Popis simulační knihovny SIMLIB

Petr Peringer

7. prosince 1997

Obsah

1 Úvod

Simulační knihovna SIMLIB je vyvíjena od roku 1990 na Ústavu informatiky a výpočetní techniky FEI VUT Brno. Knihovna je implementována na počítačích třídy PC pod MS-DOS (překladače Borland C++ nebo GNU C++) a pod operačním systémem Linux (překladač GNU C++). Knihovna je přenositelná i na jimé platformy, vyžaduje však úpravu v modulu implementujícím procesy. SIMLIB poskytuje základní prostředky pro popis spojitých, diskrétních i kombinovaných modelů a prostředky pro řízení simulace. Při tvorbě simulačních modelů a experimentování s nimi lze použít různá integrovaná prostředí, která umožňují interaktivní tvorbu a ladění modelů.

Knihovna usnadňuje efektivní popis modelů přímo v jazyce C++, není tedy nutný překladač simulačního jazyka. Tato koncepce má své výhody i nevýhody. Je možné používat všech ostatních prostředků vytvořených v C++ (např. grafické knihovny a uživatelská rozhraní). Uživatel také není nijak omezován při případném doplňování prostředků knihovny. Za nevýhodu lze považovat nemožnost dodatečných syntaktických a sémantických kontrol, které by se mohly provádět při použití překladače simulačního jazyka. Na straně uživatele se předpokládá základní znalost programování v jazyce C++.

Následující kapitoly vysvětlují základní principy použití SIMLIB pro modelování diskrétních, spojitých a kombinovaných modelů. Výklad je doplněn příklady s popisem funkce jednotlivých objektů modelu.

2 Objektově orientovaná simulace

Model je v SIMLIB chápán jako množina prvků (entit), které jsou spolu navzájem propojeny vazbami. Tyto vazby spolu s chováním prvků určují chování systému jako celku. Podobný přístup je podstatou objektově orientovaného programování, jehož principy se poprvé objevily v šedesátých letech v jazyce SIMULA 67. Objektově orientovaný program je tvořen množinou objektů, které spolu navzájem komunikují — posílají si zprávy.

Rozdělení systému na jednotlivé objekty je závislé na účelu modelu. Mezi objekty modelu můžeme najít takové, které mají shodné podstatné charakteristiky a ty pak můžeme zařadit do jedné třídy objektů. Třída definuje vnitřní strukturu objektů, reakce objektů na vstupy (zprávy) a vlastní chování objektu v čase.

Objekty modelu provádějí určité akce jako odezvu na přijímané zprávy a současně provádějí jiné akce, které jsou nezávislé na přijímaných zprávách. Akce mění stav objektu, jenž je dán obsahem jeho vnitřních datových struktur.

Každý objekt má definované akce, které realizuje když je vytvořen (inicializuje se) a když je rušen. Tyto akce se nazývají konstruktory a destruktory. Stejně jako ostatní akce, související s popisem chování objektu, patří k takzvaným metodám. Přijetí zprávy odpovídá vyvolání metody a metoda pak popisuje akce, kterými má objekt na tuto zprávu reagovat.

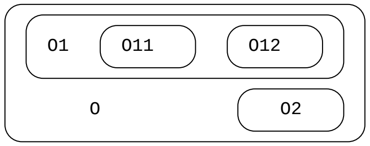
Mezi třídami objektů modelu lze nalézt jisté vzájemné vztahy. Tyto vztahy mají obvykle hierarchický charakter (částečné uspořádání). Některé třídy popisují obecné vlastnosti objektů, jiné je více konkretizují. Toho lze výhodně využít při návrhu tříd modelu tak, aby třídy konkrétní mohly využít všeho, co již definovaly třídy obecné. Tato hierarchie tříd je definována relací dědičnosti. Třída

může zdědit vlastnosti své bázové třídy, přidávat nové vlastnosti a případně modifikovat vlastnosti zděděné. Takto pojatá hierarchie tříd umožňuje v maximální míře využívat již existujících tříd, což zjednodušuje a zpřehledňuje implementaci modelů. Hierarchie tříd je v podstatě uspořádáním abstrakcí, obvykle od nejobecnějších po konkrétní. Například:

```
Objekt
   Dopravní prostředek
      Automobil
         Nákladní automobil
         Osobní automobil
   Osoba
      Učitel
      Žák
```

Postup od obecných tříd ke konkrétním odpovídá postupu shora dolů, lze však postupovat i jinak. Vytvoření vhodné hierarchie tříd obvykle závisí na řešeném problému.

Hierarchická struktura existuje i na úrovni objektů modelu. Jde o vzájemný vztah objektů z hlediska jejich zahrnutí do jiných objektů jako jejich částí. Tato relace se označuje "patří do" ("is part of"), má jiné použití než dědičnost ("is kind of") a definuje hierarchii objektů.



Obrázek 1: Příklad hierarchického uspořádání objektů

Tento způsob popisu problémů je velmi přirozený, protože lze vyjádřit jednoznačný vztah mezi objekty modelu (programu) a objekty modelovaného systému. Metod pro analýzu problémů a návrh objektově orientovaných programů lze využít i v oblasti modelování a simulace. Proto zde uvedeme základní pojmy objektově orientovaného programování.

Objektově orientované programování je metoda implementace, při níž jsou programy tvořeny spolupracujícími skupinami objektů, z nichž každý reprezentuje instanci některé třídy, a třídy jsou navzájem v relaci dědičnosti. Základní principy objektově orientovaného programování:

Abstrakce ukazuje na podstatné charakteristiky objektu, které jej odlišují od všech ostatních druhů objektů a tedy poskytuje přesně definované konceptuální hranice, vztažené k perspektivě pozorovatele.

Zapouzdření (encapsulation) Zapouzdřením se rozumí ukrývání všech detailů objektu, které nepřispívají k jeho podstatným charakteristikám, viditelným vně objektu.

Modularita vyjadřuje vlastnost systému, jenž byl dekomponován do množiny modulů s přesně definovanými vazbami.

Hierarchie je uspořádání nebo seřazení abstrakcí. Příkladem je hierarchie dědičnosti tříd v objektově orientovaném programování.

Tyto principy jsou pro objektově orientované programování podstatné, následující tři jsou méně důležité.

Typování je takové uplatnění třídy objektu, že objekty různých tříd nemohou být zaměňovány, nebo nanejvýš mohou být zaměňovány jen velmi omezeným způsobem.

Paralelismus (concurrency) je vlastnost, která odlišuje aktivní objekt od neaktivního. Při simulačním modelování dynamických systémů se tato vlastnost stává podstatnou.

Persistence je vlastnost objektu, která definuje dobu jeho existence v čase a prostoru. Například globální proměnné existují pouze při běhu programu, ve kterém jsou definovány, naopak soubory existují, i když program neběží.

Tyto principy nejsou v oblasti programování nové, ale objektově orientované programování je vhodným způsobem kombinuje. Požadavky, které jsou kladeny na objektově orientovaný jazyk můžeme shrnout do následujících bodů:

- jazyk musí podporovat objekty jako datové abstrakce; rozhraní objektů je definováno pomocí pojmenovaných operací, a každý objekt má (skrytý) vnitřní stav
- každý objekt je určitého typu (patří do určité třídy)
- třídy mohou dědit atributy z nadtříd

Existuje mnoho objektově orientovaných programovacích jazyků, největšího rozšíření dosáhly především jazyky Smalltalk a C++. Bližší informace o objektově orientovaném programování jsou dostupné v literatuře.

