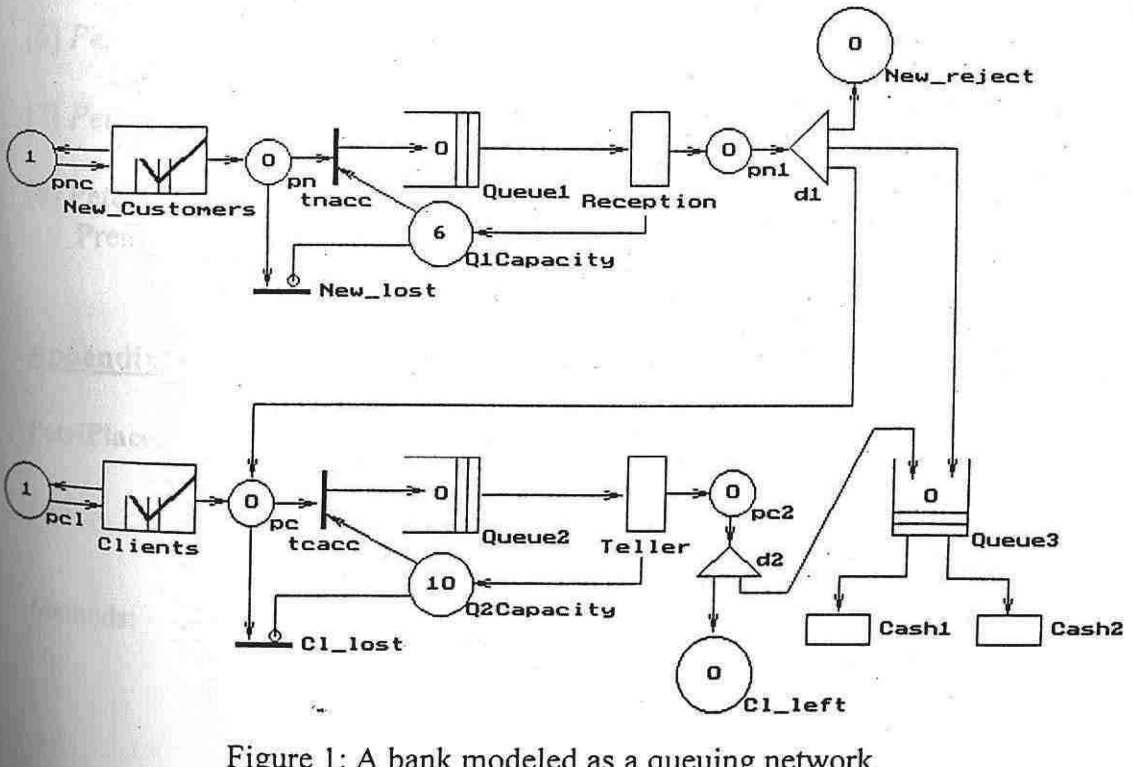


Figure 1: A bank modeled as a queuing network

Conclusion

The example model is not very complicated, but certainly not trivial. Still it was relatively easy to create it - the Appendix 2 contains the whole user written code except a simple procedure that displays the results. The results don't give time parameters (the times spent in queues and in the system). To do this - like in the model in [1] - it would be necessary to write user code to generate clients, to record arrival times, to move clients through the net and to compute the waiting times in appropriate points. All this is not complicated in PetriSim, there are supporting units for all typical operations. The purpose of this paper was to show how to use inheritance to increase functionality of a formal modeling language in clear and well-organized way.

References

[1] Sklenar, J.: *Event Driven Visual Programming in PetriSim Environment* Object-Oriented Modeling and Simulation, 1997, Stara Lesna, p. 170-179.

[2] *General Classification of Petri Nets*: http://www.dsi.unimi.it/Users/Tesi/trompede/petri/class.html

[3] *Coloured Petri Nets*: http://www.daimi.aau.dk/CPnets/intro/

[4] *Queueing Petri Nets (QPNs)*: http://ls4-www.informatik.uni-dortmund.de/QPN/

[5] Ceska, M. - Janousek,V.: *Object Orientation in Petri Nets.* Object-Oriented Modeling and Simulation, 1996, Clermont-Ferrand, p. 69-80.

[6] *Petri Nets Tool Database*: http://www.daimi.aau.dk/PetriNets/tools/db.html

[7] *PetriSim home page*: http://staff.um.edu.mt/jskl1/petrisim.html

[8] Peterson, J.L.: *Petri Net Theory and Modeling of Systems.* Prentice Hall Inc., 1981, Englewood Cliffs, N.J.

Appendix 1: Basic PetriSim classes declared in the unit *Petri*

PetriPlace	Xpos, Ypos, Size	*Position on the screen and size*
	Color, Mcolor	*Icon and marking colors*
	Marking	*Current marking (number of tokens)*
	OldMark	*Recorded marking (used for fast initialization)*
	Name, Ptype	*Place name and type (used later)*
Methods:	PInit	*Place initialization (after creating or loading)*
	PSimInit	*Initialization before simulation in user mode*
	Pshow, PErase	*Displaying/Erasing place on the screen*

PetriTransition	Xpos, Ypos	*Position on the screen*
	Size, Color, Direct	*Size, color, and direction (horizontal or vertical)*
	Name, TranNum, Ttype	*Transition name, number and type (used later)*
	Inlist, Outlist	*Lists of input and output arcs (topology)*
	Firing, FireEnd	*Firing on/off, firing end time*
Methods:	Tinit	*Transition initialization (after creating or loading)*
	TSimInit	*Initialization before simulation in user mode*
	Tshow, TErase	*Displaying/Erasing transition on the screen*
	Enabled	*Whether enabled or not (tests tokens in input places)*
	StartFire	*Activity when firing starts (removes tokens)*
	EndFire	*Activity when firing ends (adds tokens)*
PetriArc (subclass of **Poly**)	Place	*Pointer to the place*
	Transition	*Pointer to the transition*
	Weight	*Arc multiplicity (number of parallel arcs)*
Methods:	Ainit	*Arc initialization (after creating or loading)*
	Ashow, AErase	*Displaying/Erasing arc on the screen*
Poly	Corners	*List of corners on the screen (arc shape)*
	Color, Width, Style	*Color, width, and style of the line*
Methods:	Init	*Polyline initialization (after creating or loading)*
	Create, Destroy	*Create (draw)/Destroy (from RAM) the polyline*
	Show	*Display polyline on the screen*

Appendix 2: User code snippets of the example model

Transition	Firing	Code	Note
New_Customers	Starts	FireDelay(Normal3Sig(15,3))	
Clients	Starts	FireDelay(Exponential(5))	
Reception	Starts	FireDelay(UniformR(10,20))	
Teller	Starts	FireDelay(Normal3Sig(4,1))	
Cash1	Starts	FireDelay(Normal3Sig(10,3))	
Cash2	Starts	FireDelay(Normal3Sig(10,3))	
d1	Ends	x := Random;	
		If x<0.3 then t^.Branch := 1	{ 30% leave }
		Else if x<0.4 then t^.Branch := 2	{ 10% to cashiers }
		Else t^.Branch := 3	{ 60% to the teller }
d2	Ends	x := Random;	
		If x<0.4 then t^.Branch := 1	{ 40% leave }
		Else t^.Branch := 2	{ 60% to cashiers }

Notes:	*UniformR(a,b)*	is the real uniform random number from *a* to *b*
	Exponential(m)	is the exponential random number with mean *m*
	Normal3Sig(m,s)	is the normal random variable with mean *m* and standard deviation *s* limited in the range $m \pm 3s$
	x	is a local variable (declared by user)
	t	is an automatically available reference to transition object instance.

PetriTransition	Xpos, Ypos	*Position on the screen*
	Size, Color, Direct	*Size, color, and direction (horizontal or vertical)*
	Name, TranNum, Ttype	*Transition name, number and type (used later)*
	<u>Inlist, Outlist</u>	*Lists of input and output arcs (topology)*
	Firing, FireEnd	*Firing on/off, firing end time*
Methods:	Tinit	*Transition initialization (after creating or loading)*
	TSimInit	*Initialization before simulation in user mode*
	Tshow, TErase	*Displaying/Erasing transition on the screen*
	<u>Enabled</u>	*Whether enabled or not (tests tokens in input places)*
	<u>StartFire</u>	*Activity when firing starts (removes tokens)*
	<u>EndFire</u>	*Activity when firing ends (adds tokens)*

PetriArc (subclass of **Poly**)	<u>Place</u>	*Pointer to the place*
	<u>Transition</u>	*Pointer to the transition*
	<u>Weight</u>	*Arc multiplicity (number of parallel arcs)*
Methods:	Ainit	*Arc initialization (after creating or loading)*
	Ashow, AErase	*Displaying/Erasing arc on the screen*

Poly	Corners	*List of corners on the screen (arc shape)*
	Color, Width, Style	*Color, width, and style of the line*
Methods:	Init	*Polyline initialization (after creating or loading)*
	Create, Destroy	*Create (draw)/Destroy (from RAM) the polyline*
	Show	*Display polyline on the screen*

<u>Appendix 2</u>: User code snippets of the example model

Transition	Firing	Code	Note
New_Customers	Starts	FireDelay(Normal3Sig(15,3))	
Clients	Starts	FireDelay(Exponential(5))	
Reception	Starts	FireDelay(UniformR(10,20))	
Teller	Starts	FireDelay(Normal3Sig(4,1))	
Cash1	Starts	FireDelay(Normal3Sig(10,3))	
Cash2	Starts	FireDelay(Normal3Sig(10,3))	
d1	Ends	x := Random;	
		If x<0.3 then t^.Branch := 1	{ 30% leave }
		Else if x<0.4 then t^.Branch := 2	{ 10% to cashiers }
		Else t^.Branch := 3	{ 60% to the teller }
d2	Ends	x := Random;	
		If x<0.4 then t^.Branch := 1	{ 40% leave }
		Else t^.Branch := 2	{ 60% to cashiers }

Notes:	*UniformR(a,b)*	is the real uniform random number from *a* to *b*
	Exponential(m)	is the exponential random number with mean *m*
	Normal3Sig(m,s)	is the normal random variable with mean *m* and standard deviation *s* limited in the range $m \pm 3s$
	x	is a local variable (declared by user)
	t	is an automatically available reference to transition object instance.

Appendix 3: Results of the example model

```
EXPERIMENT EVALUATION  (Experiment Length =  10000)


NUMBERS:

Number of Arriving New Customers: 667       Number of Arriving New Customers: 666
Number of Lost New Customers :      2       Number of Lost New Customers :      0
Number of Rejected New Customers: 193       Number of Rejected New Customers: 198

Number of Arriving Clients :        2062    Number of Arriving Clients :        2048
Number of Lost Clients :            90      Number of Lost Clients :            0
Number of Clients leaving early :   938     Number of Clients leaving early :   951

Total number of Served Clients :    1493    Total number of Served Clients :    1544


STATISTICS OF QUEUES:

Queue   Average length   Maximum            Queue   Average length   Maximum
================================            ================================
Queue1       2.421          5               Queue1       3.944          10
Queue2       3.801          9               Queue2       7.609          30
Queue3       0.366          4               Queue3       0.461           5


UTILIZATION:

Server         Utilization [%]   Customers  Server         Utilization [%]   Customers
==========================================  ==========================================
Receptionist       99.43            661     Receptionist       99.46            663
Teller             93.88           2361     Teller             97.39           2432
Cashier 1          72.85            756     Cashier 1          76.79            769
Cashier 2          75.40            737     Cashier 2          77.23            775
```

16TH EUROPEAN SIMULATION MULTICONFERENCE

Modelling and Simulation 2002

June 3-5, 2002
Fachhochschule Darmstadt
Darmstadt, Germany

In cooperation with

Edited by

Krzysztof Amborski
and
Hermann Meuth

A Publication of
SCS Europe

SIMULATION OF QUEUEING NETWORKS IN PETRISIM

Jaroslav Sklenar
Department of Statistics and Operations Research
University of Malta
Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jaroslav.sklenar@um.edu.mt

KEYWORDS

Education, Operations research, Discrete simulation, Event-oriented models, Queueing models.

ABSTRACT

Petri networks and Queueing networks are two abstractions used to represent and analyze large-scale distributed systems. Though their basic ideas and objectives are different, there are areas where a combination of these two formalisms brings new ideas and methods of both qualitative and quantitative nature. The principal problem is the analysis of discrete distributed systems with delays, services and queues that may be formally represented as queueing networks. Mathematical models of queueing networks are based on very strong, often unrealistic assumptions. That's why simulation is mostly the only feasible quantitative analytical method. The paper shows how to use Time Petri nets together with code in a high-level language to create simulation models of queueing networks. The method is especially suitable for education purposes because the only requirements are intermediate programming skills and basic knowledge of Petri nets.

INTRODUCTION

Queueing networks are abstract models of discrete distributed system in which it is possible to identify entities (customers) being processed/served by entities (servers) with limited capacity. Waiting queues are thus formed. There are basically two classes of networks for which exact mathematical models exist – Jackson and BCMP networks. For details see for example (Chao et al. 1999). Both of them are based on strong assumptions. Although other methods, like various approximation algorithms also exist, simulation is often the only feasible method how to obtain the required results. In fact quantitative analysis of queueing systems was probably the most important reason behind the boom of discrete simulation languages in late sixties. There are two ways how to create simulation models of queueing networks. First (traditional) approach is the programming of the model in a discrete simulation language. For models with dynamically changing structure and complex behavior of the entities this is probably the only acceptable way. Alternatively we can use an existing simulator and enter the description of the network as its input data. If the input is graphical, based on drawing the network on the screen and entering numerical parameters of the components, the simulator is called "visual interactive simulation tool (environment)" like for example Arena^{TM} or Extend^{TM}. An on-line web hosted simulator described in (Sklenar 2001) belongs into this category of simulation tools even if the structure is not expressed graphically but in terms of transition probabilities. Petri net based simulators can also be put into this group of simulation tools. Software packages that support work with Petri nets typically contain processors (interpreters) of the particular Petri net modification. What remains is the description of the queueing network as a Petri net, supported by the tool.

Petri and Queueing Networks

The idea to model queueing networks as Petri nets is not new. In fact there is a high level Petri net created just for this purpose. The so-called Queueing Petri Nets (QPNs) - see (Bause 1993) - are based on another high level Petri net formalism called Coloured Petri Nets (CPNs) - see (Jensen 1998). The basic idea of QPNs is the integration of queues and servers into the so-called timed places. So a timed place is basically a single queue multichannel system with service time dependent on token's = customers' color. Qualitative analysis is based on the underlying coloured Petri net, where all timing aspects are neglected. Qualitative analysis (like searching for deadlocks) of queueing networks is probably not as important as it is in other areas of applications of Petri nets like for example verification of protocols. Nevertheless it represents a new tool in the analysis of queueing networks brought by the combination of the two formalisms. Quantitative analysis of QPNs is based on the analysis of the underlying Markov process. Increased functionality and modeling power of high-level Petri nets like QPNs and CPNs is achieved by a (rather complicated) formal generalization of tokens, places and transitions. There is an alternative to this approach: keep the Petri net specification as simple as possible; add functionality by adding code in a high-level programming language. This is the basic idea of PetriSim.

