VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

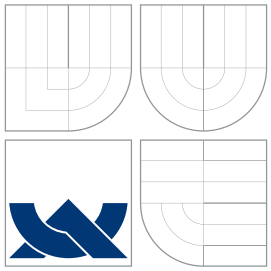
SIMULÁTOR ŽELEZNIČNÍHO STAVĚDLA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

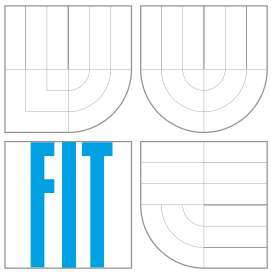
AUTOR PRÁCE
AUTHOR

BEDŘICH HOVORKA

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULÁTOR ŽELEZNIČNÍHO STAVĚDLA
SIMULATOR OF RAILWAY INTERLOCKING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — BEDŘICH HOVORKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. DAVID MARTINEK
SUPERVISOR

BRNO 2007

Zadání

1. Prostudujte funkce a součásti elektronického železničního stavědla používaného pro řízení provozu v železničních stanicích.
2. Prostudujte simulační metody použitelné pro realizaci simulátoru železniční sítě v železniční stanici, zejména diskrétní, spojitou a kombinovanou simulaci. Navrhněte vhodný simulační přístup, či kombinaci přístupů pro realizaci tohoto simulátoru. Navrhněte vhodný formát dat pro reprezentaci modelu železniční sítě.
3. Navrhněte a implementujte vybranou část simulátoru a demonstrujte funkčnost na dostatečně názorném modelu.
4. Diskutujte získané výsledky a další možné směry vývoje.

Kategorie: Modelování a simulace

Implementační jazyk: Java, nebo podle vlastního uvážení

Operační systém: Windows, Linux

Literatura: Podle pokynů vedoucího

Datum zadání: 1. listopadu 2006

Datum odevzdání: 15. května 2007

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Stavědlo je dispečerské zařízení pro řízení dopravy. Dispečer určuje nastavováním výhybek a semaforů cestu vlakům. V této práci se zabývám návrhem a výstavbou základu simulátoru takového zařízení v jazyce Java. Nastudoval jsem funkce tohoto zařízení. Zabýval jsem se strukturou celé aplikace a implementoval chování základních prvků.

Klíčová slova

železniční stavědlo, kombinovaná simulace, XML, Java, OOP

Abstract

Railway interlocking is a dispatching facility for traffic control. The dispatcher controls train paths by setting switches and signals. In this thesis, I describe the design and implementation of a simple simulator of the facility in the Java language. I have studied functions of this facility. I interested with structure of whole application. And I implemented behaviour of foundamental elements.

Keywords

railway interlocking, combined simulation, XML, Java, OOP

Citace

Bedřich Hovorka: Simulátor železničního stavědla, bakalářská práce, Brno, FIT VUT v Brně, 2007

Simulátor železničního stavědla

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Davida Martinka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Bedřich Hovorka
10. května 2007

Obsah

Kapitola 1

Úvod

V této bakalářské práci jsem se zabýval problematikou dopravní simulace. Moje práce spočívala v nastudování součástí systému, experimentováním s existujícími implementacemi her řízení železniční dopravy, nalezením vhodné simulační knihovny a jako hlavní činností – návrhem a implementací základu simulátoru. Nejprve bylo třeba navrhnout strukturu aplikace, vnitřní reprezentací modelu a formát dat. Poté jsem mohl s vnitřní reprezentací implementovat grafické rozhraní umožňující editaci. Dále jsem se už zabýval především simulačním modelem.

V druhé kapitole představím modelovaný systém. Jeho chování je lépe ukázat na již existujících programech.

V třetí kapitole se zmíním o numerickém řešení diferenciálních rovnic. V další sekci narazíte na základní pojmy z modelování a simulace. Letmo se zmíním i o tom co to je diskrétní, spojitá a kombinovaná simulace, u kombinované trochu podrobněji.