3 Struktura simulačního programu

Obecnou strukturu simulačního programu v C++ znázorňuje příklad:

```
#include "simlib.h"

<definice tříd>
<definice funkcí>
<deklarace globálních objektů>

<definice funkce  main  - popis experimentu>
```

Každý model musí obsahovat dovoz rozhraní simulační knihovny direktivou `#include`, potom následuje popis modelu a popis experimentu. V případě rozsáhlých modelů můžeme rozdělit popis modelu a experimentu do několika souborů (modulů), z nichž každý má tuto strukturu. Popis experimentu (funkce `main`) smí být uveden pouze v jednom modulu.

Deklarace tříd, objektů a funkcí mohou být v libovolném pořadí; platí pouze zásada, že objekt, funkci nebo třídu nelze použít před příslušnou deklarací.

3.1 Popis modelu

Model je tvořen množinou objektů, které jsou navzájem propojeny. Propojení objektů umožňuje jejich vzájemnou komunikaci, která definuje chování modelu.

3.1.1 Vytváření a rušení objektů

V objektově orientovaném popisu systémů je třeba používat různé druhy objektů podle způsobu jejich vytvoření. Podle požadavků na dobu existence objektu lze použít různé způsoby vytvoření objektů podle toho, má-li být objekt statický nebo dynamický. Předpokládáme-li třídu Zakaznik, můžeme vytvořit objekty této třídy například takto:

```
(1)  new Zakaznik;

(2)  Zakaznik *ptr;
     ptr = new Zakaznik;

(3)  Zakaznik Z;
```

V prvním případě je dynamicky vytvořen nový objekt třídy Zakaznik a takto vytvořený objekt není nijak identifikovatelný. Proto je tento způsob použitelný pouze pro objekty, na které nebude třeba se explicitně odkazovat. Druhý případ je doplněn o ukazatel na objekt třídy zákazník. Tento ukazatel potom slouží k identifikaci konkrétního zákazníka při komunikaci s ním. Dynamické objekty lze vytvářet i rušit v průběhu simulace. Třetí způsob odpovídá vytvoření globálního statického objektu s identifikací jménem (identifikátorem). Tento objekt existuje po celou dobu běhu simulačního programu.

Objekty, definované tvůrcem modelu, mohou být globální nebo lokální z hlediska jejich umístění v jiných objektech. Objekty globální jsou dostupné vždy, objekty uvnitř třídy (lokální) jsou dostupné pouze když to tato třída dovolí specifikací public. Lokální objekty jsou součástí jiných tříd objektů:

```
class X : public Process {
  Histogram H;
  double  StartTime;
  int i;
public:
  X() : H("X.H",0,0.1,10), i(0) {};
  ...
};
```

Z ukázky je patrná syntaxe volání konstruktoru lokálního objektu H. Každý objekt třídy X obsahuje histogram H, který je nedostupný objektům jiné třídy (má implicitně specifikaci private).

Další členění objektů je z hlediska časového. Objekty, které v modelu existují po celou dobu simulace, nazveme statické a objekty, které vznikají a zanikají v průběhu simulace, budeme nazývat dynamické.

Protože objekty modelu je nutné identifikovat (pro čitelný výstup), musí se při vytváření objektů některých tříd uvést jejich textové pojmenování:

globální objekty:

```
Facility  F1("Zařízení 1");
Store     S1("Sklad 1", Kapacita);
Histogram H("Četnost hodnot x",Od,Krok,10);
```

dynamicky vytvořené objekty:

```
Store *ptr;
ptr = new Store("Sklad X",10000);
```

objekty vnořené ve třídě:

```
class X {
  Facility F;
public:
  X() : F("Lokální zařízení")  {}
  ...
};
```

Pro zrušení objektu odvozeného od třídy Process lze použít metodu Cancel, která provede potřebné operace. Tatáž operace se provede implicitně po ukončení popisu chování objektu v metodě Behavior. Dynamicky vytvořené objekty lze rušit operátorem delete aplikovaným na ukazatel na objekt:

```
delete objptr;
```

Statické globální objekty nelze rušit explicitně, ruší se automaticky po ukončení simulačního programu. Jazyk C++ volá automaticky při rušení každého objektu speciální metodu — destruktor, který může provést případné operace, nutné pro jeho korektní odstranění z modelu.

3.2 Řízení simulace

Řízení simulačního experimentu popisuje funkce `main`. Zde se provádí inicializace modelu, vytvoření objektů modelu a jejich aktivace (tj. start procesů, které probíhají uvnitř objektů). Prototypová verze popisu experimentu je uvedena v příkladu:

```
int main() {
  <příkazy1>
  Init(<počáteční čas>,<koncový čas>);
  <příkazy2>
  Run();
  <příkazy3>
  return 0;
}
```

Na počátku funkce `main` lze provést příkazy, nesouvisející přímo s modelem a jeho inicializací (příkazy1). Zde je vhodné otevřít výstupní soubor, případně zobrazit úvodní informace o modelu.

Funkce `Init` zahajuje inicializační fázi experimentu. Nastaví počáteční modelový čas a zaznamená koncový čas pro simulaci. Zároveň inicializuje všechny objekty systému pro řízení simulace (např. kalendář). V této fázi experimentu

uživatel inicializuje svoje objekty (příkazy2). Typickým příkazem v této části popisu experimentu je vytvoření, inicializace a aktivace objektů modelu.

Potom následuje vlastní simulační běh vyvolaný funkcí `Run`. Po ukončení simulace následuje obvykle tisk a vyhodnocování výsledků experimentu (příkazy3).

Posloupnost `Init(); <příkazy2> Run(); <příkazy3>` lze mnohokrát opakovat v cyklu s různými parametry modelu, což je vhodné především pro optimalizační experimenty. Je však zapotřebí opatrnosti při inicializaci objektů modelu tak, aby výchozí stav nebyl ovlivněn předchozí simulací, pokud to není žádoucí. Musí se inicializovat všechny objekty modelu, systém řízení simulace inicializuje pouze kalendář událostí, seznam pro `WaitUntil` a svůj stav.

3.3 Výstupy modelu

Výstup informací z modelu lze provádět několika způsoby. Použití prostředků C++ je jedním z nich. Běžně se však používá standardních prostředků knihovny SIMLIB. Většina tříd definuje metodu Output, která zapisuje stav objektu v textovém tvaru do výstupního souboru. Tímto souborem je implicitně standardní výstup. Funkce `SetOutput` umožňuje přesměrování tohoto výstupu do souboru se zadaným jménem.

Prohlížení a tisk výsledků (především spojité) simulace v různých formátech je možné programem GNUplot.

Pro výstup spojitých průběhů do výstupního souboru slouží třída `Graph`. Objekty této třídy provádí periodický zápis hodnot svého vstupu do zvláštního výstupního souboru. Perioda zápisu se určuje při vytváření objektu a je možné ji dynamicky měnit.

Příklad:

```
Integrator x(vstup);
Graph Gx("x", x, 0.01);
```

4 Modelový čas

V knihovně SIMLIB je modelový čas reprezentován globální proměnnou `Time`. Tato proměnná je typu `double` (čas je nezáporné reálné číslo) a její počáteční a koncová hodnota je specifikována v popisu experimentu funkcí `Init`. Interpretace jednotky modelového času je závislá na modelovaném problému. Hodnota proměnné je nastavována systémem pro řízení simulace a nelze ji měnit přiřazovacím příkazem. Použití příkazu

```
Time = 10; // chyba!
```

vyvolá chybu při překladu modelu. Pro blokové výrazy je použitelný blok `T`, který reprezentuje blokový ekvivalent proměnné `Time`. Použití `Time` v blokovém výrazu je chyba, kterou neodhalí překladač, ani kontrola při běhu programu.