PETRISIM BASICS

PetriSim is a Petri nets tool that can be also used as a generator of discrete simulation models. It is a tool intended to be used especially in education. Originally it has been created as an easy to use editor and simulator of Petri nets. Then time has been incorporated to enable working with the so-called Time nets. Today PetriSim is an open tool with objects supporting the creation of various Petri net based environments. The purpose of this paper is the description of one such tool intended to be used as a visual environment oriented to the simulation of queueing networks. The motivation behind PetriSim development was as follows: creation of discrete simulation models for education in general high level languages is too difficult; mastering simulation languages takes quite a long time and thus can not be incorporated into a one semester course. PetriSim is a free, easy to use alternative to the professional visual simulation tools mentioned above. Together with the creating of discrete simulation models the students are also taught modeling techniques based on Petri nets and their modifications.

PetriSim Networks

There are two languages that are used to create simulation models in PetriSim. First the Petri nets graphical language defines the basic structure and relationships in the model. Details that cannot be incorporated in the Petri net model are then expressed by Pascal (the host language of PetriSim is Turbo/Borland Pascal 7). User's Pascal code is made of a number of very short snippets, because most statements just call methods of objects supplied in supporting units. The graphical language of Petri nets expresses the most difficult part of the discrete simulation: model timing and synchronization. The rest is intermediate Pascal programming. All that students have to learn are basic ideas of Petri nets. Unlike other Petri nets based tools, PetriSim nets are very close to the basic Place/Transition nets - in fact the only extension is firing duration that incorporates time, generalization of arcs and branching transition – see later. So we call the nets Time nets. Firing duration together with Petri net topology create complex timing and synchronizing structures, that are normally programmed by special statements of discrete simulation languages. There are no complicated definitions used in most theoretical high-level Petri nets. Functionality is instead added by user code associated with transitions. This simplicity is possible, because PetriSim uses the Petri net as a descriptive tool only. There is no attempt to use analytical capabilities of Petri nets. PetriSim enables a very user-friendly creation, editing, and simulation of practically any number of Petri nets at a time. Using the mouse to draw on the screen performs most operations. Networks can be stored in a disk, they can be updated and renamed, and so the user can work simultaneously on several versions of one network model.

User Code

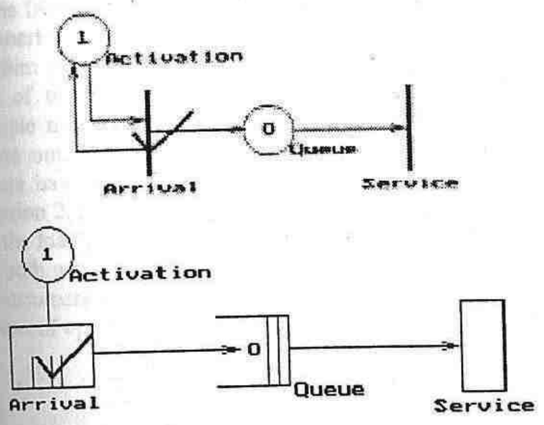
Incorporating user Pascal code into Petri nets is based on the firing duration approach. For each transition the user has the possibility to add the so-called starting snippet activated before the firing starts (its typical use is for example generation of random firing duration). The ending snippet is activated when the firing ends (its typical use is for example branching at output and/or collecting statistics). Writing code snippets is done during network editing – right click on a transition opens a menu that among others offers options to create or edit the two snippets. Their activation opens an editor (selected by the user) with an automatically generated empty code (Pascal procedure). After creating all snippets the whole model together with PetriSim has to be re-compiled which takes negligible time in today's PC. Another code global to the model contains all user-defined declarations together with the model initialization and evaluation of the experiment. Most of the user code is the same in all models, so creating a new one often means just adding a few lines into generic procedures. There are supporting Pascal units for generation of random numbers, automatic collection and computation of statistics and work with linked lists.

Queueing Networks Tool

Generic PetriSim classes are the Petri net place, the transition and the arc. These classes implement the functionality of Place/Transition nets together with methods used by the PetriSim editor and simulator. Time is also incorporated. By using inheritance it is possible to add more functionality that is necessary for example in various high-level extensions to the basic Place/Transition model. First classes implementing Petri nets with inhibitor and testing arcs have been added. These classes have then been used as the super-classes specialized into classes implementing a Petri net based tool for simulation of queueing networks. When using the queueing networks tool, places and transitions can be assigned a so-called type that affects both functionality and appearance on the screen. Icons try to follow conventions used to draw queueing systems – see fig. 1. To simplify the user code as much as possible, the objects automatically collect and compute basic statistical parameters like statistics of the queue length and the queue waiting time, utilization of servers, statistics of firing duration and number of performed firings. Four types of queueing networks objects are available: a place that represents a queue and three transitions used to model generators of customers, activity/delay (server), and branching. Next chapter shows how to use PetriSim places and transitions to represent basic types of single queue systems and how to combine them into queueing networks.

PETRISIM MODELS OF QUEUEING SYSTEMS

Queueing theory texts start with the M/M/1 model. Firing delay in PetriSim networks is program generated, so there is no assumption about the distribution of intervals between arrivals and service duration. The basic model is thus the G/G/1 model with one server, unlimited FCFS queue, unlimited population and state independent random distributions. Fig. 1 shows two versions of its PetriSim model. First net is made of a standard generator of tokens = customers, a place = queue where the customers wait and a transition that represents the service. The only difference between the first net in fig. 1 and the basic Place/Transition net is the firing delay generated by the code snippets. PetriSim procedure *GG1Report* writes the results of the G/G/1 station to the text file *R* (it can be the screen). The text in bold is literally everything that the user writes to create a G/G/1 simulator. The second net shows the use of PetriSim icons. Place and transition type is selected by the user interactively together with other attributes like size, color, etc. See also the use of the testing arc. For transitions with no firing delay a testing arc is equivalent to a pair of arcs with the same endpoints and multiplicity but opposite directions. For delayed transitions there is a difference: a testing arc does not remove tokens from the input place, it just tests their presence. If a pair of arcs is used, the token(s) are removed when the firing starts and returned back at its completion. This may obviously result in a different net behavior.



```
Procedure _PS1; {Starting snippet of Arrival}
Begin
  FireDelay(UniformR(2,8)); { 1/lambda = 5 }
End;

Procedure _PS2; {Starting snippet of Service}
Begin
{ 1/mu = 4, only values from [1,7] accepted }
  FireDelay(Normal3Sig(4,1));
End;

GG1Report(R,'Queue','Service'); { output to R: }
Report on G/G/1 made of Queue & Service
Queue length
Average: 0.218 Min: 0.0 Max:  5.00 StdDev: 0.475
Queue waiting time
Average: 1.09  Min: 0.0 Max: 19.42 StdDev: 1.777
Server utilization: 0.80
Service duration
Average: 4.006 Min:1.0024 Max:6.99 StdDev: 0.989
Average system size:     1.017
Average system wait:     5.096
Effective arrival rate: 0.1997
```

Figure 1: PetriSim G/G/1 Model

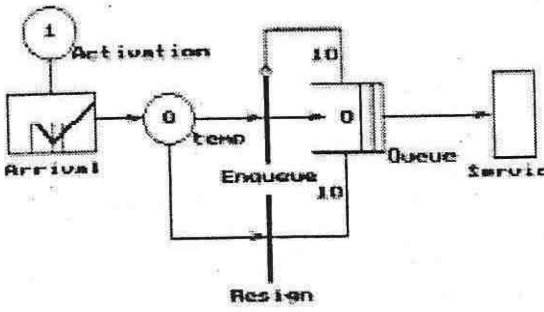
Like in Queueing theory for M/M/1, the basic G/G/1 model can be generalized by relaxing some of its rather limiting assumptions.

Multichannel Models

A place = queue can be an input place of any number of delayed transitions = servers with generally different service patterns. See the two servers of the limited population model in fig. 3. PetriSim (so far) does not support priorities of transitions. If there are more enabled transitions, one of them is selected randomly (equal probabilities). For a multichannel that means that the load is distributed equally.

Limited Capacity Models

PetriSim does not limit the number of tokens in a place (it is implemented as a long integer with maximum value close to 2×10^9). Fig. 2 shows a solution by using inhibitor and testing arcs. The transition *Enqueue* can fire if there is a token in the place *temp* (arriving customer) and less than 10 tokens in the queue. System capacity is thus limited to 11. Rejected customers are removed (and counted) by the transition *Resign* that needs 10 tokens in the queue to fire. Fig. 2 also shows standard results supplied by PetriSim with some rounding and less important figures removed to fit in. The user code of the model in fig. 2 thus contains just the snippets similar to those in fig. 1 (actually the only difference is uniform input in [2,6]).

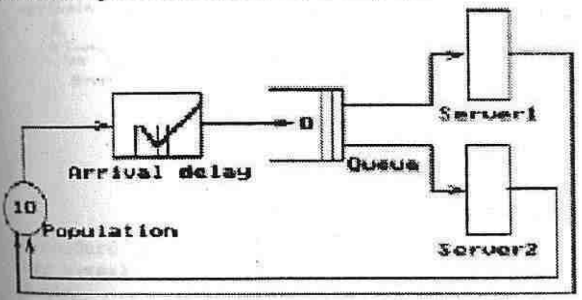


```
Report on the net   : ESMEx3 Model: G/G/1/K model
Experiment duration:    100000.000
Current time        :   100000.365

Places:
Name    Mean Min Max StD AvWait   Max  StD  Added
==================================================
Queue   4.865  0  10 2.91 19.63 48.88 11.66 24787
Activ.  1.000  1   1 0.00  0.00  0.00  0.00     0
temp    0.000  0   1 0.00  0.00  0.00  0.00 24976

Transitions:
Name    Util. MeanFire  Min  Max  StDev  Firings
=================================================
Arrival 1.00    4.004 2.00 6.00  1.154    24976
Service 0.99    4.004 1.01 7.00  0.991    24777
Resign  0.00    0.000 0.00 0.00  0.000      189
Enqueue 0.00    0.000 0.00 0.00  0.000    24787
```

Figure 2: PetriSim Limited Capacity Model

Limited Population

Limited population models drop the assumption that the arrival pattern is independent of the number of customers in the system. A typical application is maintenance of a small number of machines that break down randomly. First it is necessary to define the mapping between the system size and the arrival rate. The mathematical model of limited population assumes that the intervals between arrivals are equally exponentially distributed for all customers with the mean interval equal to $1/\lambda$. Let M be the total number of customers and let n be the current system size (number of customers in). The arrival rate is then equal to $(M-n)\lambda$. The model in fig. 3 is based on this assumption. The place *Population* contains the customers that are out. The delayed transition *Arrival delay* generates state dependent intervals between arrivals - see the code snippet. The local variable *mean* is used to improve readability. Starting snippets are activated after removing tokens from input places. That's why 1 has to be added to the marking of the place *Population* obtained by the PetriSim function *MarkingOf*. The arrival rate is 0.1 for each of the 10 customers, so the average interval (for example mean time between failures) is 10 time units. If all are out, the arrival rate to the system is 1. The two servers have both normal service with mean 8, standard deviation 2, limited to the interval [2,14]. The results show that the load is distributed equally between the two servers and both are fully utilized. Note that the average number of customers out of the system (for example average number of operational machines) is only 2.8.



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

```
Report on the net   : ESMEx4 Model: G/G/1/inf/M
Experiment duration:   100000.000
Current time        :   100000.360

Places:
Name   Mean Min Max StD AvWait   Max  StD  Added
================================================
Queue  4.213  0   8 1.67 17.04 40.51 6.64  24727
Popul. 2.826  0  10 1.76 11.43 85.85 7.37  24719

Transitions:
Name    Util. MeanFire  Min   Max  StDev  Firings
================================================
Arr.d.  0.99   3.998 0.00 86.23  5.266    24727
Server1 0.98   7.972 2.02 13.93  1.974    12379
Server2 0.99   7.989 2.04 13.91  1.971    12340
```

Figure 3: PetriSim Limited Population Model

PetriSim can of course easily express a more complex mapping between the system size and the arrival rate. The problem would be its mathematical specification because of the joining of non-Poisson input processes. Alternatively we can generate intervals between arrivals for each customer separately using any (for example empirical) distribution. A Petri net model based on this idea would need a delayed transition for each customer that is feasible only for small number of customers.

Bulk Input and Bulk Service

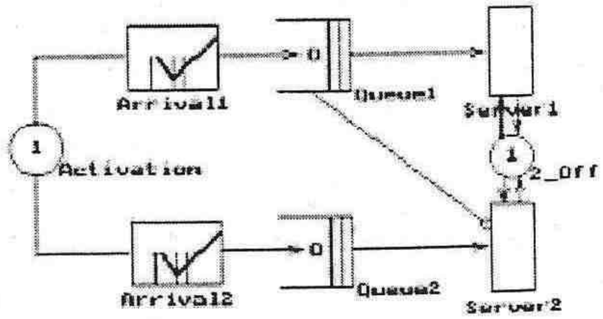
Petri nets can easily model an input of batches of a constant size by increasing the multiplicity of arcs. The situation is more complicated for random batches where the batch size is a discrete random variable. In PetriSim such models can be created, although the method is not elegant and potentially dangerous. The PetriSim procedure *ChangeMarking* can be used to modify marking of any place at any time. So for example the ending snippet of the transition Arrival of the net in fig. 1 could be as follows:

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

The procedure *ChangeMarking* should not be used unless it is really necessary. If used, the user has to turn off the collection of tokens waiting time that is implemented by lists made of records with entry times. Similarly it is possible to model a bulk service of random batches or a service of batches that are smaller than the standard size if less tokens are available in the queue.

Classes and Priorities

In PetriSim it is not possible to assign priorities to tokens or transitions. Fig. 4 shows a solution based on a semaphore place and an inhibitor arc. There are high priority customers generated by *Arrival1* and low priority customers generated by *Arrival2*. A single server is modeled by two transitions. Place *2_Off* is a semaphore that makes the firing of the two servers mutually exclusive. Low priority *Server2* can fire only if the high priority *Queue1* is empty (inhibitor arc) and *Server1* is idle. Note that there is no preemption, the high priority *Server1* waits until the low priority service is completed.