Ve čtvrté kapitole popisuji spíše než běžně známé vlastnosti použitých nástrojů, vlastní postřehy a příklady z implementace. To platí hlavně o sekcích o Javě a XML. V sekci o simulační knihovně *jDisco* stručně vysvětlím, jak se používá, aby nedošlo k některým nedorozuměním, co jsem musel dále v implementaci řešit a co ne.

Do páté kapitoly sem vybral, podle mého, důležité myšlenky z návrhu a popisu implementace. Do podrobností zacházím v případě, že mi to připadá nutné. Nejprve se zamýšlím nad celkovou strukturou aplikace. Poté popíšu organizaci prvků v systému. Následováno popisem jak je implementována simulace. A nakonec představím příklad demonstrující, to co bylo naimplementováno.

V závěrečné kapitole projdu některá možná rozšíření, zejména ty které úzce navazují na současnou implementaci. Toto sekci následuje již jen zhodnocení celé práce.

Některé názvosloví bylo použito z [1] a existujících implementací. Také názvy v obrázkách a schématech budou většinou anglicky, aby odpovídaly zdrojovým kódům. Pro pochopení tohoto textu předpokládám, že čtenář je alespoň zběhlejším studentem informatiky a rozumí pojmům z diskrétní a numerické matematiky, algoritmů a objektové orientace.

Příklad odkazu do zdrojových souborů: `Example`.

1.1 Etapy a cíle

Co bylo implementováno před bakalářským projektem

Během Semestrálního projektu vytvořil prototyp objektového modelu sítě, zejména statické vlastnosti. Na základě toho jsem navrhl XML formát a implementoval továrnu pro načítání

a ukládání. V rámci projektu do Modelování a simulace jsem vytvořil kombinovaný model pohybu vlaku podle dané posloupnosti semaforů a hran (ne v síti). Do předmětu Tvorba uživatelských rozhraní jsem odevzdal jednoduchý grafický editor výše zmíněného modelu sítě, který zobrazuje síť v zažitém schématu obdobném tomu použitém v elektronických stavědlech. Vše jsem během bakalářského projektu doplnil do následujících cílů:

Cíle bakalářského projektu

Vytvořit jádro přenositelného programu:

- na základě nastudovaných podkladů navrhnout rozšiřitelný systém tříd, propojitelné komunikující moduly
- Dynamické vlastnosti a chování základních prvků – semafor, výhybka, kolej
- Pohyb a lokalizace vlaků v síti
- Datová struktura pro manipulaci z cestou, zejména stavění cest (bez jejich rušení)
- Vytvořit příklad, který demonstruje funkčnost výše zmíněných cílů

Vlastní modelování dalších prvků a všech detailů systému bych přenechal jako rozšíření.

Kapitola 2

Modelovaný systém a existující implementace

V této kapitole budou stručně popsány funkce modelovaného systému. Dále uvedu některé již existující aplikace. Budou zde zmíněna i fakta, která je třeba brát v úvahu, i když jejich implementace nebyla dokončena.

2.1 Železniční stavědlo

Železniční stavědlo je zařízení, které umožňuje řídit dopravní síť nějakého uzlu či traťového úseku z jednoho místa. Ovládá chování několika prvků. Například nastavuje signály na semaforech a přestavuje výhybky. Některá dnešní stavědla mají počítačové rozhraní a jsou ovládána jedním dispečerem. Ten každému vlaku vymezí určitou cestu, tak že zadá počáteční a cílový semafor. Na stavědlu je pak, aby našlo nejvhodnější neobsazenou cestu.

1. Pokud ji nenajde z důvodu obsazení všech možností uloží požadavek (dvojici – počátek a cíl) do fronty. Při uvolňování kolejí testuje možnost sestavit první prvek ve frontě. Frontu je možné editovat.