5 Diskrétní simulace

Knihovna obsahuje standardní třídy pro popis diskrétního chování objektů, pro modelování standardních obslužných zařízení a pro sběr statistických údajů. Chování objektů lze popsat dvěma způsoby, buď s použitím událostí, nebo procesů.

5.1 Prostředky pro práci s náhodnými veličinami

Při simulaci diskrétních stochastických systémů je zapotřebí modelovat náhodné jevy. K tomu je nutné mít generátory náhodných čísel pro všechna potřebná rozložení. Knihovna SIMLIB obsahuje všechny obvyklé generátory ve formě funkcí, zde uvedeme pouze ty nejdůležitější:

5.1.1 Random

```
double Random();
```

Funkce Random generuje pseudonáhodná čísla s rovnoměrným rozložením na intervalu $< 0, 1)$. Tento generátor je použit jako základní generátor pro ostatní rozložení.

5.1.2 Exponential

```
double Exponential(double E);
```

Generátor exponenciálního rozložení se střední hodnotou E.

5.1.3 Normal

```
double Normal(double M, double S);
```

Normální rozložení se střední hodnotou M a rozptylem S.

5.1.4 Uniform

```
double Uniform(double L, double H);
```

Rovnoměrné rozložení na intervalu $< L, H)$.

5.1.5 Příklady použití generátorů

```
Wait(Exponential(10));

if(Random()<0.33) S1();
else              S2();

int pole[10] = { 10,10,20,20,20,30,30,30,30,30 };
//...
X = pole[Random()*10];

Wait(Uniform(100,150));
```


5.2 Události

Pro popis jednorázových dějů, které se mohou periodicky opakovat, je určena abstraktní třída `Event`. Třídy objektů, odvozené z této třídy, musí definovat chování události v metodě `Behavior`, podobně jako u procesů. Na rozdíl od procesů však popis události není přerušitelný. Pro naplánování události na čas `t` lze použít metodu `Activate(t)`.

5.3 Procesy

Pro popis třídy objektů s vlastním dynamickým chováním je v SIMLIB definována abstraktní třída `Process`. Každá třída, která zdědí třídu Process, musí specifikovat chování objektů této třídy v čase. Chování se popisuje v metodě `Behavior` posloupností příkazů. Základní struktura popisu třídy je uvedena v příkladu:

```
class Zakaznik : public Process {
  <atributy zákazníka>
  void Behavior()
  {
    <příkazy popisující chování objektu v čase>
  }
public:
  Zakaznik(<parametry>) { <inicializace atributů> }
  ~Zakaznik(<parametry>) { <rušení atributů> }
};
```

Popis chování objektů v metodě `Behavior` připomíná popis procedury, může však, na rozdíl od ní, obsahovat i příkazy, které způsobují čekání. To znamená, že popis chování je v modelovém čase na určitou dobu přerušitelný[1]. Po aktivaci objektu se začne provádět posloupnost operací v metodě `Behavior` stejně, jako při provádění obvyklé procedury. V okamžiku, kdy se narazí v tomto popisu činnosti objektu na příkaz, který představuje čekání, je provádění příkazů pozastaveno. Pokud proces čeká, mohou běžet ostatní procesy.

Příkaz `Wait` použitý v popisu chování objektu provádí potlačení činnosti objektu na určitou dobu (tato činnost odpovídá čekání objektu). Příkaz má tvar:

```
Wait( <aritmetický výraz> );
```

Hodnota výrazu udává dobu, po kterou bude pozastavena aktivní činnost objektu. Jestliže tedy objekt O v modelovém čase t provede příkaz Wait(d) ve svém popisu chování, pak se stane po dobu d pasivním a příští aktivace (obnovení činnosti) nastane v modelovém čase $t + d$. Okamžik $t + d$ budeme nazývat reaktivačním okamžikem procesu nebo také kritickým okamžikem procesu.

Příští kritický okamžik je skrytým atributem každého objektu. Čas příštího kritického okamžiku je stanoven buď explicitně příkazem `Wait(d)` (pak je roven `Time + d`, kde `Time` je aktuální hodnota modelového času), nebo implicitně

[1] Přerušitelné procedury jsou implementovány s využitím funkcí setjmp a longjmp ze standardní knihovny jazyka C. Musí být doplněny několika řádky kódu v jazyku symbolických instrukcí, proto SIMLIB není zcela přenositelná.

příkazem `WaitUntil(B)` (pak je roven nejbližší hodnotě modelového času, ve kterém se predikát B stane pravdivým).

Každý objekt je charakterizován třídou, svým stavem a identitou (například jménem). V průběhu simulace může objekt vzniknout, zaniknout, může být právě aktivní a nebo může čekat (pasivní stav procesu) na určitou událost nebo na určitý stav modelu.

Po vzniku je objekt v neaktivním stavu. Odstartování procesu (jeho aktivace) se provede metodou `Activate`. Objekt se může sám aktivovat tím způsobem, že provede svou aktivaci jako akci v konstruktoru. Poznamenejme, že k aktivaci nového objektu může dojít až po přerušení právě probíhajícího procesu.

Po provedení posledního příkazu v popisu chování objektu tento objekt automaticky zaniká. Zánik objektu lze případně kontrolovat tím, že definujeme destruktor s požadovanými operacemi.

5.3.1 Kvaziparalelní provádění procesů v SIMLIB

I když sémantika modelu je postavena na paralelně probíhajících procesech, nelze ignorovat skutečnost, že vlastní výpočet (simulace) probíhá na jednom reálném procesoru. Z toho plyne nutnost řešit zpracování simulačního programu kvaziparalelně.

Principy kvaziparalelního zpracování procesů jsou popsány v [1]. Popisu chování objektu třídy odvozené ze třídy Process odpovídá příslušná metoda Behavior. Tato metoda obsahuje příkazy, které mohou měnit stav daného objektu (změnou atributů) nebo stav ostatních objektů modelu (pokud je to dovoleno). Právě běžící proces provádí akce, popsané v metodě Behavior právě aktivního objektu, který je identifikován ukazatelem Current.

Priorita procesu je definována atributem `Priority`. Při vzniku objektu je možné zadat jeho prioritu, implicitně je nejnižší, tj. nulová. Prioritu probíhajícího procesu můžeme dynamicky měnit přiřazovacím příkazem:

```
Priority = <aritmetický výraz>;
```

V případě plánování reaktivace procesů na stejný modelový čas se nejdříve provede událost procesu s vyšší prioritou (vyšší hodnotou atributu `Priority`). V případě shodných priorit procesů se dříve provede proces, který byl naplánován dříve.

5.4 Zařízení

Obslužné zařízení popisuje třída `Facility`[2]. Zařízení je objekt, určený k popisu specifického typu interakcí procesů, označovaného jako výlučný (exkluzívní) přístup. Problém výlučného přístupu lze formulovat takto: každý z m procesů systému požaduje takový přístup k abstraktnímu zařízení nebo zdroji Z, který vylučuje, aby v kterémkoli okamžiku sdílel zařízení Z více, než jeden proces.