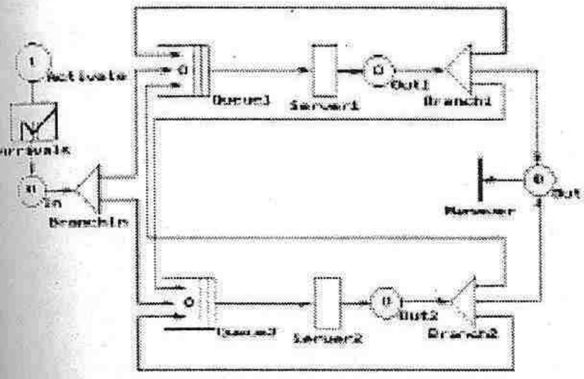
```
Places:
Name    Mean Min Max StD AvWait    Max  StD  Added
==================================================
Queue1 0.224   0   2 0.42  2.23    7.39  1.67 10041
Queue2 1.448   0  14 1.79 14.53  142.01 17.24  9968
```

Figure 4: PetriSim Model with Priorities

The results in fig. 4 were generated with uniform arrivals in [5,15] both and a normal service with mean 4.8, standard deviation 1 for both priorities. A different behavior of the two queues is evident. Note that in the model in fig.4 the two priorities can have different arrival and service patterns. The model can also be used for two classes of customers without priorities. After removing the inhibitor arc the customers from the two queues will be served randomly without any precedence.

Networks

The elementary constructs described so far can be combined into networks in an obvious way. So far customers have vanished in servers. An output arc can be used to move served customers into another queue, etc. One problem remains to be solved: random movement of customers. In PetriSim the problem is solved by the so-called branching transitions displayed as triangles – see the net in the fig. 5.



```
Procedure _PE3; {Ending snippet of BranchIn}
var x:real;
Begin
   x := Random;
   If x<0.6 then t^.Branch := 1     {60% up}
   Else          t^.Branch := 2   {40% down}
End;
```

Figure 5: Network Model in PetriSim

There is no difference between branching and other transitions with respect to input arcs. They can also introduce a delay. The difference is that after firing only one output arc is activated. The activated arc is supposed to be selected by the ending snippet like the one in fig. 5. If there is no user selection, the arc is selected by PetriSim randomly (all outputs equally likely). So after firing the transition *BranchIn* the customer enters either *Queue1* (probability 0.6) or *Queue2* (probability 0.4). Similarly after the service the customers either leave the system or join again one of the two queues. To check the results, exponential arrival intervals and service duration were used. Results obtained from PetriSim simulation were very close to the figures computed for the Jackson network in fig. 5. The model in fig. 5 was also used to compare the speed of the PetriSim simulator with other simulation tools. The same network model created in Extend^{TM} is shown in fig. 6 where most icons are self-explaining. The bottom right block is a plotter showing lengths of queues.

Another model has been created by Arena^{TM} and by the JavaScript network simulator (Sklenar 2001) available at: *http://staff.um.edu.mt/jskl1/simweb/ggs5/ggs5main.html.* Numerically all the results were very similar and very close to the theoretical figures.

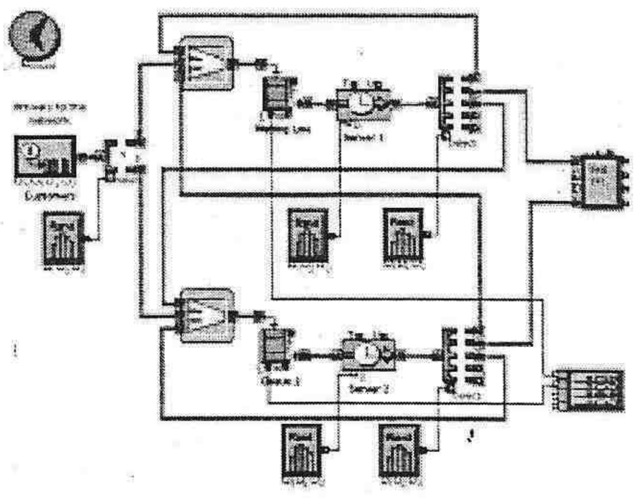


Figure 6: Network Model in Extend^{TM}

Table 1 shows the duration of an experiment of the length 200,000min. The average interval was 10min, average service duration 4min and 3min respectively, all times were exponential. Branching probabilities after service (top down) were these: *Branch1* (0.3, 0.2, 0.5), *Branch2* (0.4, 0.4. 0.2). These resulted in the average queue length 1.75 and the average wait 9.8min for *Queue1* and 0.44 and 2.7min for *Queue2*.

Table 1: Speed of Queueing networks Simulators

Simulator	Arena^{TM}	Extend^{TM}	JavaScript	PetriSim
Time [s]	5	10	25 (IE 5)	13

Conclusion

PetriSim as an educational tool can help to teach both discrete simulation as such together with principles of modeling using Petri nets. It is a free tool available at: *http://staff.um.edu.mt/jskl1/petrisim/.*

References

Bause, F. 1993 "Queueing Petri Nets - a formalism for the combined qualitative and quantitative analysis of systems". In: *PNPM'93*, IEEE Press,14-23.

Chao, X. et al. 1999 *Queueing Networks, Customers, Signals and Product Form Solutions.* John Wiley.

Jensen, K. (1998) "An Introduction to the Practical Use of Coloured Petri Nets". In: Reisig, W., Rozenberg, G. (eds.): *Lectures on Petri Nets II: Applications,* Lecture Notes in Computer Science vol. 1492, Springer-Verlag, 237-292.

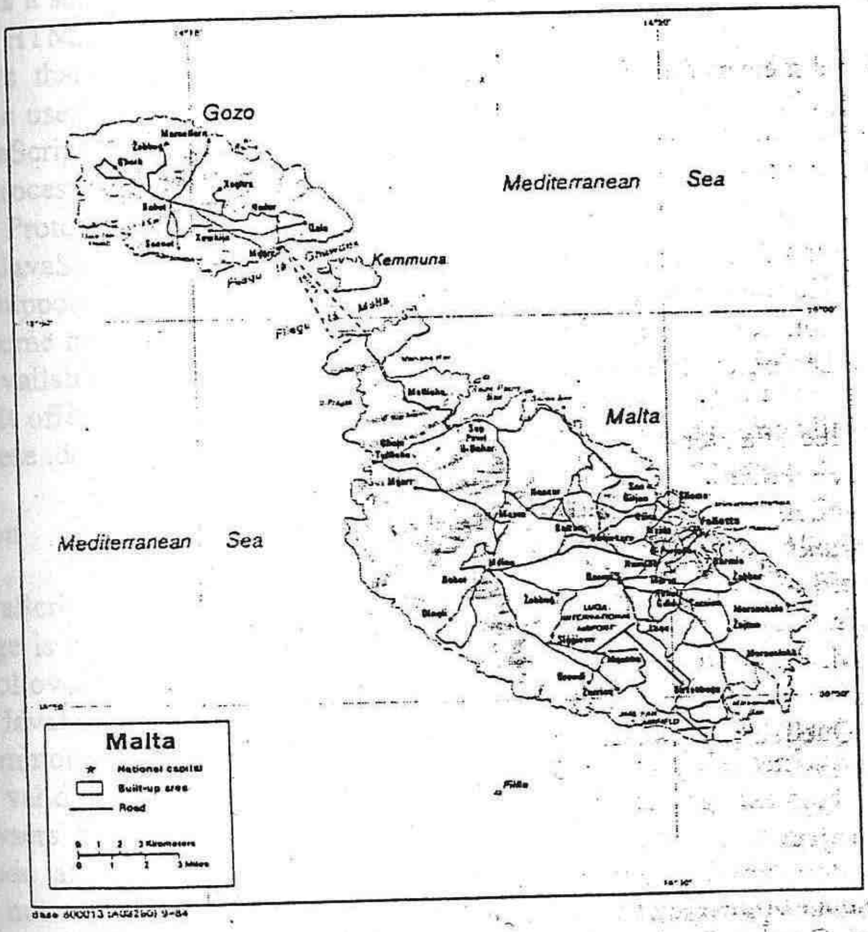
Sklenar, J. (2001) "Interactive Simulators in JavaScript". In: Proceedings of *15th European Simulation Multiconference ESM2001*, Prague, 247-254.

PROCEEDINGS

of

The 26th ASU CONFERENCE

OBJECT ORIENTED MODELING AND SIMULATION



Edited by Jaroslav Sklenar, University of Malta

Organized by ASU (Association of Simula Users)
and
University of Malta, Faculty of Science
Department of Statistics and Operations Research

September 26-28, 2000

Hotel Golden Sands, Malta

Object Oriented Programming in JavaScript

Jaroslav Sklenar
Department of Statistics and Operations Research
University of Malta, Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jskl1@stator.um.edu.mt

Abstract

JavaScript [1] is a scripting language intended to create dynamic interactive web documents. Together with HTML [2] JavaScript supports easy creation of documents with input and output elements that enable user-friendly entering of validated data that can be further processed in the user's computer and then sent to the server or displayed by the browser. So the user of JavaScript and HTML has a tool that contains ready-made parts to design the user interface. To process data a high level interpreted language is available. The paper explores the JavaScript Prototype oriented paradigm from the Object Oriented Programming (OOP) point of view. JavaScript is not a classical compiled strongly typed Object Oriented Language (OOL), but it supports use of most important techniques typical for OOP. Moreover its loose typing offers some new possibilities like dynamic updating and/or addition of properties and methods not available in classical OOLs. Inheritance programmed explicitly by the user is very flexible. It offers for example *Partial Inheritance* of selected methods only. The paper will present these ideas on concrete examples.

1. Introduction

The name JavaScript (used in this paper) is owned by Netscape, Microsoft's implementation of the language is called JScript. Fortunately both are more or less compatible and both are supposed to follow the European Computer Manufacturers Association standard ECMA-262 (ISO-10262). JavaScript features described in this paper are based on behavior of interpreters of the two common browsers: Communicator 4 or higher and Internet Explorer 4 or higher. Most are also valid for older versions and other browsers (like HotJava). Anyway, differences between browsers exist, so it is suggested to check the HTML documents and associated JavaScript code at least in these two browsers. In spite of the beginning of its name, JavaScript is not a version or modification of Java [3]. The only common thing is C/C++ based syntax, so it is better to consider Java and JavaScript as two different languages. In [4] there is a description of basic ideas of the JavaScript Prototype oriented paradigm that can be viewed as an alternative to classical Class oriented paradigm of compiled OOLs in interpreted environment that prevents early (compile time) binding and strong typing as such. Next chapters give more details and suggest how to use prototypes to implement techniques of classical OOP together with new techniques enabled by loose JavaScript typing.

2. Everything is an object

Some authors would not call JavaScript an OOL, but in fact a JavaScript programmer works only with objects and their methods and properties. If not stated explicitly, properties mean value attributes, but in JavaScript there is actually very little difference between value attributes – *properties* and procedure attributes - *methods*. Due to loose typing a property can become a method and vice versa. Most JavaScript "programs" are short snippets incorporated into HTML documents by using the tag <SCRIPT> that can also reference a separate file with JavaScript code (default extension `.js`). JavaScript code is processed by the browser's interpreter as a part of the (re)loading process. Such code typically defines functions and declares and initializes global variables. Using OOP terminology, such code is interpreted in the context of the so-called **global object**. For each browser window or each frame within a window a separate global object is created. So for example a global variable declared by the following statement:

```
    var expduration = 1000;  //default experiment duration
```

in fact creates (declares) a property of the global object that can be called directly by its name, but also as `this.expduration` or `window.expduration`. The keyword `this` is a reference (handle) of the current object. Similarly each global function declared by the user is a method of the particular global object. All standard properties and functions are also pre-defined properties of the global objects (for example mathematical functions and constants are accessible through the `Math` object, that is an automatically created property of the global object). What about the statements written directly in <SCRIPT> tags? Here the best analogy is the "life" as defined for Simula class instances. There can be a lot of such code in several <SCRIPT> tags and in `js` files. All this code is performed in lexical order as the life of the global object (obviously without any possibility to break and postpone its execution). Interpreted JavaScript has only run-time, compile-time does not exist. Still there are two stages: (re)loading is a sort of early or first run-time stage. During this stage the programmer creates all global data that includes "declaration" of prototypes and creation of global object instances. Second run-time stage is made of execution of functions activated by browser events (like for example pressing a button). Even now, all is done in the context of the so-called **call object**. Each activation of a function creates a separate call object, so declaration of a local variable (say `var x = 0;`) in fact creates a property of the call object. Actual parameters of the function call are also considered as properties of the call object (there is another array property `arguments` that contains all parameters passed to the function, so in JavaScript it is possible to write functions accepting variable number of parameters - see later).

3. Every function can be a constructor

In JavaScript object instances are created by the operator `new` similarly as in practically all OOLs:

```
    queue = new FifoQueue("q1",0);   // empty queue
```

The difference between JavaScript and compiled OOLs is that there is no class declaration. So the constructor has a much more important role. Unlike constructors of compiled OOLs that basically just initialize the instance, JavaScript constructor actually creates (constructs) an instance by creating its properties and possibly methods. There is no special keyword to write constructors, so each function can potentially be a constructor. We already know that for each function call JavaScript creates a call object. It in fact creates another initially empty object (here we shall call it **created object**) that can be in the function body referenced by the keyword `this`. Functions that are not intended to be constructors ignore it. Note that call object and created object are really two distinct objects. For example consider the following statements anywhere in a function body:

```
    var x = 1;
    this.x = 2;
    alert(x);
```

The value displayed will be 1 (property of the call object), `alert(this.x)` would display 2 (property of the created object). A constructor function is not supposed to return any value, but it can. If it returns an object, the standard created object is ignored and the returned object will be used instead. But there is more: in JavaScript each function is in fact represented by a function object created during parsing a function. With respect to possible use of the function as a constructor, let's call this object **constructor object** or simply **constructor**. Unlike the other two objects associated with a function (call and created objects) that exist only temporarily during the function execution, constructor object is permanent (it is removed by loading another document). Because it is an object, it can have properties and methods. The most important one is *prototype* – see the next chapter. Using OOP terminology each constructor (and so each function) has - or can have - the same role as class declaration in compiled OOLs. Whether a particular function really represents a class depends only on its body (constructors are supposed to create properties by statements like the above one: `this.x = 2;`) and especially on its use after the `new` operator. Note that constructors can create value properties and methods in unified way (example taken from [4]):

```
    function point(xcoor, ycoor) {
        this.x = xcoor;
        this.y = ycoor;
        this.shift = function(dx,dy){this.x+=dx; this.y+= dy};
    }
```

Note that all instances created by the above constructor will have its own code of the method `shift`. The obvious requirement is a code stored only once for all instances. In JavaScript this is achieved by prototypes.