2. V opačném případě nastaví prvky v cestě

Toto souvisí i z bezpečností. Systém nesmí na každé koleji dovolit více než jednu vlakovou cestu[1]. Jak může takový simulátor vypadat prozradí následující sekce.

2.2 Existující implementace

Pro lepší představu jak taková aplikace funguje zde popisuji některé již existující simulátory řízení železniční dopravy. Sice se vesměs jedná o hry, ale svojí koncepcí se blíží k trenažéru reálného systému. Zachovávají si však jednoduchost ovládání. Použil jsem je také jako prostředek pro analýzu systému. Experimentováním s těmito programy jako metodou zpětného inženýrství[2] lze nastudovat strukturu a chování většiny systému. Každý má svá národní specifika (detaily v pravidlech řízení), však mnoho principů je společných, které popisuji u prvního. U dalších zmiňuji již jen rozdíly.

[1] Existují i tzv. posunové, ale ty zatím neuvažuji

[2] http://en.wikipedia.org/wiki/Reverse_engineering

Staničář

Tento herní simulátor má modelovat české JOP. Uživatel si zvolí čas a stanici. Stanice a vlaky je uložena v 1 velkém XML souboru. Poté se mu zobrazí kolejiště a seznam vlaků v systému. Nyní může ovládat jednotlivé prvky a sledovat chování systému. Do kolejiště určenými místy vjíždějí vlaky (podle jízdního řádu, generátor s exponenciálním rozložením doby mezi generováními). Úkolem dovést vlaky na místo podle jízdního řádu a udržet plynulost dopravy. Každá stanice či větší systém má svůj zásobník (tak je zde nazývaná FIFO fronta) povelů, kam se uloží když zrovna nemohou být vykonány. U každého vlaku je možnost prohlížet jízdní řád (čas, zastávka, číslo koleje). Stanice je možné vytvářet vlastní v odděleném editoru. Dále je tu i editor jízdních řádů. V současné verzi funguje tato hra pod .NET 2.0 ve Windows. Dřívější verze byly spustitelné v dotGNU.

SimSig

Tento britský simulátor podporuje sice více stanic, ale každá má svůj vlastní instalátor. Funkčně je obsáhlejší. Umí simulovat výpadek elektřiny, náhodné výluky v různých částech sítě. a další poruchy vlaků a síťových prvků. Je tu také určitá míra automatického řízení i v rozvětveném kolejišti (provedení jízdního řádu). Sice určená pro Windows, ale na rozdíl od předcházejícího sem ji spustil i ve wine[3].

Gordikon, Multikon, Brno Síť je nastavena napevno. Oproti ostatním jsou na začátku simulace také vlaky umístěny na kolejích v síti, nejenom, že napřed musí vstoupit do systému zvláštním prvkem v síti, což je reálnější.

[3] wine-0.9.36

Kapitola 3

Teorie

V této kapitole popisuji některé teoretické základy. Pojmy jako *množina* či *relace* předpokládám, že čtenář zná.

3.1 Numerické metody

K výpočtu diferenciálních rovnic na číslicových počítačích bývají nejvhodnější numerické metody, neboť jsou zde operace prováděny diskrétně. Spojité výpočty je nutné na ně převést. A právě tímto převodním prostředkem jsou numerické metody. V závislosti na použití je nezbytné prozkoumat jejich vlastnosti jako je přesnost a stabilita řešení. Touto problematikou se podrobněji zabývá [10]. Zmenšováním kroku se zvyšuje přesnost výpočtu (různě u různých metod), však při velmi malých krocích už hraje roli nepřesnost elementárních operací číslicové aritmetiky.