S příklady výlučného přístupu se setkáváme téměř v každém systému hromadné obsluhy. Příkladem může být benzinové čerpadlo, poštovní úředník u přepážky, nebo terminál počítače. Rovněž samotný mechanismus kvaziparalelního provádění procesů, kdy pouze jediný z procesů může být aktivním, je ukázkou výlučného přístupu procesů k zařízení — reálnému procesoru. Deklarace zařízení má tvar:

[2] Zařízení jsou navržena tak, aby se co nejvíce podobala zařízením v jazyce SOL


```
Facility <identifikátor> ("<jméno zařízení>");
```

Příklad deklarace:

```
Facility Fac("Fac");
```

Zařízení je buď obsazeno některým objektem modelu nebo je volné. Stav zařízení je možné testovat metodou–predikátem `Busy`, která vrací nenulovou hodnotu (TRUE) v případě, že zařízení je obsazeno:

```
if(Fac.Busy()) Print("obsazeno\n");
```

Základní operace zařízení jsou obsazení (metoda `Seize`) a uvolnění (metoda `Release`). Třída Process má definovány také metody Seize a Release, jejich parametrem je zařízení, se kterým se pracuje:

```
Seize(<identifikátor zařízení>[,<výraz>]);
Release(<identifikátor zařízení>);
```

Každé zařízení má vstupní frontu pro požadavky na obsazení, které nelze okamžitě uspokojit. V případě, že není uveden výraz jako druhý parametr Seize, je sémantika této operace jednoduchá: je-li zařízení volné, pak se obsadí, je-li zařízení obsazeno, pak se proces pozastaví a zařadí do vstupní fronty u zařízení. Uvolnění zařízení může provést pouze ten proces, který je obsadil. V případě, že vstupní fronta je neprázdná, zařízení je uvolnění znovu obsazeno prvním objektem (procesem) z fronty.

Poznámka: Současná implementace zařízení není aktivní, tj. vstoupí-li nějaký proces přímo do fronty a zařízení je volné, nedojde k jeho obsazení (k tomu dochází jen při provádění operace Seize, resp. Release).

Příklad:

```
Seize(Fac);
Wait(20);
Release(Fac);
```

V případě uvedení druhého parametru Seize tento parametr znamená *prioritu obsluhy*. Rozsah priority obsluhy je od nuly do 255. Předchozí varianta příkazu Seize je ekvivalentní příkazu Seize s nulovou prioritou obsluhy.

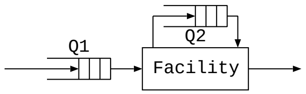
Je-li zařízení Z obsazeno procesem p_1 s prioritou obsluhy o_1 a požaduje-li obsluhu zařízením Z další proces p_2 s prioritou obsluhy o_2, pak mohou nastat dva případy:

$o_1 < o_2$ způsobí přerušení obsluhy procesu p_1 a zařízení je přiděleno procesu p_2. Po skončení pokračuje obsluha p_1, pokud nedošlo k dalšímu přerušení.

$o_1 >= o_2$ proces p_2 se zařadí do vstupní fronty zařízení Z

Z toho plyne existence další fronty u zařízení — fronty přerušených procesů. Obě fronty jsou řazeny podle těchto kritérií:

1. priority obsluhy



Obrázek 2: Zařízení

2. priority procesu
3. FIFO

Každé zařízení automaticky uchovává statistiky, potřebné k výpočtu průměrného využití. Výstup statistik zařízení lze provést metodou `Output`. Do standardního výstupního souboru se tisknou tyto informace:

- využití zařízení (je v rozsahu od nuly do jedné)
- maximální délka vstupní fronty
- průměrná délka vstupní fronty
- průměrná doba čekání ve frontě

5.5 Sklad

Na rozdíl od zařízení, pro které je charakteristický výlučný přístup, umožňuje sklad (objekt třídy `Store`) popisovat simultánní přístup ke zdroji s určitou kapacitou. Jako příklad skladu můžeme uvést parkoviště nebo paměť počítače. Sklad může obsadit více procesů v závislosti na kapacitě skladu a na požadavcích těchto procesů. Proces, který požaduje méně jednotek kapacity než je volné místo, může obsadit požadovanou část kapacity a volné místo se tím zmenší. Pokud proces požaduje více, než je volná kapacita, musí čekat až bude požadované místo volné. Zařízení lze tedy považovat za sklad s kapacitou jedna s tou výjimkou, že sklad nemá možnost přerušovat obsluhu. Deklarace skladu má tvar:

```
Store <identifikátor> ( "<jméno skladu>", <výraz-kapacita> );
```

Příklad:

```
Store Sto("Sto",100);
```

Sklad má metody pro zjištění volné kapacity (`Free`) a predikáty pro testování, je-li prázdný (`Empty`) nebo plný (`Full`). Procesy obsazují sklad operacemi `Enter` a `Leave`.

```
Enter(<identifikátor skladu>, <výraz>);
Leave(<identifikátor skladu>, <výraz>);
```

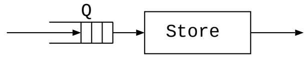
Výraz udává obsazovanou, resp. uvolňovanou kapacitu skladu. Je chybou, když požadovaná kapacita je větší, než deklarovaná kapacita skladu. Příklad ukazuje obsazení a uvolnění deseti jednotek kapacity skladu `S`:


```
Enter(S,10);
Wait(10);
Leave(S,10);
```

Příkaz `Enter` může způsobit čekání procesu na volnou kapacitu. Čekající procesy se řadí do fronty podle priorit, první je proces s nejvyšší prioritou. Příkaz `Leave` uvolňuje zadanou kapacitu a v případě neprázdné vstupní fronty obsazuje sklad první objekt z fronty[3].

Sklad automaticky uchovává statistiky, potřebné k výpočtu průměrného využití. Výstup statistik skladu lze provést metodou `Output`. Do standardního výstupního souboru se tisknou tyto informace:

- deklarovaná kapacita
- maximální použitá kapacita
- průměrná použitá kapacita
- maximální délka vstupní fronty
- průměrná délka vstupní fronty
- průměrná doba čekání ve frontě



Obrázek 3: Sklad

5.6 Sběr statistik

Typickou činností při simulaci diskrétních stochastických systémů je sběr a vyhodnocování statistických informací, získaných z modelu. K tomu jsou určeny standardní třídy `Histogram`, `Stat`, `TStat`.

5.6.1 Histogramy

Objekty třídy `Histogram` slouží k záznamu četností zapisovaných hodnot v zadaných intervalech. Deklarace objektu této třídy má tvar:

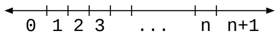
```
Histogram <identifikátor>("<jméno>",<od>,<krok>,<kolik>);
```

Význam jednotlivých parametrů je patrný z obrázku 4.
Zápis do histogramu se provádí příkazem:

```
<identifikátor histogramu>(<výraz-hodnota>);
```

Výstup histogramu do standardního výstupního souboru provede metoda `Output`. Tiskne tabulku četností a statistiku vstupních hodnot (tj. minimální vstupní hodnotu, maximální vstupní hodnotu, počet vstupních hodnot, průměrnou hodnotu a směrodatnou odchylku).