4. Prototypes

Prototype is the property of each function (constructor), so there is one prototype for every class (provided the function is used as a constructor). JavaScript enables objects (instances) to access prototype properties and methods in the same way as its own private properties and methods: `o.m()` activates the method `m` of the object `o`. It can be either private local method of the instance or a prototype method common to all instances of the class `o`. Sharing prototype methods by all instances is no problem, but what about value properties? JavaScript avoids clashes and side-effects by making prototype value properties read-only. They are created and initialized at parse time, instances can only read their values. They can be useful as constants common to all instances. Creating prototype properties (not typical) and prototype methods thus completes the class definition. Let's summarize: each class is represented by two permanent objects: constructor and prototype (constructor's property) and then there can be any number of instances that can have their own properties and methods and that can access properties and methods of the two permanent objects. Using the prototype paradigm the class *point* can be implemented in this way:

```
function point(xcoor,ycoor) {
   this.x = xcoor;
   this.y = ycoor;
}
point.prototype.shift = function(dx,dy) {
   this.x+=dx; this.y+=dy
}
```

Now the code of the method shift is stored only once with obvious consequence - after its modification all instances will use the modified version.

5. Standard OOP techniques

The two previous chapters give a hint how to create the usual four types of properties and methods associated with a certain class, see also [1]:

Instance value properties are created by the constructor (or by a function called by the constructor – see later) using the handle `this`.

Instance methods can be created either directly by the constructor or preferably by the prototype – see the two above examples.

Class value properties that are associated with the class as such, not with the individual instances can be created as properties of the constructor:

```
point.count = 0; // number of created points
```

Class constants can be properties of either the constructor:

```
point.maxX = 800; // maximum x co-ordinate
```

or the prototype which ensures the read-only access:

```
point.prototype.maxX = 800; // maximum x co-ordinate
```

Class **methods** associated with the class as such, not with the individual instances can be created as methods of the constructor:

```
point.distance = function(p1,p2) {
      var dx = p1.x - p2.x;
      var dy = p1.y - p2.y;
      return Math.sqrt(dx*dx + dy*dy)
}
```

Note that the above declaration is logical – distance applies to two points, so it is more natural to declare it as a class method and not as an instance method.

5.1 Inheritance

Let's consider only instance value properties and instance methods and let's create a subclass `colorpoint` with one more property `color` and one more method `changecolor`. Unfortunately we have to consider separately inheritance of value properties and inheritance of methods.

Regarding **value properties** it is possible to repeat creation of superclass properties in the subclass constructor. There are two reasons against this approach: first there is redundant code and what's more important, any modification in the superclass must be repeated in all subclasses with obvious danger of inconsistency. The solution is a function used both by the superclass and the subclass:

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

Inheritance outlined in the previous example can continue – colorpoint can be used as superclass etc. Note that in order to allow inheritance, constructors just call functions that create the properties. Of course private methods can be inherited in the same way as value properties.

Methods stored in prototypes can be inherited either by replacing the whole prototype object or by copying the methods. The first method can be used in our example in this way:

```
colorpoint.prototype = new point(0,0);
colorpoint.prototype.changecolor = function(c) {
   this.color = c
};
```

The first statement replaces the standard prototype by a point instance, the second statement creates the new method. There are two flaws – first the new prototype has also the two value properties (`x` and `y`). It is not a mistake. Assuming that the constructor creates properties with the same names, prototype read-only properties are not visible and the only problem is wasted memory. The second flaw is more serious. All prototypes have a property `constructor` that is a reference to the function object of the constructor function that has created this object. It can be very useful to check types of instances (like "is" in Simula). Obviously by replacing the standard prototype by another object as in the above example, the `constructor` property also changes. To get exactly what we want, methods have to be copied from superclass to subclass. The following function uses the fact, that objects in JavaScript can be treated as associative arrays and a special form of the "for" statement based on this fact:

```
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
};
```

The first statement copies all prototype methods, the second statement creates the new method.

5.2 Polymorphism

Polymorphism is basically not compatible with strong typing. That's why classical compiled OOLs introduce late binding through virtual methods that represents probably the most complicated part of OOP. Note that Java does not have virtual methods, but it means that all methods are in fact virtual. Programmer's comfort is paid by time. Interpreted loosely typed JavaScript is polymorphic by nature. Moreover polymorphism is not limited to inheritance sequences. Consider the next two statements: `p = x; p.m();` They will work for any object `x` that has a method `m`. JavaScript does not perform any type check, everything is the programmer's responsibility. So the flexibility is paid by very limited security. That's why JavaScript can not compete with classical OOLs in large projects.

6. Other techniques

Interpreted JavaScript enables techniques not conceivable in compiled strongly typed OOLs. Among others the following is possible:

Adding instance properties and methods can be a fast alternative to inheritance, especially in case of small number of instances. At any time it is possible to create new properties and methods:

```
origin = new point(0,0);
p = new point(ix,iy);
p.d = point.distance(origin,p); //New property of p
p.dupdate = function() {        //New method of p
   this.d = point.distance(origin,this)
};
```

Modification of instance properties and methods is also possible. It can be used for example as an alternative to status variables. Instead of testing a status, activate directly the method that has been updated accordingly. It is also possible to change a method into a property and vice versa, but then they have to be treated accordingly.

Partial inheritance is a technique not allowed in OOLs. Nevertheless there are situations when a simplified version of a certain class is required. I have used this technique in the simulator of queueing networks to create a *discrete random distribution* object as a simplified version of *general random distribution* object that has already been available. The point is inheriting only some methods (obviously they must be self-sufficient). The following example shows another version of the function `inherit`. It first tests the number of actual parameters passed to it. If there are two parameters it copies all methods as in the above example. Otherwise it assumes that after the first two object parameters there is a list of strings – names of methods to be copied:

```
function inherit(from,to) {
  if (arguments.length == 2) { // Inherit all
    for(var p in from.prototype)
     if (typeof from.prototype[p] == "function")
       to.prototype[p] = from.prototype[p];
  } else {                      // Inherit the listed methods
    for (var i=2; i<arguments.length; i++)
     to.prototype[arguments[i]] = from.prototype[arguments[i]];
  };
};

inherit(Distribution,DiscreteDistribution,
        'fixtable','last','compareWithLastValue','showTable',
        'accept','edit','insert','deleteit','cleartable',
        'confirmtable','generate','save','load');
```

Only the 13 listed methods are inherited.

Conclusion

JavaScript programmers can use all fundamental techniques of OOP together with new techniques offered by loose JavaScript typing. The appendix contains an example of multiple inheritance – compare with similar example in [4].

References

[1] Flanagan, D.: *JavaScript - The Definitive Guide.* O'Reilly & Associates, Inc., 1998.

[2] Darnell, R. et al.: *HTML 4 Unleashed. Professional Reference Edition.* Sams.net Publishing, 1998.

[3] Eckel, B.: *Thinking in Java.* Prentice Hall. Inc., 1998.

[4] Sklenar, J.: *Prototype Oriented Simulation in JavaScript.* Proceedings of the 25th Conference of the ASU: System Modelling Using Object Oriented Simulation and Analysis, Balatonfured 1999, pp 61 – 71.

Appendix: Multiple inheritance example

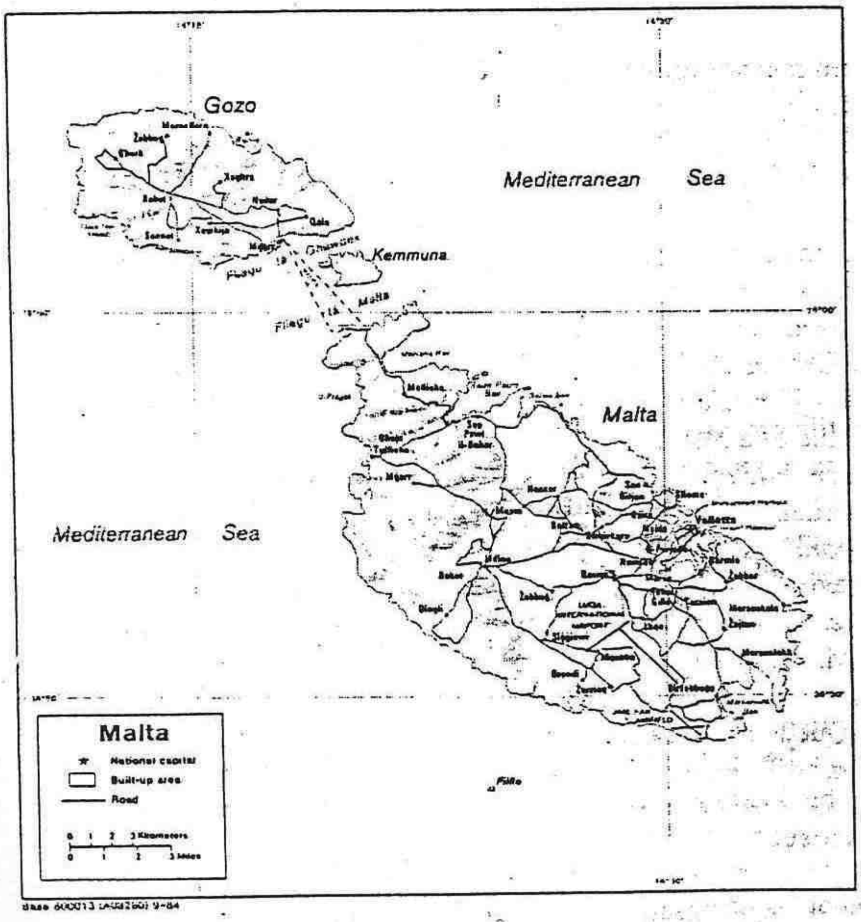
```
//=== FIFO Queue (Inherits from GenQueue and StatQueue) ===

function FifoQueueProperties(name,size,obj)
{  GenQueueProperties(name,size,obj);
   StatQueueProperties(obj);
   obj.qhead = 0;          // points to the first item
   obj.qtail = 0;          // points to first free slot
}

function FifoQueue(name,size) {          // Constructor
   FifoQueueProperties(name,size,this);
}

// "Inherited" methods:
inherit(GenQueue,FifoQueue);
inherit(StatQueue,FifoQueue);

// New FIFO Queue methods:
FifoQueue.prototype.initiate = function (size,time)
{ with (this) {
   setsize(size);
   settime(time);
   qhead = 0;
   qtail = 0;
}};

FifoQueue.prototype.enqueue = function (x) { with (this) {
  if (full()) {
   return false;
  } else {
   qlength++;
   qarray[qtail++] = x;
   if (qtail == qsize) { qtail = 0 };     // wrap around
   if (qlength > maxqlength) { maxqlength = qlength };
   return true;
  }
}};

FifoQueue.prototype.removefirst = function () { with (this) {
  if (empty()) {
   return null;
  } else {
   qlength--; var x = qarray[qhead++];    // 1st item
   if (qhead == qsize) { qhead = 0 };     // wrap around
   return x;
  }
}};
```

PROCEEDINGS

of

The 26th ASU CONFERENCE

OBJECT ORIENTED MODELING AND SIMULATION



Edited by Jaroslav Sklenar, University of Malta

Organized by ASU (Association of Simula Users)
and
University of Malta, Faculty of Science
Department of Statistics and Operations Research

September 26-28, 2000

Hotel Golden Sands, Malta

Simulator of Queueing Networks

Jaroslav Sklenar
Department of Statistics and Operations Research
University of Malta, Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jskl1@stator.um.edu.mt

bstract

avaScript [1] and HTML [2] are tools used for creation of dynamic interactive web ocuments. Among others they support programming of forms - documents with input and utput elements that enable user-friendly entering of validated data that can be further rocessed in the user's computer and then sent to the server or displayed by the browser. So avaScript and HTML contain ready-made parts to design easily the user interface. To process ata a high level interpreted language is available. Paper [3] explores the JavaScript prototype riented paradigm from the Object Oriented Programming (OOP) point of view. To explore avaScript capabilities to create medium-size software tools, a simulator of queueing networks as been implemented using exclusively JavaScript and HTML. The paper outlines the design f the simulator from the OOP perspective. The results show clearly that JavaScript nterpreters of the two commonly used browsers (Communicator 4.7 and Internet Explorer .0) are fast enough to enable creation of web hosted non-trivial simulation models.

1. Introduction

The paper [3] explains the basic ideas of the JavaScript prototype oriented paradigm. It can be iewed as an alternative to classical class oriented paradigm of compiled Object Oriented anguages (OOL) in interpreted environment that prevents early (compile time) binding and trong typing as such. The paper [3] also suggests how to use JavaScript prototypes to mplement all important techniques of classical OOP together with new techniques enabled by oose JavaScript typing. So even if terminology in JavaScript is sometimes different (for xample the keyword *class* does not exist), in the next chapters we shall use the standard OOP erminology because the OOP functionality can be implemented in JavaScript almost entirely.

2. Why queueing networks?

ueueing networks are maybe the most frequently simulated systems. There are results for athematical analysis of queueing networks – see for example the monograph [4], but still ther limited. Many even simple practical situations can be described by queueing networks r which the analytical solution is not known. This is true especially for networks with other an exponential service times – in other words all practical cases. To create simulation odels of queueing networks there are basically two feasible methods: writing a tailor-made mulation model in a discrete simulation language or similar tool (Simscript, class Simulation f Simula, etc.) or creating a model by using a visual interactive simulation environment Arena, Extend). The project that is being described in this paper belongs to the second ategory. It was intended to create an easy to use interactive general simulation tool pporting simulation of networks with varying size and topology. The tool was created for e web platform, so it is available literally to everyone connected to Internet. The only requirement is a browser with an interpreter supporting JavaScript 1.2 (both common browsers do). The first version of the simulator is rather limited, so it should be taken more as an attempt to assess JavaScript abilities in this area than as a professional simulation tool. Nevertheless it enables simulation of all networks that satisfy the following restrictions:

- There is only one class of customers. After generation a customer can randomly enter any service station. After each service a customer can randomly enter another service station or leave the network.
- Unlimited population is supposed (random intervals between arrivals do not depend on the number of customers in the network).
- Simulation starts in time 0 by scheduling first arrivals for all generators. Initially all service stations are empty.
- Service stations can have limited capacity. Customers entering a full station are lost (they leave the network).
- Service stations are made of a possibly limited queue and any number of parallel identical channels. Service duration is random, state independent. Queue discipline is either FIFO or LIFO. Load among servers is distributed evenly (no results are available for individual servers).
- Routing of customers is represented by random destination that exists for each generator and for each service station. Destination is the service station number or 0 for the environment respectively.
- Simulation experiment either takes a certain time or it can be terminated by some number of generated and/or served (departed) customers. Out of these three conditions the one that comes first is applied.

The simulator gives all typical simulation results. Tables of user defined random distribution can be used for both intervals between arrivals and service times. Next chapters give more details.