Jako příklad zde uvádím jednokrokovou metodu řešení ODR z počátečními podmínkami – *Runge-Kutta-Fehlberg* (rovnice 3.1 - 3.7), protože je dále v této práci použita. Je pátého řádu a navíc můžeme mezihodnoty k_n zkombinovat na aproximaci čtvrtého řádu (rovnice 3.8). Z té pak můžeme vypočítat odhad chyby $y_{n+1} - y^*_{n+1}$. Využití těchto vlastností viz sekce 3.2. Na obrázku 3.1 vidíme její použití, když očekáváme výsledek blízký polynomu druhého stupně.

$$k_1 = f(x_n, y_n), \tag{3.1}$$

$$k_2 = f(x_n + \frac{1}{4}h, y_n + \frac{1}{4}hk_1), \tag{3.2}$$

$$k_3 = f(x_n + \frac{3}{8}h, y_n + \frac{1}{32}h(3k_1 + 9k_2)), \tag{3.3}$$

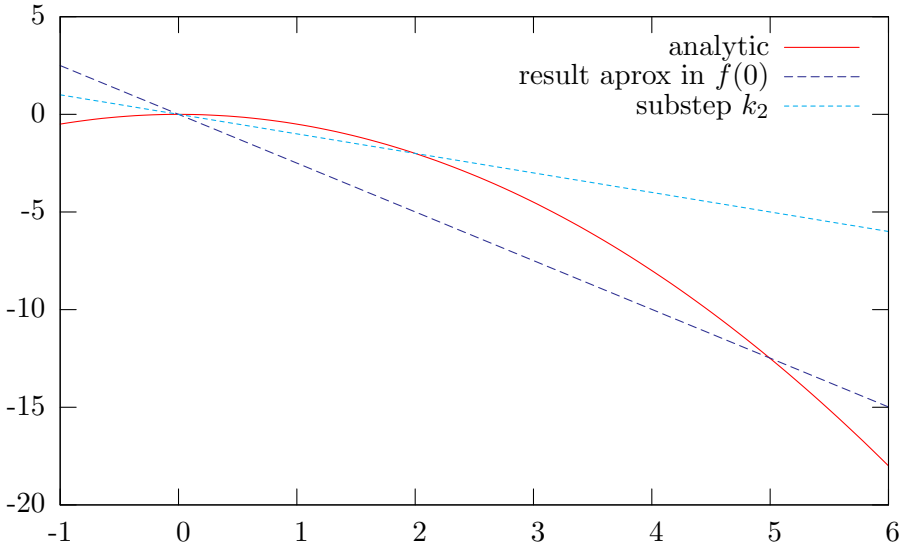
$$k_4 = f(x_n + \frac{12}{13}h, y_n + \frac{1}{2\,197}h(1\,932k_1 - 7\,200k_2 + 7\,296k_3)), \tag{3.4}$$

$$k_5 = f(x_n + h, y_n + h(\frac{439}{216}k_1 - 8k_2 + \frac{3\,680}{513}k_3 - \frac{845}{4\,104}k_4)), \tag{3.5}$$

$$k_6 = f(x_n + \frac{1}{2}h, y_n + h(-\frac{8}{27}k_1 + 2k_2 - \frac{3\,544}{2\,565}k_3 + \frac{1\,859}{4\,104}k_4 - \frac{11}{40}k_5)) \tag{3.6}$$

$$y_{n+1} = y_n + h(\frac{16}{135}k_1 + \frac{6\,656}{12\,852}k_3 + \frac{28\,561}{56\,430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6) \tag{3.7}$$

$$y^*_{n+1} = y_n + h(\frac{25}{216}k_1 + \frac{1\,408}{2\,565}k_3 + \frac{2\,197}{4\,104}k_4 - \frac{1}{5}k_5) \tag{3.8}$$



Obrázek 3.1: Demonstrace kroku metody

3.2 Modelování a simulace

V této sekci uvádím základní pojmy z modelování a simulace. Podrobnosti v [11, 6]

Systém je cokoliv u čeho můžeme popsat jeho chování. Může být reálný i imaginární. Například železniční síť s vlaky a řídícími prvky.

Prvek systému je elementární, dále nedělitelná, část systému. Vzájemná interakce prvků určuje chování celého systému. Například kolejový oddíl. Zde záleží na míře abstrakce.