[3] Současná implementace se chová jinak. Podívejte se do zdrojového textu modulu 'store.cc'



Obrázek 4: Parametry histogramu

Příklad:

Histogram `H` sledující četnost hodnot ve 100 intervalech od nuly s krokem 0.1 :

```
Histogram  H("Histogram1",0,0.1,100);
double  x;
  ...
  H(x);           // záznam hodnoty proměnné x
  ...
  H.Output();     // výstup histogramu
```

5.6.2 Statistiky

Objekty třídy `Stat` uchovávají tyto hodnoty:

- součet vstupních hodnot
- součet čtverců vstupních hodnot
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet zaznamenaných hodnot

Metoda `Output` vytiskne tyto hodnoty a navíc průměrnou hodnotu a směrodatnou odchylku.

Příklad:

Testujeme střední hodnotu generátoru exponenciálního rozložení:

```
Stat  TestStat("St1");
  ...
  for(int i=0; i<10000; i++)
    TestStat(Exponential(10));  // záznam hodnoty
  TestStat.Output();            // tisk statistiky
```

5.6.3 Časové statistiky

Objekty třídy `TStat` sledují časový průběh vstupní veličiny. Používají se k výpočtu průměrné hodnoty vstupu za určitý časový interval. Objekty třídy `TStat` uchovávají tyto hodnoty:

- sumu součinů vstupní hodnoty a časového intervalu

- sumu součinů čtverce vstupní hodnoty a časového intervalu
- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet vstupních hodnot
- počáteční čas

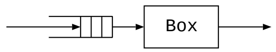
Metoda `Output` tiskne kromě uložených hodnot také průměrnou hodnotu vstupu za čas od inicializace statistiky metodou `Clear` do okamžiku volání metody `Output`.

Příklad:

```
TStat  S("TStat1");
  ...
  S(x);        // záznam hodnoty v čase Time
  ...
  S.Output();  // výstup statistiky
```

5.7 Příklad diskrétního modelu

Uvažujme jedno obslužné středisko se vstupní frontou, kterým procházejí zákazníci:



Obrázek 5: Obslužné středisko Box

```
//  model  MODEL1

#include "simlib.h"

//  deklarace  globálních  objektů
Facility  Box("Linka");
Histogram Tabulka("Tabulka",0,50,10);

class Zakaznik : public Process { // třída zákazníků
  double Prichod;                 // atribut každého zákazníka
  void Behavior() {               // popis chování zákazníka
    Prichod = Time;               // čas příchodu zákazníka
    Seize(Box);                   // obsazení zařízení Box
    Wait(10);                     // obsluha
    Release(Box);                 // uvolnění
    Tabulka(Time-Prichod);        // doba obsluhy a čekání
  }
};

class Generator : public Event {  // generátor zákazníků
```

```
  void Behavior() {               // popis chování generátoru
    new Zakaznik->Activate();     // nový zákazník v čase Time
    Activate(Time+Exponential(1e3/150)); // interval mezi příchody
  }
};

//  popis  experimentu
int main()
{
  Print("***** MODEL1 *****\n");
  Init(0,1000);              // inicializace experimentu
  new Generator->Activate(); // generátor zákazníků, aktivace
  Run();                     // simulace
  Box.Output();              // tisk výsledků
  Tabulka.Output();
  return 0;
}
```

Na začátku popisu modelu musíme použít příkaz `#include`, který definuje rozhraní simulační knihovny. Dále následují deklarace globálních objektů modelu, v tomto příkladu je deklarováno zařízení `Box` a histogram `Tabulka`.

Následuje definice třídy zákazníků, kteří mají chování popsáno v metodě `Behavior`. Každý zákazník má atribut `Prichod`, kterým je doba jeho příchodu do modelovaného systému. Zákazník obsadí zařízení `Box` na dobu 10 časových jednotek (není důležité, jde-li o hodiny či sekundy) a potom zařízení uvolní. Je zajištěno, že v případě již obsazeného zařízení bude zákazník čekat ve frontě, která se vytvoří u zařízení.

Po uvolnění zařízení se do histogramu `Tabulka` poznamená doba, strávená zákazníkem v systému (doba obsluhy plus doba strávená čekáním ve frontě u zařízení). Potom zákazník opouští námi sledovaný systém, a proto je po ukončení procesu automaticky zrušen.

Vytváření zákazníků je realizováno objektem třídy `Generator`, který periodicky se opakující událostí modeluje příchody zákazníků do systému tak, že vytváří nové zákazníky a aktivuje je.

Popis experimentu je součástí funkce `main`. Je inicializován model a nastavena doba simulace funkcí `Init` od času nula do 1000. Potom je zajištěno vytvoření generátoru příchodů zákazníků do modelu. Po inicializaci spustíme vlastní simulaci voláním funkce `Run`. Po ukončení experimentu se vytisknou informace, získané v histogramu `Tabulka`.

6 Spojitá simulace

Spojité chování modelu popisujeme v SIMLIB propojením objektů, které reprezentují integrátory, stavové bloky a různé nelinearity. Propojení objektů se realizuje při jejich vytváření. Konstruktor dostává jako první parametr odkaz na vstupní objekt. Tento odkaz se používá při vyhodnocování objektu. Každý objekt má definovanou metodu Value, která vrací hodnotu objektu. Pokud je k výpočtu hodnoty objektu zapotřebí hodnota vstupu, je objekt na vstupu požádán o svou hodnotu opět metodou Value. Takto proběhne výpočet všech potřebných hodnot objektů. V případě, že vznikne cyklický odkaz (rychlá smyčka), může být detekována.

Pro zvýšení efektivity výpočtu si některé objekty po vyhodnocení pamatují výslednou hodnotu, aby se při několika požadavcích na vyhodnocení v témže modelovém čase nemusely opakovat tytéž výpočty.

Reprezentace výrazu, který je obvykle na vstupu bloku, se dynamicky vytváří při volání konstruktoru. Této vlastnosti bylo dosaženo přetížením obvyklých aritmetických operátorů tak, aby při operandech typu blok dynamicky vytvořily odpovídající grafovou strukturu výrazu.

Příklad:

Uvažujme kmitavý článek:

```
class TEST : public aContiBlock {
  Integrator i1;
  Integrator i2;
public:
  TEST() : i1(-i2,1), i2(i1*0.5) {}
  double Value() { return i2.Value(); }
};
```

6.1 Standardní třídy pro spojitou simulaci

Třída `aContiBlock` definuje obecný blok spojitého modelu s operací vyhodnocení metodou `Value`. Všechny třídy spojitých bloků jsou odvozeny z této třídy a definují metodu `Value` pro vyhodnocení, metodu `Init` pro inicializaci a konstruktor pro definici propojení objektů. SIMLIB obsahuje třídy pro základní aritmetické operace ($+,-,*,/$) a některé funkce (Abs,Sin,Log,...). Uživatel SIMLIB může též definovat libovolné vlastní bloky, které jsou rovnocenné standardním blokům.