3. Outline of the solution

After some simplification and generalization we can identify two types of objects in queueing networks: *generators of customers* and *service stations*. There can be practically any number of both. The implementation platform (HTML & JavaScript) does not directly support graphics. That's why it has been decided not to visualize the network topology automatically. For networks where each generator can pass customers to any service station and where served customers can move to any further service the diagram is not very useful anyway. There can be a lot of data for each generator and each service station. So the simulator always shows one generator, one service station, and global simulation results. For the selected generator and service station the simulator displays the two distributions (interval between arrivals or service duration and random departure destination), parameters, and simulation results. Optional report in a separate browser window contains summary of all results. The main browser window is divided into the Control frame and the Network frame. The former contains the elements used to enter global data (number of generators and number of servers) and to control simulation (experiment duration etc.). There are also selection lists, so it is very easy to switch among generators and service stations. The network frame contains all the data and the results of the selected objects together with network results.

4. Network objects

This chapter lists the classes used to implement the network behavior from the user's point of view. Some classes declare attributes that are instances of other classes - for example every *Generator* has a property *interval* that is an instance of the class *Distribution*. As the name suggests it supports entering, editing and generation of random intervals between arrivals into the network from this generator. This technique called **composition** (creating classes from other classes) is very common in OOP and very natural in JavaScript. **Inheritance** is also used, but it has to be programmed - see more in [3]. JavaScript does not support nesting of classes in Simula sense.

Appendix 1 is the complete commented code that declares the class ***Generator***. During simulation, instances of this class will generate customers with random intervals entering randomly various service stations. Note that "class declaration" in JavaScript has two parts: the so-called constructor function (whose name is the name=type of the class) creates value attributes. It can also create methods, but these methods would be repeated in each instance. That's why methods are usually implemented as properties of the so-called prototype, that is automatically created for each constructor function. Note how the constructor creates the two object properties: *interval* and *output* together with initialization of their properties. The first one is the random interval between arrivals. It can be either a theoretical distribution - the default initial value is the exponential distribution with mean 1, or a user defined distribution given by an empirical table. The other distribution is a discrete user-defined distribution that defines random distribution of entering customers among network service stations. This distribution is initialized in such a way that the table contains one entry: value 1 with probability 1 (by default all customers proceed to the service station 1). The class *DiscreteDistribution* has been declared as a simplification of the general class *Distribution* by inheriting only some methods. Programmed JavaScript inheritance allows this - see the outline in [3]. Other value properties of the class *Generator* are the generator number (note how the number is used to create unique names of the two distributions) and statistical parameters - see comments in Appendix 1. The class *Generator* has three methods: the method *initialize()* tests whether user defined empirical tables (if used) have been confirmed. If not, simulation can not start. The rest is initialization of statistics. The method *nextarrival()* schedules the arrival of the next customer. Note how the method *generate()* of the object property *interval* is used to generate the length of the interval. The method then creates an event notice and schedules the arrival event - see chapter 5. The rest is updating statistics. The last method *garrival()* is the event routine activated by the simulation engine. It generates and checks the number of the service station. If it is correct, statistics is updated, a customer is created and passed to the station - the actual arrival to the station is implemented by the method *sarrival(cust)* of the class *Server*. Then the method schedules the next arrival.

Another class ***Server*** implements the service stations. Its constructor creates two distribution object properties (service duration and random output) and a number of statistical parameters. The methods are these: *initialize()* - server initialization, *sarrival(cust)* - arrival of a customer, and *endofservice(cust)* - finishing the service of a customer. The code is too long to be presented here. In the method *initialize()* there are among others these statements:

```
if (queueorg == "FIFO") {  // preparing queue/stack
    queue = new FifoQueue("queue"+name,0);
} else {
    queue = new LifoQueue("stack"+name,0);
};
queue.initiate(x,0);        // x=queue size, time=0
```

Note the polymorphic behavior, that is very natural in loosely typed interpreted languages like JavaScript. The last statement would be correct for all *queue* objects with a method *initiate* (number of parameters is not tested). Of course such flexibility is paid by very limited security.
The last network class is ***Customer***. So far it has only two value attributes: time of arrival to the network and time of arrival to a service station. It is used to compute the time spent in the network and the time spent in individual server stations.

5. Simulation engine

The simulator is based on classical event approach. There are in fact only two types of events: *customer arrival* and *end of service*. The simulation control is implemented by two classes. The instances of the class ***evnotice*** are event notices of the scheduled events. The class is declared by the following self-explaining code:

```
function evnotice(s,c) { // next arrival or end of service
  this.station = s;    // generator/station number
  this.cust = c;       // customer (null for generator)
  this.key = 0;        // event time (set by schedule)
};

evnotice.prototype.schedule = function(tim) {
  if (tim<time) tim=time; // cannot go back in time
  this.key = tim;         // event time
  SQS.insert(this);       // enqueue
};
```

This simplicity is enabled by the fact that the simulator is not a general discrete-event simulation tool by rather a special purpose simulator. The SQS is implemented as an instance of the class ***Heap***. In this context "heap" is a data structure, not a pool of dynamically allocated memory. Heap is a balanced (ideally shaped) binary tree where for each node - assuming ascending ordering - all its children have the key greater or equal than this node. So it is guaranteed that the first item is always the smallest one. Unlike in a sorted tree, heap does not keep any relationship between the siblings. The algorithms to insert a new item into a heap and to remove the first one are both very simple and both have the performance O($\log_2 n$) where n is the umber of items in the heap. So heap can be used as a host data structure to implement a priority queue - fast and simple algorithms for both typical operations. For a detailed description of heap algorithms see for example [5].
Appendix 2 is the main part of the simulation control function. It first calls the function *Initialization()* that initializes all global variables and all network objects. For example generators are initialized by the following code:

```
for (var i = 1; i <= numofgeners; i++) {
  if (!generators[i].initialize()) {
    return false;
  }; // Scheduling first arrivals:
  generators[i].nextarrival();
};
```

In the main loop the function *Finish()* tests whether the experiment is to be finished, SQS should never be empty. If not, the first event is removed from SQS and an appropriate generator or service station is activated. The last statement in the loop updates the status bar of the browser window.

6. Example simulation

Let's consider an abstraction of a certain commercial bank as outlined by the block diagram in Figure 1 (in [6] there is a Petri net model of the same system).

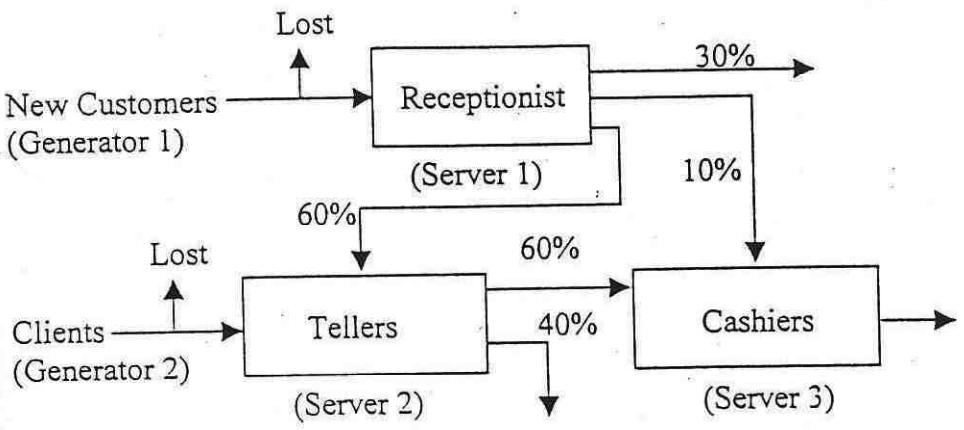


Figure 1: Queueing network abstraction of a bank operation.

The bank operates in this way: new customers wait in the receptionist's queue with the limited capacity 6, so they may be lost. They are then served by a receptionist and either leave the bank (30%), proceed to the tellers section queue, that also has limited capacity 10 (60%) or proceed directly to the queue of unlimited capacity served by cashiers (10%). Customers served by tellers either leave the bank (40%) or proceed to the cashiers section (60%). Clients enter directly the tellers queue (if it is not full). Random intervals between arrivals of the two types of customers are both exponential with the mean 15 minutes for new customers and 2 minutes for clients (busy period of the day). Random duration of all services are also known. It is assumed that the receptionist serves the new customers with uniformly distributed time from 5 to 20 minutes. Service durations of tellers and cashiers are given by empirical tables with 5 and 6 points respectively. The average duration is approximately 4 and 2.3 minutes. Simulation is supposed to find numbers of lost and served customers, lengths of queues, and utilization of servers for various numbers of tellers and cashiers. The purpose of the simulation study is to find such configuration that would minimize the number of lost customers and the lengths of queues while keeping reasonable utilization of employees.
The model described above has been created and tested in three different browsers using a typical desktop PC (PII 300MHz, 64MB RAM, W98). Always all tasks except the browser were closed. The experiment duration was measured in the following way:

```
start = new Date();
parent.frames[1].Simulation();
finish = new Date();
alert('Duration in ms: '+(finish.getTime()-start.getTime()));
```

The code is executed when pressing the *Run* button, so the time includes the experiment and its evaluation (report in a separate browser window was not generated). The results are summarized in the following table that gives experiment duration in seconds for various model times and browsers. The system configuration was one receptionist, three tellers and two cashiers that means 8 sources of events (two generators and together 6 service channels). Note how a simple output to the status bar affects the experiment duration.

Model time [min]:	1000	10000	20000
IE 5.0	3.02	31.25	64.32
IE 5.0 (no status)	1.15	10.82	21.86
NC 4.7	5.22	53.22	107.66
NC 4.7 (no status)	1.92	20.48	41.75
HotJava 3.0	49.2	-	-

HotJava port to PC has apparently problems with memory management, so the above table should not be taken as the browser's assessment as such. Internet Explorer 5.0 has clearly the fastest JavaScript interpreter (computation takes about 50% of the time needed by Communicator 4.7 - for the status bar turned off). Anyway both browsers are stable during simulation and the table shows that experiments with medium-size models are relatively very fast. Note that experiments with model time 10000 minutes that represents about one month of the bank's operation were completed in both browsers in less than one minute. Appendix 3 contains a part of a report taken from the browser's window (in this case IE 5.0) with results rounded to 3 decimal places.

Conclusion

JavaScript can be used as a simulation tool. It supports all important techniques of OOP and the interpreters are fast enough to perform experiments in acceptable time. JavaScript & HTML is thus an ideal tool to produce documents that contain data processing (including simulation) and that can be distributed to large number of users very fast using the web platform. This is the basic idea of the IFORS (International Federation of Operational Research Societies) initiative called tutORial project - for more details see **http://www.ifors.org/tutorial/** whose part is also the simulator described in this paper. It can be also reached directly at:
http://staff.um.edu.mt/jskl1/simweb/net1/netmain.html.

References

[1] Flanagan, D.: *JavaScript - The Definitive Guide.* O'Reilly & Associates, Inc., 1998.

[2] Darnell, R. et al.: *HTML 4 Unleashed. Professional Reference Edition.* Sams.net Publishing, 1998.

[3] Sklenar, J.: *Object Oriented Programming in JavaScript.* Proceedings of the 26th Conference of the ASU: Object Oriented Modelling and Simulation, Malta 2000.

[4] Chao, X., Miyazawa, M., Pinedo, M.: *Queueing Networks, Customers, Signals and Product Form Solutions.* John Wiley & Sons, 1999.

[5] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms*. The MIT Press, McGraw-Hill, 1997.

[6] Sklenar, J.: *Using Inheritance to Implement High Level Petri Nets*. Proceedings of the 24th Conference of the ASU: Object Oriented Modelling and Simulation, Salzau 1998.

Appendix 1: Declaration of the class *Generator*.

```
function Generator(num) {      // Prototype function
  this.number = num;
  this.interval = new Distribution("i"+num);
   this.interval.disttype = "Exp"; //Default Exponential, Mean=1
   this.interval.prompt1 = "Mean";
   this.interval.par1 = 1;
  this.output = new DiscreteDistribution("go"+num);
   this.output.xv[0] = 1;   // By default all to server 1
   this.output.px[0] = 1;   // (value 1 with probability 1)
   this.output.fx[0] = 1;
   this.output.size = 0;    // Last index (0 = 1 entry)
   this.output.status = "Table OK";
   this.output.confirmed = true;
  this.arrivals = 0;        // Number of generated customers
  this.intervals = 0;       // Number of generated intervals
  this.sumofint = 0;        // Sum of arrival intervals
  this.mininterval = 0;     // Minimum Interval
  this.maxinterval = 0;     // Maximum Interval };
//................... Generator methods ........................

Generator.prototype.initialize = function () { with (this) {
  if ((interval.disttype=="User")&&(!interval.confirmed)) {
   alert("Simulation can not start because the Distribution of "+
    "intervals table of generator "+number+" is not confirmed.");
   return false;
  };
  if (!output.confirmed) {
   alert("Simulation can not start because the Routing of "+
    "arrivals table of generator "+number+" is not confirmed.");
   return false;
  };
  intervals = 0;              // Number of generated intervals
  arrivals = 0;               // Number of generated customers
  sumofint = 0;               // Sum of arrival intervals
  mininterval = Number.POSITIVE_INFINITY;    // Minimum Interval
  maxinterval = 0;                           // Maximum Interval
  return true;
}};
```

```
Generator.prototype.nextarrival = function () { with (this) {
  intervals++;                  // Number of generated intervals
  var inter = interval.generate();              // Interval
  sumofint += inter;               // Sum of arrival intervals
  if (inter<mininterval) mininterval=inter;  // Minimum Interval
  if (inter>maxinterval) maxinterval=inter;  // Maximum Interval
  var ev = new evnotice(number,null);  // Creating event notice
  ev.schedule(time+inter);              // Scheduling arrival
}};

Generator.prototype.garrival = function () { with (this) {
  var out = Math.round(output.generate()); // Generating server #
  if ((out>0)&&(out<=numofservs)) {        // Existing server ?
    arrivals++;               // Number of generated customers
    netarrivals++;    // Total customers for the whole network
    var cust = new Customer(time);        // Creating customer
    servers[out].sarrival(cust);       // Moving to the server
  };
  nextarrival();               // Scheduling the next arrival
}};
```

Appendix 2: Simulation control

```
function Simulation() {     // Simulation run
  if (!Initialization()) {
    return false;          // Message shown during initialization
  };
  var event = null;
  var stat = 0;
  var cus = null;
  while ((!Finish())&&(!SQS.empty())) {
    event = SQS.getfirst(); // Next event notice
    time = event.key;       // Updating time
    stat = event.station;  // Generator/Station number
    cus = event.cust;
    if (cus==null) {       // Generator
      generators[stat].garrival();
    } else {            // Server
      servers[stat].endofservice(cus);
    };
    status = "Running - Time: " + Math.round(time) + " of " +
          runlength;
  };
     ⋮
};
```

Appendix 3: Part of a result report (tellers section of the bank example)

Server # 2

User defined service duration:

#	x	p(x)	F(x)
0	0	0	0
1	1	0.05	0.05
2	2	0.1	0.15
3	3	0.3	0.45
4	4	0.2	0.65
5	10	0.35	1

Routing of departures:

#	x	p(x)	F(x)
0	0	0.4	0.4
1	3	0.6	1

Number of channels : 3
Queue capacity=10, FIFO organization

Number of arrivals : 5420
Number of not waiting arrivals : 2518 (46.46%)
Number of lost customers : 3 (0.06%)
Number of services : 5415

Average service duration : 4.059
Minimum service duration : 0.006
Maximum service duration : 9.999

Average waiting time : 1.795
Average non zero waiting time : 3.353
Maximum waiting time : 18.252

Average time in server : 5.853
Minimum time in server : 0.007
Maximum time in server : 25.594

Average queue length : 0.972
Maximum queue length : 10
Utilization of server(s) : 0.733

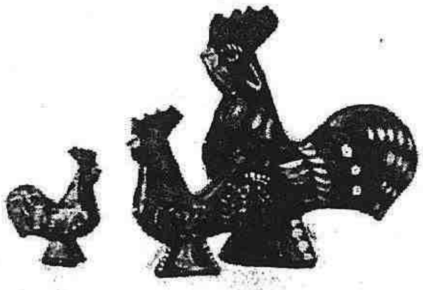
Proceedings of:

The 27th ASU Conference Model Oriented Programming and Simulation

October 11-16, 2001

The Gärdeby Hotel

Rättvik, Sweden



Organized by The Association of SIMULA Users

ASU Secretariat

Box 54

SE-193 22 SIGTUNA

Sweden

Client Side Web Simulation Engine

Jaroslav Sklenar
Department of Statistics and Operations Research
University of Malta, Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jskl1@stator.um.edu.mt

Abstract

JavaScript together with HTML support easy creation of documents that can contain user-friendly input of validated data. The data can be then processed in the user's computer and the results sent to the server or just displayed by the browser in lucid readable way. For data processing JavaScript represents a high level interpreted language where all important ideas of Object Oriented Programming (OOP) can be applied either directly or with minimum extra effort. All solutions based on JavaScript and HTML are intended to be placed on the web and made thus available literally to everybody who has a browser supporting particular versions of these two languages. Latest versions of both common browsers (Microsoft Internet Explorer and Netscape Communicator) are freely available for personal use, so web is a big uniform environment ready to host resources of practically any kind. In this context several simulation models have been implemented and placed on the web with rather encouraging response. This paper describes a simulation engine that was written entirely in JavaScript and that together with appropriate HTML documents supports a user-friendly development of web hosted simulation models. Design of a simple simulation model is described to show the main ideas. Speed of today's computers and browsers enables simulation of medium size models in reasonable time.

1. Introduction

The name JavaScript used in this paper is owned by Netscape, Microsoft's implementation of the language is called JScript. Both are more or less compatible and both are supposed to follow the standard ECMA-262 (ISO-10262). Anyway, differences between browsers exist, so it is suggested to check the HTML documents and associated JavaScript code at least in the two common browsers. Code presented in this paper works with IE 5.5 and NC 4.73. Obviously explanation of JavaScript constructs is out of the scope of the paper, so if you are not familiar with its basics see for example (Flanagan 1998). Note also that in spite of the beginning of its name, JavaScript is not a version or modification of Java – see for example (Eckel 1998). Papers (Sklenar 1999, 2000a) describe the basic ideas of the JavaScript prototype oriented paradigm, how to use all important techniques of OOP in JavaScript and also some new techniques not available in strongly typed compiled Object Oriented Languages. All these techniques have been used to create reusable code open to future expansion and modifications. The simulation engine described in this paper is made of a number of classes and routines placed in several JavaScript source files (extension js). Next chapter deals with the basic engine that implements the primitives of event based discrete simulation. The following chapter describes the simulation support facilities, in particular queues and transparent collection of statistics. The last chapter is a simple simulation example.

2. Model Timing

Discrete simulation languages and tools are usually classified according to the way of model time control. The advanced process-oriented approach is difficult to implement. Also JavaScript simulation will probably not be applied to large-scale systems. So the classical easy to use and easy to learn event-oriented approach is probably the best choice. The basic ideas are well known and can be found in most simulation textbooks – see for example (Zeigler 1984) and (Pidd 1998). Proper timing of events is implemented by a data structure containing event notices ordered by the time of event occurrence in ascending order. To call it let's use the Simula name Sequencing Set (SQS). Simulation control routine repeatedly removes the first event notice from SQS, copies the event occurrence time to the model time (typically a global variable that can not be modified by the user), and activates a corresponding event routine. There are two basic primitives for programming event-based models:

- Schedule an event at a certain time
- Cancel an already scheduled event.

2.1 SQS

From user's point of view SQS is an ordered list. Linear ordered list with O(n) insertion time would be too slow for bigger models with possibly many scheduled events. We need a fast data structure with preferably O($\log_2 n$) speed of both basic operations: inserting and removing the first item. Another natural requirement is dynamic size of the structure. JavaScript has dynamic arrays, so to host SQS we use a heap[1]. Heap is a perfectly balanced binary tree stored in an array with these properties assuming ascending ordering of items by a certain key:

- The root with minimum key is at the position 1,
- The two children of a node at position i are at the positions $2i$ and $2i+1$,
- Both children have bigger (or equal) key than the parent.

Note that heap is not a sorted tree, the keys of the two children are not related. There can be mostly one node with one child, all following nodes then have no children. Alternatively there are nodes with two children followed by nodes with no children. Note that to compute the location of a parent, the index of a child is just shifted by one bit to right. Let *size* be the number of items in the heap (*size*+1 is the first free location). These are the simplified algorithms of the two operations, for details see (Cormen et al. 1997).

```
Insert x to heap h                  // sift up operation
     size <- size+1
     i <- size
     while (x.key < h(i div 2).key) and (i>1)
          h(i) <- h(i div 2)        // move parent down
          i <- i div 2
     endwhile
     h(i) <- x
```

[1] Note on terminology: The word "heap" has double meaning. Usually it is a dynamically allocated part of RAM. In the context of data structures "heap" is a special type of a binary tree as described in this paper. Both interpretations are standard and should not be modified.

The algorithm starts in the first free position and moves up in the tree. If the parent has bigger key than the new item, it is moved down. When smaller key or root are reached, the new item is stored. The maximum number of steps is $\log_2 size$ rounded down.

```
Remove first item from heap h          // sift down operation
     first <- heap(1)
     heap(1) <- heap(size)
     size <- size-1
     while (there is a smaller child)
          swap the parent and the child
          move down
     endwhile
     return first
```

The algorithm takes the first (smallest) item and moves the last item to its place. Then it moves down in the tree. If the parent has bigger key than a child, they are swapped. When both children (if any) are bigger or end of tree is reached, the movement stops. Again the maximum number of steps is $\log_2 size$ rounded down.
Removing an item is more complicated. So far we use a simple not very fast approach assuming that canceling an event is relatively rare. The heap is first searched linearly for the item to be removed. If found, it is replaced by the last item. Then it is necessary to restore heap properties be sifting either up or down because there is no relation between last and intermediate items that are not on the path from the last item up to the root.
Heap is implemented as a class with two value properties (array and size) and these methods:

`empty()` = test if the heap is empty,
`clear()` = deleting all items from memory (initialization),
`insert(x)` = inserting *x* to heap (using *x.key*),
`remove(x)` = removing *x* from heap,
`restoreheap(i)` = restoring heap properties from the position *i* down,
`getfirst()` = removing the first item.

Heap is used by the engine to host SQS, but in the model it can also be used as a priority queue. Loose JavaScript typing allows storing various items in one queue. The only requirement is that they all must have a value property *key* comparable by the < , > operators of JavaScript (numbers or texts).

2.2 Simulation control

The basic simulation engine without supporting objects is made of three parts - two JavaScript source files and the user JavaScript code. The two files are:

- `heap.js` = heap implementation (see the previous chapter).
- `engine.js` = the simulation routines that use a heap as the SQS.

These files have to be listed by two <SCRIPT> tags in the heading of the HTML document in this order. The user code can follow or (better) it can be placed into another JavaSript source file(s). These are the basic ideas. During (re)loading of the document the engine creates two global variables: *time* and SQS (empty heap). Events are represented by event notices created by the user and stored in the SQS. Each event notice has a time of the event and any other user-defined data. The time is set by the engine when the event is scheduled. After activation the engine repeatedly removes the first event notice, updates the model time, and activates a user routine that is given the reference to the event notice. Simulation ends by empty SQS or by any user supplied condition. These are the Engine routines:

`evnotice()` is the event notice constructor. It returns an object with the time property, that is used by the engine and should not be accessed by the user. The user can add any other properties especially to distinguish between various types of events and to store any other model dependent data.

`initialize_run(debug)` is a routine that clears the SQS (the previous experiment may have finished with nonempty SQS) and sets the model time to zero. It should be called at the beginning of the model initialization. If the Boolean parameter is true, simulation starts in the debugging mode. Repeatedly after a certain number of debugging messages the user can close the debugging mode and complete the experiment in fast mode.

`schedule(event,tim)` schedules the event whose notice is the first parameter at the time given by the second parameter. The SQS contains the reference to the object instance, so the same reference variable can be used to schedule more events, but all event notices have to be created by the user. Scheduling is a fast insertion into a heap – balanced binary tree.

`cancel(event)` cancels a scheduled event. It first searches for the event notice sequentially. If it is not found, the function returns false. Otherwise the event notice is removed from SQS. This operation can be rather slow if there are many event notices in the SQS, but canceling events is much less frequent than scheduling.

`simulation_run(stats,length)` is the simulation experiment as such. This routine should be called after the model initialization that has to schedule at least the first event. More events can be scheduled later by other preceding events. The two parameters just affect the progress reporting. If the first parameter is true, the time will be reported in the status bar of the browser. This is very time consuming, so turn it off for longer experiments. The second parameter is the expected duration of the experiment. It is shown in the status bar together with time to see the relative progress of the experiment. Nevertheless the experiment can finish earlier or later. This routine ends by reaching the empty SQS or by the user supplied condition - see the routine `finish_run()`.

The above routines are common to all simulation models. Model specific behavior is implemented by two routines that have to be supplied together with code (preferably also a routine) that starts simulation. These are the routines that represent the user's part of simulation control.

`finish_run()` tests whether simulation should be terminated. It is called by the engine after updating the model time just before activating the next user event. It can just test the time against the experiment duration, it can implement a more complicated terminating condition (for example finish the experiment after serving a certain number of customers, etc.) or it can be empty (just return false). In the last case the run will be finished by the empty SQS.

`eventroutine(event)` is activated by the engine. The routine is given the reference to the event notice that has been removed from the SQS. The rest is the user's responsibility.

Typically there will be some properties created by the user used to switch between various types of events. It might be a good idea to keep this routine short and simple and to write routines for various types of events. These routines will then be called from a switch statement testing the event type property of the event notice.

`Starting simulation` also has to be programmed. For example it can be a function activated by pressing a button "Run". This code is supposed to perform the following four activities in the same order:

– Initialization of the Engine by `initialize_run(debug)`,
– Model specific initialization,
– Starting simulation by `simulation_run(stats,length)`,
– Model specific experiment evaluation (that can be alternatively performed by an event scheduled at the required experiment termination time).

3. Simulation support

Model timing is only one of many supporting facilities that are supposed to be included in event-oriented discrete simulation tools:

– *Time control*
– *Generation of (pseudo)random numbers*
– *Automatic collection of statistical data*
– *Statistical analysis*
– *Report generation*
– *Facilities for work with high-level data structures (queues)*
– *Control over the dynamically allocated memory*
– *Good diagnostics and debugging facilities*

Time control has been described in the previous chapter. Generation of random numbers is so far based on the standard JavaScript generator `Math.random()` together with few routines that generate random numbers with exponential, uniform, and normal distribution. Future expansion in this area is supposed. Report generation is solved naturally by the HTML part of the model. JavaScript code just updates the values in the HTML generated layout. There is no need to control the dynamically allocated memory because JavaScript's garbage collector automatically recovers lost memory (familiar situation for a Simula programmer). Debugging is currently a problem unless there is the Microsoft Development Studio installed. Otherwise the basic tool is the function `alert(message)` that stops temporarily the interpreter and displays the message. A function is supplied (together with few other useful functions for common use) that creates and displays a string with names and values of all properties of an object. Next we shall deal briefly with classes that implement queues and transparent collection and computation of statistical data.

3.1 Queues

The paper (Sklenar 2000a) shows how to program multiple inheritance in JavaScript and in the Appendix there is the complete code that implements the FIFO queue. Properties and methods of the FIFO queue are inherited from two superclasses:

- *Generic queue* is the basic queue implemented in an array. It can just test if it is full or empty and it can fix its size.
- *Statistic queue* is able to keep the time integrals of its length and of the square of its length. Methods are available to compute the average queue length, the variance, and the standard deviation.

These two queues do not implement any particular insertion and removing operations. They are implemented as additional methods of the FIFO queue. FIFO queue implementation is based on a circular list within an array. The result is a queue with three methods: initialization, adding a new item at the end, removing the first item. Keeping and updating statistics is transparent for the user, at the end of the experiment the results (maximum length, average length, variance, standard deviation) are just used - for example displayed in the HTML document. LIFO queue (stack) is also implemented, so together with heap based priority queue there are all three standard queueing methods. So far items can not be removed from FIFO and LIFO queues, so impatient customers are not supported.

3.2 Statistics

Some discrete simulation tools contain transparent collection of statistical data together with automatic computation of statistical parameters. This very user-friendly feature can be found for example in SIMSCRIPT II where it is based on the so-called *left monitoring*. Left monitoring means that if a certain variable (declared to be monitored on the left) appears on the left side of an assignment, a special routine is activated before the actual storing of the right hand side value. This routine is given the value to be stored and can modify it or just use it to update statistics. The second is done automatically in SIMSCRIPT II for all statistically observed variables[2]. Inspired by this tool two classes of statistically observed numbers have been declared: accumulator and tally.

Accumulator class takes into account time. It is in fact similar to the observation of a queue length. Accumulator can be for example the number of working machines. Accumulator keeps automatically the time integrals of its value and of the square of its value. Work with accumulators is done by these methods:

`initiate(x)` prepares accumulation and stores the initial value. The method is supposed to be called during model initialization.
`updateto(newx)` has to be used instead of assignment by the "=" operator. It updates the integrals and stores the new value. There is a similar method `updateby(delta)` with increment as its parameter.

[2] Thanks go to Eugene Kindler who gave me a copy of the article (McNeley 1968) that contains probably the first description of these ideas. There the *left monitoring* is called *store association*. It is interesting that during the discussion O.J.Dahl has suggested an implementation based on class methods as described in this paper (wheel re-invented again).

Another three methods are available to get the time average, the variance, and the standard deviation. Three value properties contain the current, the minimum, and the maximum values.