Model je systém, který napodobuje vlastnosti a chování původního systému.

Modelování je postupné vytváření modelů. Z pozorování a studování systému vznikne *konceptuální model*, což je soubor neurčitých informací (nástin). Z něj vytváříme *abstraktní model* – formální popis systému. A z toho pak izomorfním zobrazením naprogramujeme *simulační model*.

Simulace Získávání nových znalostí o systému experimentováním se simulačním modelem.

Reálný čas skutečný čas, ve kterém běží model

Modelový čas časová osa modelu

Strojový čas čas, který byl potřebný k výpočtu (tj. jak dlouho byl procesor přepnut do kontextu daného procesu)

3.2.1 Diskrétní simulace

Stav všech prvků je definován pouze v diskrétních časových okamžicích. Během tohoto okamžiku je provedena atomická operace zvaná událost, která tento stav může změnit.

K formálnímu popisu se používá často Petriho síť. Simulace je řízena pomocí kalendáře událostí tímto jednoduchým algoritmem:

```
for event in calendar.remove_iterator :
  model_time = event.time
  event.behaviour()
```

3.2.2 Spojitá simulace

Stav každého prvku je definován v každém okamžiku vymezeného časového intervalu. Abstraktním modelem jsou algebraické, diferenční či diferenciální rovnice a jejich soustavy. U jednokrokových metod lze na základě odhadu chyby měnit krok a tím urychlit výpočet. Metoda uvedená v sekci 3.1 má takové vlastnosti.

```
#zneplatnění předcházejících změn (derivace=0)
for variable in all_variables : variable.state_fix()
#výpočet změn stavů z předcházejících
for continous in all_cont_processes : continous.derivatives()
integrate() #krok numerické metody s posunem modelového času
```

3.2.3 Kombinovaná simulace

Systém obsahuje spojité i diskrétní prvky. Řízení simulace musí při výpočtu spojitých stavů měnit krok, aby dokročila k naplánované diskrétní události. Dále musí detekovat stavové podmínky a naplánovat provedení události k podmínce přiřazené. Z numerických metod jsou proto vhodné jednokrokové, protože u nich záleží jen na předchozím výsledku.