6.1.1 Třída Integrator

Třída `Integrator` slouží k implementaci integračního mechanismu spojité simulace. Integrátor má definovány tři základní operace:

- numerickou integraci
- nastavení počáteční podmínky (inicializaci)
- nastavení hodnoty (skokovou změnu stavu)

Numerická integrace vstupní hodnoty je nejdůležitější operací integrátoru, provádí se automaticky v průběhu simulace. Nastavení počáteční hodnoty lze provést více způsoby. Zadání druhého parametru konstruktoru objektu je nejčastější případ.

```
 Integrator <identifikátor>(<obj-výraz>,<číselný výraz>);
```

Při inicializaci modelu v inicializační části popisu experimentu je použitelná metoda `Init`.

```
 <identifikátor>.Init(<číselný výraz>);
```

Zadanou počáteční hodnotu si integrátor pamatuje a nastaví ji při startu simulace ve funkci `Run` automaticky. Nastavení hodnoty integrátoru je proveditelné při běhu simulace buď přiřazovacím příkazem, nebo metodou `Set`.

```
<identifikátor> = <číselný výraz>;
<identifikátor>.Set(<číselný výraz>);
```

Konstruktor má jako první parametr odkaz na objektový výraz — vstup. Lze zadat volitelný druhý parametr s počáteční hodnotou integrátoru. Deklarace

```
Integrator <identifikátor integrátoru>(<objekt-výraz>);
```

vytvoří integrátor se vstupem zadaným objektovým výrazem s implicitně nulovou počáteční hodnotou. Získání hodnoty integrátoru provedeme voláním metody Value příslušného integrátoru:

```
x = <identifikátor integrátoru>.Value();
```

6.1.2 Nelinearity

Pro modelování nelineárních bloků jsou v SIMLIB definovány standardní třídy, které jsou podrobněji popsány v referenční příručce. Nelineární bloky je nutné deklarovat, formát deklarace má tvar:

```
<třída bloku> <identifikátor>(<vstup>,<parametry>);
```

Typickým příkladem je blok omezení (třída Lim):

```
Lim  omez(x+y,-1,1);
```

Výstupem objektu omez je součet hodnot objektů x a y omezený na interval mezi -1 a 1. Blok omez je použitelný jako vstup jiného objektu.

6.1.3 Stavové bloky

Třída Status

Třída `Status` popisuje vlastnosti stavových proměnných. Je to bázová třída pro všechny třídy objektů s vnitřním stavem, kromě třídy `Integrator`. Jako příklad si uvedeme pouze relé a hysterezi, ostatní nelineární bloky jsou stručně popsány v referenční příručce a jejich použití je stejné jako u uvedených objektů.

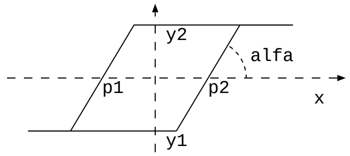
Třída Hyst

Objekty třídy `Hyst` mají charakteristiku podle obrázku 6.

Konstruktor

```
Hyst(Input x, double p1, double p2,
              double y1, double y2, double tga);
```

vytvoří objekt se zadaným vstupem x a s příslušnými parametry (viz obrázek 6). Parametr `tga` je tangentou úhlu alfa. Hodnotu výstupu tohoto objektu získáme voláním metody `Value`.



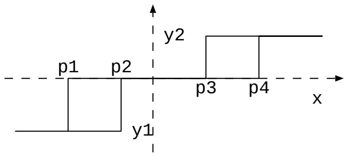
Obrázek 6: Charakteristika hystereze

Příklad:

```
Hyst H(x,-1,1,-5,5,3.5);
```

Třída Relay

Třída `Relay` popisuje objekty s reléovou charakteristikou:



Obrázek 7: Charakteristika relé

Konstruktor

```
Relay(Input i, double p1, double p2, double p3, double p4,
               double y1, double y2);
```

vytvoří objekt se zadanými parametry.

Poznámka:

U objektů s nespojitými charakteristikami (relé, hystereze,...) nastává při simulaci problém 'dokročení' numerické integrace na zlom charakteristiky (například do okamžiku, kdy relé sepne). Pro řešení tohoto problému[4] se používá metod, uvedených v odstavci o kombinované simulaci.

6.2 Příklad spojitého modelu

Jako příklad spojité simulace budeme uvažovat systém kola automobilu. Experiment bude sledovat odezvu kola na jednotkový skok. Rovnice, popisující tlumené kmitání kola:

$$M\ddot{x} + D\dot{x} + kx = F(t)$$

kde

[4] třída Realy v SIMLIB skutečně umí řešit tento problém

v je rychlost pohybu kola,
y je výchylka kola z klidové polohy,
$F(t)$ je vstupní budicí funkce (např. jednotkový skok) a
k, D, M jsou konstantní parametry systému kola
Rovnici převedeme na soustavu diferenciálních rovnic prvního řádu

$$\dot{v} = (F - Dv - ky)/M$$
$$\dot{y} = v$$

a potom můžeme přímo psát simulační program:

```
// model KOLO.CPP - Model tlumení kola automobilu v C++

#include "simlib.h"

const double F = 1.0;

class Kolo {
  Graph G;               // výstup polohy kola
  Integrator v,y;        // stav systému kola
public:
  Kolo(Input F, double M, double D, double k):
     G("Výchylka",y,0.01),
     v((F-D*v-k*y)/M),
     y(v) {}
};

Kolo k1(F, 2, 5.656, 400);

int main() {                   // popis experimentu
  Print(" Model tlumení kola automobilu v C++ \n");
  OpenOutputFile("kolo.out");
  Init(0,1.5);                 // inicializace parametrů experimentu
  SetStep(1e-3,0.1);           // krok integrace
  SetAccuracy(0.001);          // max. povolená rel. chyba integrace
  Run();                       // simulace
  Print(" Konec simulace \n");
  return 0;
}
```

Spojitý model popisujeme propojením funkčních bloků — objektů modelu. Každý objekt je inicializován tak, že prvním parametrem jeho konstruktoru je jeho vstup. Na místě vstupu může být výraz, ve kterém lze použít objekty (jako proměnné) nebo číselné hodnoty (jako konstanty).

Výstup informací o chování modelu probíhá prostřednictvím objektu třídy Graph. Tato třída zabezpečuje rovnoměrné vzorkování vstupu objektu (v našem případě s periodou 0.01) a výstup do výstupního souboru, který lze prohlížet výstupním editorem.

Celý model je tvořen jedním objektem třídy `Kolo` (globální objekt `k1`). Při vytváření objektu musíme zadat jeho vstup a parametry. Řízení experimentu ve funkci `main` zajišťuje otevření výstupního souboru, inicializaci pro modelový čas od nuly do 1.5 sekundy, nastavení povoleného rozsahu kroku numerické integrace (`SetStep`) a nastavení požadované přesnosti numerické integrace (`SetAccuracy`). Vlastní simulace proběhne v rámci volání funkce `Run`.

7 Kombinovaná simulace

Kombinovaná simulace předpokládá použití spojitého i diskrétního přístupu, přináší však navíc některé problémy, spojené se vzájemnou interakcí spojité a diskrétní části modelu. Změny, způsobené diskrétními událostmi ve spojité části modelu, nepřinášejí téměř žádné problémy. Po skokové změně stavu spojité části modelu je zapotřebí pouze znovu inicializovat integrační metodu a je možné pokračovat v simulaci. Složitější situace nastává při potřebě vyvolat diskrétní událost při dosažení určitého stavu spojité části modelu. Řešení tohoto problému je náplní dalších odstavců.

7.1 Stavové podmínky a stavové události

Reakce na změny ve spojité části modelu jsou popsány formou stavových podmínek a stavových událostí. Stavová podmínka může být specifikována například booleovským výrazem. Akci, která je podmíněna změnou pravdivostní hodnoty stavové podmínky, nazveme stavová událost. Příkladem může být událost, která má nastat při překročení nastavené maximální teploty místnosti v modelu automaticky řízeného vytápění domu.