Tally class is similar, but time is not taken into account. It can be used for example to get statistics on the time spent in a queue. Here the statistics is based only on the values assigned to the particular variable. The methods are similar to the ones of accumulate, but there is only one `update(newx)`. The value properties are the minimum and the maximum values and the number of updates.

Using accumulators and tallies is simple. First they are created and initialized. Then instead of JavaScript assignment by "=" they have to be updated by calling the updating methods. At the end the results are just accessed and used (displayed).

4. Example simulation

Simulation models based on the engine are made of an HTML document that contains script tags referencing the supplied js files and model specific JavaScript code. As an example this chapter describes a simple simulation experiment. The simulated system is a queueing system defined as follows:

- Exponential arrival of customers from an unlimited population, one class of customers.
- Unlimited FIFO queue.
- One server with normally distributed service duration.
- The server breaks down randomly with exponentially distributed intervals between failures.
- Uniformly distributed repair time.
- Customer whose service is interrupted is after the repair served again.

The system is kept simple intentionally to concentrate on the use of the engine and not on the model specific details. Many assumptions can be easily relaxed. The required outputs are:

- Numbers of arrivals and failures.
- Average and maximum waiting time.
- Average, minimum and maximum time spent in the system.
- Average and maximum queue length.

The simulation model is in the Appendix. Commented user JavaScript code is included in the HTML document, alternatively it can be placed into another js file. HTML code is kept as simple as possible, nevertheless the interface is user-friendly. Most of the JavaScript code can be used in other models with minimum changes, because the model specific behavior is implemented by the four event routines. You can visit the author's page to try the model in your browser. You can also download the js files to create models locally. Hint: try to modify the model in order to find the utilization of the server (portion of the time when the server was working). For this purpose use another accumulator with two possible values: 1 when the server works, 0 when it is idle or under repair. Its average is then the utilization.

5. Conclusion

JavaScript programmers can use the engine described in this paper to create web hosted simulation models. Bigger models than the simple one described in chapter 4 will just contain more event routines and more model parameters, the basic principles of simulation remain the same. It might be interesting to find out how big models can still be successfully simulated by JavaScript. Note that during long simulation experiments the browser may display a message that there is a long script running and whether to continue (IE 5.5). Just click yes. The author will very appreciate all comments and suggestions especially regarding possible improvements of the engine. More examples will be provided in future, the simulation support is under permanent development. So consider visiting the author's page to learn about updates.

References

Cormen Thomas H., Leiserson Charles E., Rivest Ronald L. (1997) *Introduction to Algorithms*. The MIT Press.

Darnell Rick et al. (1998) *HTML 4 Unleashed. Professional Reference Edition.* Sams.net Publishing.

Eckel Bruce (1998) *Thinking in Java.* Prentice Hall. Inc.

Flanagan David (1998) *JavaScript - The Definitive Guide.* O'Reilly & Associates Inc.

McNeley John L. (1968) *Compound Declarations.* Proceedings of the IFIP Working Conference on Simulation Languages, Oslo, May 1967. North Holland, p.292 – 303.

Pidd Michael (1998) *Computer Simulation in Management Science*. John Wiley & Sons.

Sklenar Jaroslav (1999) *Prototype Oriented Simulation in JavaScript*. Proceedings of the 25th Conference of the ASU System Modelling Using Object Oriented Simulation and Analysis, Balatonfured, p. 61 – 71.

Sklenar Jaroslav (2000) *Object Oriented Programming in JavaScript.* Proceedings of the 26th Conference of the ASU Object Oriented Modeling and Simulation, Malta, p. 35 - 42.

Sklenar Jaroslav (2000) *Simulator of Queueing Networks.* Proceedings of the 26th Conference of the ASU Object Oriented Modeling and Simulation, Malta, p. 127 - 135.

Zeigler Bernard P. (1984) *Theory of Modelling and Simulation*. Robert E. Krieger Publishing Company.

Appendix: Example simulation (a complete HTML document with JavaSript code).

```
<HTML><HEAD>
<TITLE>Example Simulation in JavaScript #2</TITLE>
<SCRIPT LANGUAGE="JavaScript" SRC="utilities.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="heap.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="engine.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="statist.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="queuesm.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript" SRC="randvar.js"></SCRIPT>
<SCRIPT LANGUAGE="JavaScript">

//******************** USER SIMULATION ROUTINES ***************************

function finish_run(){           // Whether to finish the simulation run
   return (time>runlength);      // has to be supplied
};

//...................................................................

function eventroutine(event) {   // The event routine (must be supplied)
  switch (event.eventtype) {     // switching between 4 types of events
  case 1:
    arrival(); break;
  case 2:
    end_of_service(); break;
  case 3:
    breakdown(); break;
  case 4:
    end_of_repair(); break;
  default:
    alert("Wrong eventtype: " + event.eventtype);
  };
};

//=========================== Event routines ===============================

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

//...................................................................

function end_of_service() { // service ends
   systemtime.update(time - served.nettimein);
   served=null;
   if (!queue.empty()) {     // queue empty ?
     served = queue.removefirst(); // no - serve the 1st one
     waittime.update(time - served.nettimein);
```

```
      var ev = new evnotice(); // scheduling end of service
      ev.eventtype = 2;
      served.event = ev;
      schedule(ev,time + normal_pos(meanservice,serviceStd));
   };
};

//..............................................................................

function breakdown() {  // server failure
   failures++;
   serverOK = false;
   if (served!=null) {  // service on ?
     cancel(served.event);  // yes - cancel it
   };
   var ev = new evnotice(); // scheduling end of repair
   ev.eventtype = 4;
   schedule(ev,time + uniform(repairfrom,repairto));
};

//..............................................................................

function end_of_repair() {  // server repaired
   serverOK = true;
   var ev = new evnotice(); // scheduling next failure
   ev.eventtype = 3;
   schedule(ev, time + exponential(meanbreakinterval));
   if ((served==null)&&(!queue.empty())) { // new service can start ?
     served = queue.removefirst(); // yes - serve the 1st one
     waittime.update(time - served.nettimein);
   };
   if (served!=null) {    // starting service ? (new or old)
     ev = new evnotice(); // scheduling end of service
     ev.eventtype = 2;
     served.event = ev;
     schedule(ev,time + normal_pos(meanservice,serviceStd));
   };
};

//==============================================================================

function set_parameters() {       // Copying model & control parameters
   meaninterval = parseFloat(doc.I11.value); // mean arrival interval
   meanservice = parseFloat(doc.I21.value);  // mean service duration
   serviceStd = parseFloat(doc.I22.value);   // service standard deviation
   meanbreakinterval = parseFloat(doc.I31.value);// mean failures interval
   repairfrom = parseFloat(doc.I41.value);   // minimum repair time
   repairto = parseFloat(doc.I42.value);     // maximum repair time
   runlength = parseFloat(doc.CI1.value);    // length of the experiment
};

//..............................................................................

function evaluation() {        // Experiment evaluation
   doc.R1.value = arrivals;
   doc.R2.value = waittime.average();
   doc.R3.value = waittime.max;
   doc.R4.value = systemtime.average();
   doc.R5.value = systemtime.min;
   doc.R6.value = systemtime.max;
   doc.R7.value = queue.average();
   doc.R8.value = queue.maxqlength;
   doc.R9.value = failures;
};
```

```
//.....................................................................

function simulation() {          // Simulatiom experiment
   doc = document.forms[0];
   initialize_run(doc.CI3.checked); // This prepares the engine
   set_parameters();             // Model & Control parameters
   // Model & statistics initialization
   queue.initiate(1000,0);
   served = null;
   serverOK = true;
   arrivals = 0;
   failures = 0;
   waittime.initiate();
   systemtime.initiate();
   // Preparation - at least one event has to be scheduled:
   var ev = new evnotice();   // Scheduling 1st arrival
   ev.eventtype = 1;
   schedule(ev,exponential(meaninterval));
   ev = new evnotice();          // Scheduling 1st failure
   ev.eventtype = 3;
   schedule(ev,exponential(meanbreakinterval));
   // This starts the simulation:
   simulation_run(doc.CI2.checked,runlength);
   evaluation();                 // Experiment evaluation
};

//.....................................................................

function customer(tim) {         // customer constructor
   this.nettimein = tim;         // time of entering system
   this.event = null;            // end of service event notice
};

//=============== Head Code (creating global variables) ===================

// Model parameters
var meaninterval = 0;            // mean interval between arrivals
var meanservice = 0;             // mean service duration
var serviceStd = 0;              // service standard deviation
var meanbreakinterval = 0;       // mean interval between failures
var repairfrom = 0;              // minimum repair time
var repairto = 0;                // maximum repair time
// Model objects
var queue = new FifoQueue('Q',1000);  // the queue
var served = null;               // served customer
var serverOK = true;             // server status
// Control parameters
var runlength = 0;               // length of the experiment
// Statistics
var waittime = new tally();      // waiting time
var systemtime = new tally();    // system time
var arrivals = 0;                // # of arrivals
var failures = 0;                // # of failures
// Auxiliary
var doc = null;                  // to access document objects

//=========================================================================
</SCRIPT>
</HEAD>

<BODY BGCOLOR="#000000" TEXT="#00ff00" LINK="#ffff00">
<FORM NAME="tester">
<CENTER>
```

```
<H1>JavaScript Simulation Engine - Example Simulation #2</H1></CENTER>

<H3><B>Model Parameters:</B></H3>
(All times in minutes)
<TABLE BORDER>
<TR>
<TD>Exponential intervals between arrivals</TD>
<TD>Mean:&nbsp;<INPUT TYPE="text" NAME="I11" SIZE=5 VALUE="5"></TD>
<TD></TD>
</TR>
<TR>
<TD>Normal service duration</TD>
<TD>Mean:&nbsp;<INPUT TYPE="text" NAME="I21" SIZE=5 VALUE="4"></TD>
<TD>Std:&nbsp;<INPUT TYPE="text" NAME="I22" SIZE=5 VALUE="2"></TD>
</TR>
<TR>
<TD>Exponential interval between failures</TD>
<TD>Mean:&nbsp;<INPUT TYPE="text" NAME="I31" SIZE=5 VALUE="20"></TD>
<TD></TD>
</TR>
<TR>
<TD>Uniform repair time</TD>
<TD>From:&nbsp;<INPUT TYPE="text" NAME="I41" SIZE=5 VALUE="1"></TD>
<TD>To:&nbsp;<INPUT TYPE="text" NAME="I42" SIZE=5 VALUE="10"></TD>
</TR>
</TABLE>

<P><HR>
<H3><B>Simulation Control:</B></H3>
Experiment Duration:&nbsp;<INPUT TYPE="text" NAME="CI1" SIZE=5 VALUE="100">
&nbsp;&nbsp;&nbsp;&nbsp;
Show Status:&nbsp;<INPUT TYPE="checkbox" NAME="CI2" CHECKED>
&nbsp;&nbsp;&nbsp;&nbsp;
Debugging:&nbsp;<INPUT TYPE="checkbox" NAME="CI3">
&nbsp;&nbsp;&nbsp;&nbsp;
<INPUT TYPE="button" VALUE="&nbsp;Run&nbsp;" NAME="CB1"
onClick="simulation()">

<P><HR>
<H3><B>Results:</B></H3>

<TABLE BORDER>
<TR>
<TD> Number of arrivals </TD>
<TD> <INPUT TYPE="text" NAME="R1" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB1"
   onClick="alert('Total number of arrivals to the system.')"></TD>
</TR>
<TR>
<TD> Average waiting time </TD>
<TD> <INPUT TYPE="text" NAME="R2" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB2"
   onClick="alert('Average time spent in the queue till first start of
service.')"></TD>
</TR>
<TR>
<TD> Maximum waiting time </TD>
<TD> <INPUT TYPE="text" NAME="R3" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB3"
   onClick="alert('Maximum time spent in the queue till first start of
service.')"></TD>
</TR>
<TR>
<TD> Average time in system </TD>
```

```
<TD> <INPUT TYPE="text" NAME="R4" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB4"
   onClick="alert('Average time spent in the system.')"></TD>
</TR>
<TR>
<TD> Minimum time in system </TD>
<TD> <INPUT TYPE="text" NAME="R5" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB5"
   onClick="alert('Minimum time spent in the system. Note that minimum
waiting time is zero.')"></TD>
</TR>
<TR>
<TD> Maximum time in system </TD>
<TD> <INPUT TYPE="text" NAME="R6" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB6"
   onClick="alert('Maximum time spent in the system.')"></TD>
</TR>
<TR>
<TD> Average queue length </TD>
<TD> <INPUT TYPE="text" NAME="R7" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB7"
   onClick="alert('Average queue length including time when it was
empty.')"></TD>
</TR>
<TR>
<TD> Maximum queue length </TD>
<TD> <INPUT TYPE="text" NAME="R8" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB8"
   onClick="alert('Maximum reached length of the queue (must be less than
1000).')"></TD>
</TR>
<TR>
<TD> Number of failures </TD>
<TD> <INPUT TYPE="text" NAME="R9" SIZE=25> </TD>
<TD ALIGN="CENTER"> <INPUT TYPE="button" VALUE="Note" NAME="RB9"
   onClick="alert('Number of server failures.')"></TD>
</TR>
</TABLE>
<P><HR>
</FORM></BODY></HTML>
```

Proceedings of 28th ASU Conference

The Simulation Languages

September 26 - October 1, 2002
The Savings Bank Academy Hotel
Brno, The Czech Republic

The conference is arranged by

The Brno University of Technology
Faculty of Information Technology

in cooperation with

The **Association of SIMULA Users**
ASU Secretariat
Box 54
SE – 193 22 SIGTUNA
Sweden

Discrete Event Simulation in JavaScript

Jaroslav Sklenar
Department of Statistics and Operations Research
University of Malta, Msida MSD 06, Malta
Web: http://staff.um.edu.mt/jskl1/
E-mail: jaroslav.sklenar@um.edu.mt

Abstract

JavaScript is a high-level interpreted language where all important principles and techniques of Object Oriented Programming can be applied either directly or with minimum extra programming effort. Together with HTML JavaScript supports creation of documents that can contain user-friendly input of validated data, data processing and lucid presentation of outputs. Solutions based on JavaScript and HTML are intended to be placed on the web and made thus available literally to everybody who has a browser supporting particular versions of these two languages. These capabilities can be applied to create various web-hosted problem-solving tools. Some of them can contain simple and medium-scale simulation models. Several simulation models have already been implemented and placed on the web with rather encouraging response. A simple event oriented simulation engine written entirely in JavaScript was implemented and made available for download. The paper describes its use to build simulation models in JavaScript with special emphasis on their link to corresponding HTML documents in order to simplify programming as much as possible. Design of a simulation model of a general single queue system is used to demonstrate the main ideas.