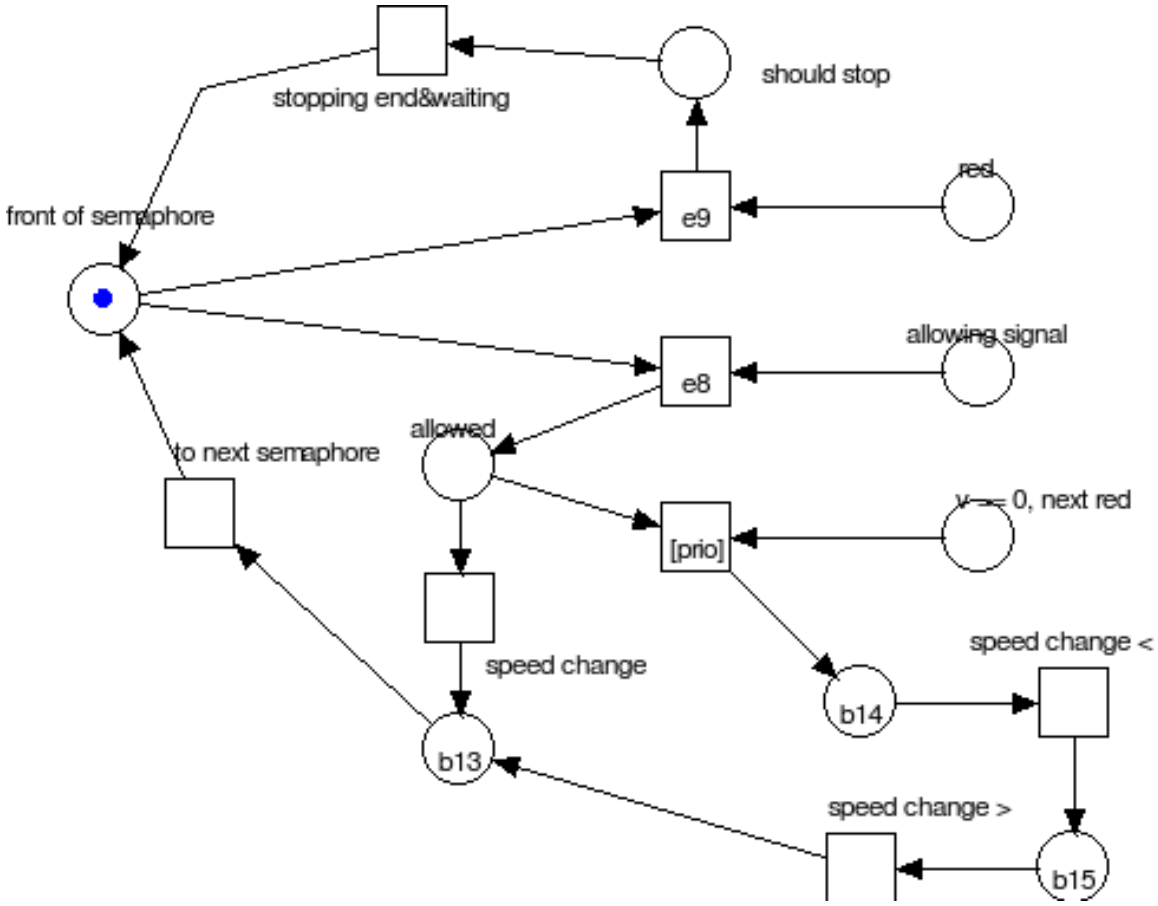
Petriho síť při vytváření kombinovaného modelu

Během návrhu aplikace jsem použil Petriho sítí k přehlednému vyjádření procesů. Takto jsem si lépe představil průběh simulace a jak ho mám vyjádřit v programovacím jazyce. Vesměs se jednalo o C/E sítě, tj. místo, zde zvané podmínka, obsahuje maximálně jednu značku a přechod, zde nazývaný událost, je proveditelný v případě, že všechny podmínky před ním platí a podmínky za ním neplatí. V závislosti na lepším vyjádření jsem přechody prohlásil buď za události reagující na stavovou podmínku, která ruší platnost značky před ní, nebo za spojitý proces a místo za ním je stavová podmínka, na kterou čeká následující událost.

Mimo jiné jsem je také mohl ověřit pomocí nástroje CESim (viz [8]), pokud se síť dala vyjádřit, aniž by to bylo na úkor přehlednosti. Například síť na obrázku 6.1 je tímto nástrojem ověřená.

V síti na obrázku 3.2 se navíc nacházejí externí podmínky a priorita. Proces se vrací do stejného stavu – tj. je před návěstidlem , když vlak dojede k dalšímu návěstidlu nebo změnou signálu skončilo čekání před ním. Podmínky „red“ a „allowing signal“ představují stav tohoto návěstidla a samozřejmě vždy[1] platí právě jedna z nich. Podmínkou „$v \leq 0$, next red“ a prioritou ošetřuji možnost zastavení na konci následujícího bloku, kterému přecházelo rozjetí na jeho začátku.

[1] v každém diskrétním okamžiku stavu procesu



Obrázek 3.2: Petriho síť řízení vlaku podle signálů na semaforech

Kapitola 4

Použité jazyky a nástroje

Tato kapitola popisuje jazyky resp. nástroje, které jsem použil ve své práci. S Javou a XML sem se seznámil již dříve, proto nechci zacházet do podrobností. Uvádím zde hlavně rysy jazyků resp. nástrojů týkající se této práce jako zdůvodnění, proč sem tyto jazyky či nástroje použil a případně pro co byly v mé v mé práci nevhodné.