V C++ lze stavové podmínky implementovat třídami, které definují chování objektů - bloků citlivých na změnu vstupu. Vstupem takového bloku může být booleovský výraz. Pro zjištění času změny vstupní hodnoty takových podmínek lze proto použít pouze metodu půlení intervalu, tj. zkracování kroku integrace na polovinu. Existují i jiné metody, například metoda Regula-Falsi, ty však vyžadují spojitý vstup podmínky.

Můžeme také požadovat, aby stavová podmínka byla citlivá pouze na některé změny pravdivostní hodnoty vstupu (například pouze na změnu FALSE na TRUE, případně na překročení mezní hodnoty směrem nahoru). Protože C++ nedefinuje speciální Booleovský typ, používá se místo něj typ int. Pravdivostní hodnota TRUE potom odpovídá nenulové hodnotě typu int, pravdivostní hodnota FALSE odpovídá nulové hodnotě.

Při numerické integraci, kdy výpočet probíhá po krocích, nemusí dojít k detekci některých stavových událostí. Tato situace nastane v případě, že krok integrace je příliš dlouhý a dojde při něm k 'překročení' několika změn stavových podmínek. Podobný problém může nastat v důsledku nepřesnosti numerické integrace, kdy při nevhodně zvolené podmínce nemusí dojít k její změně a tím k vyvolání požadované události.

7.1.1 Třída BoolCondition

Bázová třída BoolCondition popisuje pouze chování, potřebné pro detekci změn vstupní podmínky. Uživatel musí tuto třídu zdědit a doplnit metodu, popisující vstup podmínky a metodu, popisující akci při změně podmínky. Vstup podmínky popisuje metoda Test, která vrací pravdivostní hodnotu. Pro detekci změny pravdivostní hodnoty používáme metodu půlení intervalu. Při změně stavu podmínky se integrační krok zkracuje tak dlouho, až dosáhne své minimální povolené hodnoty (proměnná MinStep). Tím zajistíme 'dokročení' v čase, který se nejvíce blíží skutečnému okamžiku změny podmínky. Teprve potom nastane reakce na tuto změnu vyvoláním metody Action. Metoda Mode umožňuje nastavení citlivosti podmínky na různé změny pravdivostní hodnoty vstupu podmínky.

Příklad:

```
class MyCondition : BoolCondition {
    Input inp;
    int Test()    { return inp.Value()<0; }
    void Action() { Print("změna znaménka z + na -"); }
  public:
    MyCondition(Input i) : inp(i) { Mode(DetectUP); }
};

Integrator x(vstup);
MyCondition  Test(x);
```

Příklad popisuje podmínku, reagující na změnu znaménka vstupní hodnoty z plus na minus. Na vstupu objektu Test je integrátor x.

7.2 Příklad kombinovaného modelu

Jako příklad kombinovaného modelu použijeme model skákajícího míčku. Spojité chování míčku odpovídá volnému pádu, pro detekci dopadu míčku používáme stavovou podmínku. Stavová událost (dopad míčku) vyvolá skokovou změnu rychlosti míčku.

```
//  model  MICEK.CPP -  skákající  míček

#include "simlib.h"

const double g = 9.81;       // gravitační zrychlení

class Micek : BoolCondition {
  Graph G;
  Integrator v,y;
  int Test() { return y.Value()<0; }
  void Action() {
      Print(" Odraz míčku v čase t=%g \n",Time);
      G();
      v = -0.8 * v.Value();
      y = 0;  // nutné pro detekci následujícího dopadu
    }
  }
public:
  Micek(double position) :
     G("Výška míčku",y,0.05),
     y(v,position), v(-g)  {
    Mode(DetectUP);
  }
};

Micek  micek(1);             // vytvoření objektu micek
```



```
int main() {                    // popis experimentu
  Print(" Model skákajícího míčku v C++ \n");
  OpenOutputFile("micek.out");
  Init(0,5);                    // inicializace experimentu
  SetStep(1e-10,0.5);           // krok integrace
  SetAccuracy(1e-5,0.001);      // max. povolená chyba integrace
  Run();                        // simulace
  Print(" Konec simulace \n");
  return 0;
}
```

Chování míčku je popsáno jako volný pád. Integrátor v integruje tíhové zrychlení g a jeho hodnota je rovna rychlosti míčku. Integrátor y integruje rychlost a jeho hodnota představuje výšku míčku nad zemí.

Pro detekci okamžiku dopadu míčku na zem ($y = 0$) je použito stavové podmínky `y.Value()<0` v metodě `Test`. Tato metoda je volána systémem řízení simulace při každém kroku numerické integrace. V případě změny hodnoty podmínky se krok zkracuje tak, abychom okamžik dopadu míčku určili s maximální přesností (přesnost určení doby dopadu je dána minimální délkou kroku, tj. hodnotou proměnné `MinStep`). V okamžiku dopadu míčku se provede akce, popsaná v metodě Action, tj. obrácení a zmenšení vektoru rychlosti. Tímto způsobem modelujeme ztrátu energie při dopadu míčku.

Řízení experimentu popisuje funkce `main`. Po volání Init následuje nastavení povoleného rozsahu kroku integrace (SetStep) a určení požadované přesnosti integrace (SetAccuracy). Běh simulace se odstartuje voláním funkce Run. Výstup se řeší podobně, jako u spojité simulace, je zde však nutné zajistit výstup též v okamžicích dopadu míčku na zem.

8 SIMLIB-3D rozšíření

Pro usnadnění popisu modelů, které vyžadují popis vektorovými diferenciálními rovnicemi byla SIMLIB doplněna o 3D abstrakce.

Prostorové modely jsou popsatelné vektorovými dif. rovnicemi. Například pohyb hmotného bodu v gravitačním poli je možné popsat rovnicí:

$$\ddot{\vec{r}} = \frac{\vec{F}}{m}$$

kde: $\vec{r}$ je vektor — pozice bodu
$\vec{F}$ je vektor síly působící na hmotný bod
m je hmotnost bodu

SIMLIB dovoluje popis tohoto systému jak skalárními prostředky, tak vektorově.

Vektorový popis je výrazně kratší:

```
class MassPoint {  // model hmotného bodu
  const double m;  // hmotnost
  Integrator3D v;  // rychlost
  Integrator3D r;  // pozice
 public:
  MassPoint(Input3D F, Value3D ini_pos) :  // konstruktor:
     m(1000),       // hmotnost je konstantní
```

```
    v(F/m),         // rychlost je integrál zrychlení
    r(v, ini_pos)   // pozice je integrál rychlosti
    {}
};
```

Z uvedeného příkladu je zřejmé, že třírozměrný popis problému je stejně jednoduchý jako popis skalární (operace jsou vektorové a v roli operandů jsou vektory místo skalárů).

8.1 Hierarchie tříd 3D

```
aBlock
  aContiBlock3D      - bázová třída
    Constant3D       - konstanta
    Expression3D     - blokový výraz
    Integrator3D     - vektorový integrátor
    Function3D       - obecná vektorová funkce
       _Add3D, _Mul3D - (skryté) třídy pro operátory
    Parameter3D      - parametr modelu

Value3D              - hodnota
Input3D              - odkaz na blok
```

Třída `Value3D` definuje vektorovou hodnotu a má tři složky (x,y,z) typu `double`. Používá se na předávání a uchovávání vektorových hodnot.