Introduction

The papers (Sklenar 1999, 2000) describe the basic ideas of the JavaScript[1] prototype oriented paradigm, how to use all important techniques of Object Oriented Programming (OOP) in JavaScript and also some new techniques not available in strongly typed compiled Object Oriented Languages (OOL). In particular the programmed inheritance as defined in the paper (Sklenar 2000) enables creation of "subclasses" that inherit only selected methods of the superclass. Thus we can create simplified versions of general superclasses. All these techniques can be used to create reusable code open to future expansion and modifications. In other words in an interpreted JavaScript environment with loose typing we can use the techniques typical for classical compiled strongly typed OOLs like for example Simula or Java[2]. The paper (Sklenar 2001) deals with the implementation of a simulation engine that was written entirely in JavaScript and that together with appropriate HTML documents supports a user-friendly development of web hosted tools that contain simple and medium scale simulation models. The engine is based on the classical event-oriented approach with two primitives: *schedule an event at a certain time* and *cancel a scheduled event*. These primitives are implemented as calls to routines with appropriate parameters. Other routines are

[1] The name JavaScript used in this paper is owned by Netscape, Microsoft's implementation of the language is called JScript. Both are more or less compatible and both are supposed to follow the standard ECMA-262 (ISO-10262). Anyway, differences between browsers exist, so it is suggested to check the HTML documents and associated script code at least in the two common browsers.
[2] Note that in spite of the similar C-based syntax, Java and JavaScript should be considered as two different high-level languages.

available in order to prepare the simulation experiment, to start the simulation and to create event notices. These routines are imported by JavaScript source files (extension js) listed in the heading of the HTML document. Simulation support facilities are also available, in particular generation of random numbers, work with queues and transparent collection and computation of statistics. Visit the following page to read a simple manual and to download the engine: *http://staff.um.edu.mt/jskl1/simweb/engine/engman.html.* This paper will concentrate on the use of the engine and how to link it with the HTML document in order to simplify programming of web hosted simulation models.

The Engine

This chapter summarizes the basic ideas without implementation details. During (re)loading of the document the engine creates two global variables: the *time* and the empty SQS. Events are represented by event notices created by the user and stored in the SQS. Each event notice has a time of the event and any other user-defined data. The time is assigned by the engine when the event is scheduled. After activation the engine repeatedly removes the first event notice, updates the model time, and activates a user routine that is given the reference to the event notice. Simulation ends by the empty SQS or by any user supplied condition. These are the Engine routines:

`evnotice()` is the event notice constructor. It returns an object with the time property, that is used later by the engine and should not be accessed by the user. The user can add any other properties especially to distinguish between various types of events and to store any other model dependent data.

`initialize_run()` is a routine that clears the SQS (the previous experiment may have finished with nonempty SQS) and sets the model time to zero. It should be called at the beginning of the model initialization.

`schedule(event,tim)` schedules the event whose notice is the first parameter at the time given by the second parameter. The SQS contains the reference to the object instance that has to be created by the user.

`cancel(event)` cancels a scheduled event. If the event is not scheduled, the function returns false. Otherwise the event notice is removed from SQS.

`simulation_run(stats,length)` starts the simulation experiment. This routine should be called after the model initialization that has to schedule at least the first event. More events can be scheduled later by other preceding events. The two parameters just affect the progress reporting. If the first parameter is true, the time will be reported in the status bar of the browser. The second parameter is the expected duration of the experiment. It is shown in the status bar together with the time to see the relative progress of the experiment. Nevertheless the experiment can finish earlier or later. This routine ends by reaching the empty SQS or by the user supplied terminating condition - see the routine `finish_run()`.

The above routines are common to all simulation models. Model specific behavior is implemented by two routines that have to be supplied together with the code (preferably also a routine) that starts simulation. These are the routines that represent the user's part of the simulation control. The examples are taken from the model outlined later.

`finish_run()` tests whether simulation should be terminated. It is called by the engine after updating the model time just before activating the next user event. It can just test the time against the experiment duration, it can implement a more complicated terminating condition (for example to finish the experiment after serving a certain number of customers) or it can be empty (just return false). In the last case the run will be finished by the empty SQS. The following is the function of a model where the experiment is finished by either reaching its maximum duration `runlength` or by serving a certain number (`numofcserf`) of customers. If this value is 0, the criterion is ignored :

```
function finish_run() {      // Whether to finish simulatiom run
   if (numofcserf == 0) {
     return (time>runlength)
   } else {
     return (time>runlength)||(numofcser >= numofcserf)
   };
};
```

`eventroutine(event)` is activated by the engine. The routine is given the reference to the event notice that has been removed from the SQS. The rest is the user's responsibility. Typically there will be some properties created by the user used to switch between various types of events. It might be a good idea to keep this routine short and simple and to write routines for various types of events. These routines will then be called from a switch statement testing the event type property of the event notice. The following is the function of a model with two types of events:

```
function eventroutine(event) { // The event routine
  switch (event.eventtype) {  // switching between types of events
  case 1:
    next_arrival(); break;
  case 2:
    end_of_service(event.servnum); break;
  default:
    alert("Wrong eventtype: " + event.eventtype);
  };
};
```

`Starting simulation` also has to be programmed. For example it can be a function activated by pressing a button "Run". This code is supposed to perform the following four activities in the same order:

- Initialization of the Engine by `initialize_run()`,
- Model specific initialization,
- Starting simulation by `simulation_run(stats,length)`,
- Model specific experiment evaluation (that can be alternatively performed by an event scheduled at the required experiment termination time).

The following is the function activated by pressing the button Run and its link to HTML:

```
<INPUT TYPE="button" VALUE="Run" onClick="simulation()">
```

```
function simulation() {          // Simulatiom experiment
   if ((arrival.disttype == "User") && !(arrival.confirmed)) {
     alert("Can't start simulation. Arrival intervals table has
            not been confirmed.");
     return false;
   };
   // Similar tests for other tables
   initialize_run();              // This prepares the engine
   initialization();              // Model & statistics initialization
   var ev = new evnotice();       // Scheduling first arrival
   ev.eventtype = 1;              // User defined property
   var x = arrival.generate();//  Generation of first interval
   intstat.update(x);             // Arrival interval statistics
   schedule(ev,time + x);         // Scheduling the first event
   simulation_run(showstatus,runlength); // Start the experiment
   evaluation();                  // Experiment evaluation
};
```

The user routines perform the tasks typical for event-oriented simulation, their description is out of the scope of the paper.

Simulation Support

In addition to timing there are other typical facilities that are supposed to be included in a user-friendly discrete simulation tool. This chapter lists them together with their implementation in the JavaScript simulation tool dealt with in this paper.

- *Generation of (pseudo)random numbers* is implemented by rather complex classes that declare objects that are basically random numbers with either theoretical or empirical (user-defined) distribution. User-friendly input of the cumulative distribution function table is supported.

- *Automatic collection of statistical data and Statistical analysis* is implemented by the classes *Accumulator* and *Tally* (Simscript II^{TM} terminology is used). They differ in time treatment. *Tally* ignores the time; the statistics is based on the collection of assigned numbers. *Accumulator* statistics is based on time integrals. Basically they are real variables with transparent collection of statistics[3]. The consequence for the user is the assignment. The usual `a = x` has to be replaced by a method call `a.updateto(x)`.

- *Validated input and report generation* makes use of the capabilities of the HTML language. HTML forms are intended to collect data entered by the user, to validate them at user side and to send the results to the server. The last step is skipped, the data is used instead during the simulation experiment. Similarly at experiment evaluation the results can be displayed in form elements and/or written as a text into a separate

[3] Simscript II^{TM} calls this mechanism *left monitoring*. It is based on the idea suggested by McNeley who used the name *store association*. By coincidence the paper (McNeley 1968) was presented at the same conference where Simula was introduced. It is interesting that during the discussion O.J.Dahl has suggested an alternative implementation based on class methods as described in this paper (this was not known to me until I have finished my implementation).

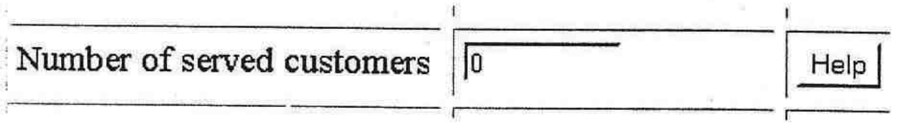
browser window to be used later. Links between JavaScript objects and HTML code are outlined in the next chapter.

- *Facilities for work with high-level data structures (queues)* are represented by three types of queues: FIFO, LIFO and priority queue that is based on a data structure called *heap* that is also used to implement SQS. For details see (Sklenar 2001). The first two queues use multiple inheritance (that can be programmed in JavaScript) to get properties and methods of a generic queue and a statistically observed queue. Then they implement their own methods based on linked lists.

Linking Objects to HTML Code

The following is an example of a validated input. The global variable `numofcserf` contains the number of customers to be served during the experiment or 0 if not used. Note that after a modification there is a test whether the text can be parsed into a non-negative integer that is then displayed. The old value is displayed alternatively. It might be a good idea to derive the name of the form object from the name of the variable (here by adding "in") to avoid confusion of names. The following HTML fragment generates a table row with a label (prompt), the text entry field and a help button that shows text in an alert window.

```
<TR>
<TD> Number of served customers </TD>
<TD><INPUT TYPE="text" NAME="numofcserfin" SIZE=10 VALUE="0"
     ONCHANGE="if (testNonNegIntValue(numofcserfin.value))
                  {numofcserf = parseInt(numofcserfin.value)};
               numofcserfin.value = numofcserf"></TD>
<TD ALIGN="CENTER">
   <INPUT TYPE="button" VALUE="Help"
    onClick="alert('You can terminate the experiment ...')"></TD>
</TR>
```



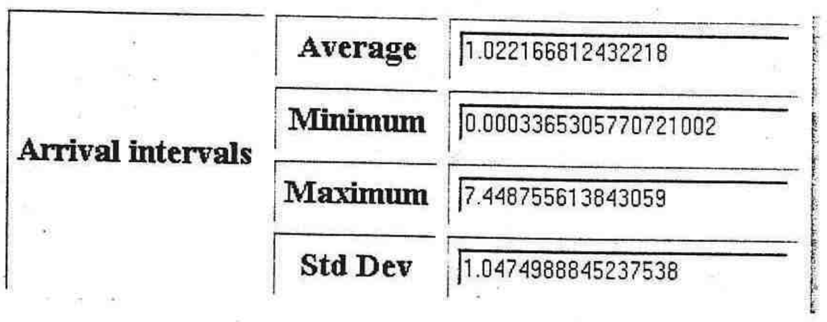
The next example is an output generated by the methods of the associated object. The global variable `intstat` is a *Tally* object used to collect statistics on generated intervals between arrivals. It is declared in the head script code as follows:

```
var intstat = new tally();     // interval statistics
```

At the beginning of an experiment it is initialized and then updated after each generation of an interval between arrivals (see the updating in the function `simulation()` above). At the end of the experiment we need the output of its basic statistics: the average, the minimum and the maximum values and the standard deviation. Let the output be arranged as a part of a table generated by the following HTML fragment:

```
<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>
```



Arrival intervals	Average	1.022166812432218
	Minimum	0.0003365305770721002
	Maximum	7.448755613843059
	Std Dev	1.0474988845237538

The figures were generated by an experiment with the duration 1000; intervals were exponential with mean 1. The results were outputted by a method of the Tally object that was called by the statement `intstat.scrupdate("document.form1.intstat")`:

```
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()");
}};
```

Note that the JavaScript code to be performed is generated automatically by the method and then performed by the function `eval()`. Alternatively it is possible to give the object a name at initialization and to use the name in the outputting method instead of passing the name as its parameter. This approach was used for random numbers. Here the relation between the variable name (`intstat`) and the form objects names like `intstatav` can be used to write generic pieces of HTML code with formal parameters to be replaced by actual ones (`intstat`). The replacement can be done by the "Replace all" operation available in practically all editors or possibly by a special pre-processor (later). There is a similar method that writes the results directly as an HTML document into a separate browser window. The following is the method, its call and the output.

```
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>");
}};

intstat.winupdate("Arrival intervals",resultwindow):

Arrival intervals statistics:
    •  Average: 1.022166812432218
    •  Minimum: 0.0003365305770721002
    •  Maximum: 7.448755613843059
    •  Std Dev: 1.0474988845237538
```

Example Simulation

A model has been developed that enables simulation of single queue multi-channel systems with bulk (batch) arrivals and bulk service. Batch sizes can be constant or random. Server(s) either wait for complete batches to start service or alternatively smaller batches can be served. For arrival intervals and service duration the user either selects a theoretical distribution or enters a distribution in table form (either probabilities or directly the cumulative distribution). Then the user selects the number of servers and the maximum queue length in case of limited system capacity. The population is unlimited due to batch arrivals. There is another model with single arrivals and single service with limited population. Queue organization can be either FIFO or LIFO (stack). All usual simulation results are available. Results can be saved in a separate browser window. Visit the page *http://staff.um.edu.mt/jskl1/simweb/sq2/sq2.html* to experiment with this model. The model is not very complex but certainly not trivial. The code has about 500 lines, most of it declaration, initialization, updating and displaying of quite a big number of parameters and results. Note that during long simulation experiments the browser may display a message that there is a long script running and whether you want to abort the script (IE 5.5). Just click No.

Conclusion

JavaScript together with HTML can be used to create web hosted problem-solving tools that contain simple or medium-scale simulation models. Regarding the speed it seems that if a single queue system has intervals and service duration in the range of minutes and if the experiment duration is such that months long continuous operation is simulated, the experiments take tens of seconds or few minutes.

References

Darnell Rick et al. (1998) *HTML 4 Unleashed. Professional Reference Edition.* Sams.net Publishing.

Eckel Bruce (1998) *Thinking in Java.* Prentice Hall. Inc.

Flanagan David (1998) *JavaScript - The Definitive Guide.* O'Reilly & Associates Inc.

McNeley John L. (1968) *Compound Declarations.* Proceedings of the IFIP Working Conference on Simulation Languages, Oslo, May 1967. North Holland, p.292 – 303.

Pidd Michael (1998) *Computer Simulation in Management Science.* John Wiley & Sons.

Sklenar Jaroslav (1999) *Prototype Oriented Simulation in JavaScript.* Proceedings of the 25th Conference of the ASU System Modelling Using Object Oriented Simulation and Analysis, Balatonfured, p. 61 – 71.

Sklenar Jaroslav (2000) *Object Oriented Programming in JavaScript.* Proceedings of the 26th Conference of the ASU Object Oriented Modeling and Simulation, Malta, p. 35 - 42.

Sklenar Jaroslav (2001) *Client Side Web Simulation Engine.* Proceedings of the 27th Conference of the ASU Model Oriented Programming and Simulation, Rattvik, p.1 - 13.

Nástroje Diskrétní Simulace

Habilitační práce

Vysoké učení technické v Brně

Fakulta informačních technologií

Jaroslav Sklenář Duben 2003