4.1 XML

XML je standardizovaný značkovací jazyk pro ukládání různých typů dat. Rozšiřuje text o značky, zvané tagy, ty jsou uzavřeny do < >, mají název a mohou mít několik atributů. Tagy jsou počáteční (začínají <), koncové (začínají < /) nebo "obojetné" (počáteční s / před >). Využívá v různých oblastech, jako například webové stránky, vektorová grafika, katastr nemovitostí, serializace objektů (v JavaBeans), a samozřejmě také pro ukládání modelů v M&S (např. Ptolemy II.).

XML definuje obecnou syntaxi, ve které je možné vytvářet vlastní značky. Vnitřní reprezentaci představuje n-ární strom různých typů uzlů jako element, atribut, text atd. Data tedy mohou být rozmístěna hierarchicky podle logických vazeb. Snadno se rozšiřuje a podporován mnoho jazyky, což umožňuje vytváření datových formátů postupně prototypováním. Díky jmenným prostorům lze do dokumentu jednoho typu vložit část jiného (třeba již existujícího). Například do popisu dopravní sítě, jež představuje hlavně topologii, lze přidat ke každé cestě její geometrické vlastnosti pomocí externího vektorového grafického formátu. Pro strojovou výměnu dat je to výborný jazyk, však pro ruční zápis, či dokonce programování (např. XSLT) se mi zdá nevhodný. Často je vhodné jej před odesláním přes síť zkomprimovat, neboť se zde projevuje nevýhoda tohoto typu značkování – opakovaná informace navíc.

4.1.1 Analyzátory XML

Validace dokumentů Lexikální a syntaktická pravidla jsou stejná pro všechna XML. Tvůrce XML aplikace určuje až sémantiku. Částečně mu mohou být nápomocny následující definiční nástroje. Pro určení jak má vypadat tzv. *správně formulovaný dokument* v XML slouží Definice typu dokumentu (DTD), kde se určí metadata – co může dokument obsahovat za tagy, jaké mají atributy a co mohou obsahovat, ale jen na úrovni prvků XML, chybí například typová kontrola atributů ap. Tudíž DTD nestačilo a muselo vzniknout XML Schema [2], což je XML aplikace popisující navíc také objektové vazby mezi značkami.

Pro zvolení XML jako datového formátu hovoří také to, že již existují analyzátory pro různé jazyky Postarají se o lexikální a syntaktickou analýzu podle obecné XML gramatiky a případně provedou validaci. Existují dva hlavní – SAX a DOM. SAX prochází dokumentem a volá metody zaregistrovaného `Handleru`, když narazí na začátek či konec elementu. Během tohoto průchodu může být provedena i validace. Při analýze DOM se použije SAX k převodu textu na standardizovanou vnitřní podobu dokumentu (strom). Proto se používá spíše v programech, které potřebují náhodný přístup k různým úrovním dokumentu, jako jsou editory a prohlížeče. Avšak má-li aplikace vlastní lepší vnitřní reprezentaci dat či je třeba provádět dlouhé sekvenční transformace, je vhodnější SAX. Více o XML v [5]

4.2 Java

Java je obecný objektově-orientovaný programovací jazyk. Jeho hlavní výhodou je, že se spouští v prostředí virtuálního stroje, jehož implementace je dostupná pro mnoho bežných operačních systémů. Takže umožňuje vyvářet přenositelné spustitelné soubory. Vybral jsem jej, že s tímto jazykem mám nejvíc zkušeností. K implementaci jsem nakonec zvolil verzi 6 Standard Edition. Existuje i překladač v GCC je však s nejnovější oficiální verzí nekompatibilní. Navíc od verze 7 by měla být otevřená již i oficiální verze (viz projekt OpenJava). Existuje mnoho publikací – např. [9, 12]. Java od svého vzniku prošla řadou inovací, leccos bylo doplněno a některé knihovní třídy jsou zavržené. Proto některé informace ve starší literatuře jsou zastaralé. Nejspolehlivějším zdrojem je tedy [3]. V Javě jsem zvyklý psát kód tak, že názvy proměnných a metod samy o sobě komentují.