Třída `Input3D` definuje odkaz na objekt–blok. Použití odkazu v objektovém výrazu je transparentní, tj. blok na jehož vstupu je uveden tento odkaz si poznamená cíl tohoto odkazu a nikoli odkaz samotný (pozdější změny odkazu již nic neovlivní). V případě, že nám toto chování nevyhovuje, je možné použít třídu `Expression3D`, která se chová jako blok–identita.

Třída Integrator3D obsahuje tři skalární integrátory, které jsou napojeny na speciální objekty pro transformaci rozhraní 3D/skalární. To je možné proto, že integrace je lineární operátor.

Ostatní třídy definují konstanty, parametry modelu a funkce podobně jako jejich skalární ekvivalenty.

8.2 Blokové výrazy

Operace $+ - */$ jsou implementovány jako operátory, které dynamicky vytvoří příslušný objekt a zapojí jeho vstupy. Situace je stejná jako u běžných skalárních operátorů.

Přehled definovaných operátorů:

```
// binární operátory:
Input3D operator + (Input3D a, Input3D b);  // součet vektorů
Input3D operator - (Input3D a, Input3D b);  // rozdíl
Input3D operator * (Input3D a, Input3D b);  // součin
Input3D operator * (Input3D a, Input b);    // vektor * skalár
Input3D operator * (Input a, Input3D b);    // skalár * vektor
Input3D operator / (Input3D a, Input b);    // vektor / skalár
```



```
// unární operátory:
Input3D operator - (Input3D a);             // unární -
```

Funkce v blokových výrazech 3D jsou definovány pouze pro základní operace s vektory:

```
// funkce:
Input Abs(Input3D x);                       // absolutní hodnota vektoru
Input3D UnitVector(Input3D x);              // jednotkový vektor
Input ScalarMultiply(Input3D x, Input3D y); // skalární doučin x.y

Input Xpart(Input3D a); // složka x vektoru
Input Ypart(Input3D a); // složka y vektoru
Input Ypart(Input3D a); // složka z vektoru
```

8.3 Příklad

Uvažujme systém Země–Měsíc a družici, která má vhodně zvolenou počáteční pozici a rychlost. Takový systém můžeme popsat takto:

```
// druzice.cc -- model Země--Měsíc--družice
#include "simlib.h"
#include "simlib3D.h"

typedef Value3D Position, Speed, Force;

const double   gravity_constant = 6.67e-11; // gravitační konstanta
const double   m0 = 1000;           // hmotnost družice
const double   m1 = 5.983e24;       // hmotnost Země
const double   m2 = 7.374e22;       // hmotnost Měsíce
const Position p0(36.0e6, 0, 0);    // poloha družice
const Position p2(384.405e6, 0, 0); // poloha Měsíce
const Speed    v0(0, 4.5e3, 0);     // počáteční rychlost družice
const Speed    v2(0, 1022.6, 0);    // oběžná rychlost Měsíce

Constant3D Zero(0,0,0); // pomocný objekt

struct MassPoint {
  double m;              // hmotnost
  Expression3D inforce;  // vstupní síla
  Integrator3D v;        // rychlost
  Integrator3D p;        // pozice
  MassPoint(const double mass, Position p0, Speed v0=Speed(0,0,0)) :
    m(mass),
    inforce(Zero),
    v(inforce/m, v0),
    p(v,p0) {}
   void SetInput(Input3D i) { inforce.SetInput(i); }
};

struct MyWorld {  // digitální svět
  enum { MAX=10 };
  MassPoint *m[MAX];
  unsigned n;
```

```
  MyWorld();
};

MyWorld *w;  // vznikne až později :-)

// gravitační síla pusobící na hmotný bod p
class GravityForce : public aContiBlock3D {
  MassPoint *p;
  MyWorld *w;
 public:
  GravityForce(MassPoint *_p, MyWorld *_w) : p(_p), w(_w) {}
  Force Value() {
    Force f(0,0,0);  // gravitační síla
    for(int i=0; i < w->n; i++) {
      MassPoint *m = w->m[i];
      if (m == p) continue;
      Value3D distance = m->p.Value() - p->p.Value();
      double d = abs(distance);  // vzdálenost
      f = f + (distance * gravity_constant * p->m * m->m / (d*d*d)) ;
    }
    return f; // součet všech gravitačních sil
  }
};

typedef MassPoint Planet, Satelite; // pohyblivé planety

MyWorld::MyWorld() {  // --- vytvoříme digitální svět
  n=0;
  m[n++] = new Planet(m1,Position(0,0,0),Speed(0,0,0));
  m[n++] = new Planet(m2,p2,v2);
  m[n++] = new Satelite(m0,p0,v0);
  for(int i=0; i<n; i++)  // --- zapneme silové působení
    m[i]->SetInput(new GravityForce(m[i],this));
}
```

Třída `MassPoint` definuje model hmotného bodu. Vstupem tohoto modelu je síla, která způsobí jeho pohyb. Protože model vytváříme postupně, je implicitně tato síla nulová a nastaví se až po vytvoření všech hmotných bodů v systému (viz konstruktor třídy `MyWorld`). Celý model je tvořen třídou `MyWorld`, která obsahuje seznam všech hmotných bodů.

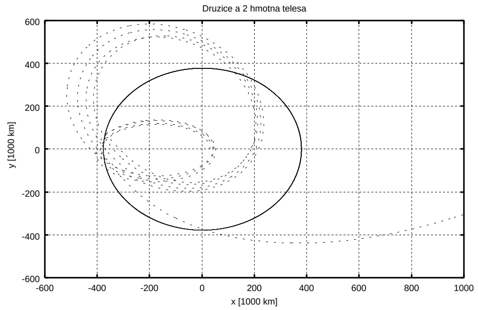
Třída `GravityForce` modeluje gravitační sílu a je pro každý hmotný bod definována jako součet všech gravitačních sil působících na hmotný bod. Gravitační síla se počítá se podle gravitačního zákona.

Výsledná dráha družice a Měsíce je uvedena na obrázku 8.

Reference

[1] Rábová,Z., Zendulka,J., Češka,M., Peringer,P., Janoušek,V : *Modelování a simulace*, Nakladatelství VUT Brno, 1992, 226 stran

[2] Fishwick P.: *Simulation Model Design and Execution: building digital worlds*, Prentice Hall, 1995



Obrázek 8: Družice v gravitačním poli

[3] Peringer P.: *Object-Oriented Description of Continuous Systems*, 5th International Symposium on Modeling and System Simulation, June 1-4 1993, Olomouc, 1993

[4] Peringer, P.: *Basic Abstractions for Object-Oriented Model Description*, Proceedings of Conference MOSIS'94, 30.5-2.6 1994, Zábřeh na Moravě, Czech Republic, p. 274-277

[5] Peringer, P.: *Modelování na bázi komunikujících objektů*, Disertační práce FEI VUT Brno, prosinec 1996

[6] Leška D.: *Objektově orientovaný přístup k numerickým metodám*, Diplomová práce FEI VUT Brno, květen 1997

[7] Peringer, P.: *SIMLIB/C++*, dokumentace k simulační knihovně, FEI VUT Brno, prosinec 1997

[8] Peringer, P.:*Hierarchical Modelling Based on Communicating Objects*, Proceedings of Conference MOSIS'97, April 1997, Hradec nad Moravicí, Czech Republic, p. 122-127

[9] Peringer, P.:*Tools for continuous Simulation in 3D space*, Proceedings of ASIS 1997, September, 1997, Krnov, Czech Republic, p.327-330