Kontejnery (Kolekce) Jsou obdobou STL z C++ a slouží k ukládání předem nedefinovaného množství dat. V Javě jsou to objekty, tedy sami mohou být prvky jiné kolekce. Například `Map` slouží k vytváření zobrazení objektu na objekt, čímž snižuje vytváření referencí přímo v objektech a tedy i závislost kódu jedné třídy na jinou. Problematika kontejnerů je více rozvedena v [7], avšak při použití Javy 5 je třeba brát zřetel na to, že kontejnery jsou parametrizované. Skládáním kolekcí jsem vytvořil vlastní datové struktury. Především jsem doplnil nebo zcela definoval rozhraní a implementoval neoptimalizované prototypy. Z nich se mohou vyvinout úložiště přímo na míru, však si stále mohou zachovat určitou míru obecnosti. Například `Array2DMap` v rychlosti je srovnatelná s `TreeMap`, ale nejčastěji volanou je právě vybrání položky a provádí se za kritických výpočtů. Zde šlo hlavně o to minimalizovat vytváření klíčových objektů dvojicí celých čísel a přitom zachovat kompatibilitu z `Map`.

Výhody a nevýhody Svou syntaxí více inklinuje k obecným jazykům (C, C++), přesto množstvím standardních knihoven dohání mnohý skriptovací jazyk. Přísná statická typová kontrola může na první pohled zdržovat vývoj, ale při správném návrhu a použití vhodných editorů naopak zrychluje odladění, protože se můžu soustředit hlavně na logické chyby v kódu. Co však narozdíl od skriptovacích jazyků postrádá je podpora funkcionálního programování. Pro modelování komplexních systémů se strukturami s rekurzivní agregací (např. cesta, která je podtřídou dráhy, rozsekaná na části, které jsou také podtřídami dráhy) je pro určení zprávy posílané všem částem z cesty bych raději implementoval pomocí delegace a referencí na metody. Zde totiž může vzniknout neúměrné množství podobného (rozkopírovaného kódu), což jsem částečně vyřešil pomocí reflexe, ale toto řešení je neefektivní, protože se musí nalézt metoda podle řetězce se jménem a ty řetězce „evidovat“ navíc.

4.3 JDisco

Implementovat simulační rutiny a navíc ještě kód modelu je dlouhodobější záležitost. Existuje spoustu simulačních knihoven[1], povětšinou jsou však pouze diskrétní. Tyto knihovny bych mohl o kombinovanou simulaci rozšířit. Ale proč, když existuje již hotová, celkem snadno použitelná knihovna. Navíc se obávám, že by tato implementace sama o sobě vydala minimálně na jeden samostatný projekt. Další možností je spojení přes nativní metody s knihovnou v jiném jazyce (např. SIMLIB). To je také časově náročné.

JDisco – [6] je balík javových tříd pro popis a simulaci kombinovaných modelů. Byla vyvinuta na Roskilde University v Dánsku. Existují zde dva typy procesů: *diskrétní* a *spojitý* (jak bylo řečeno v 3.2). Mezi naplánovanými *diskrétními* procesy jsou neaktivní fáze,ve kterých mohou být aktivní *spojité*. Pro popis spojitých proměnných slouží třída `jDisco.Variable`. Ta má dva hlavní atributy: okamžitou hodnotu – `state` a okamžitou derivaci – `rate`. Spojité procesy vytvořím zděděním od `jDisco.Continuous` a povinnou implementací metody `derivatives`, ve které se popíší vztahy mezi proměnnými. Obě tyto třídy mají metody `start` a `stop`, jimiž můžu určit, kdy má běžet spojitý výpočet jejich zavoláním v popisu události. Jinak řečeno lze takto definovat intervaly spojitosti procesů či proměnných. Zděděním od `jDisco.Process` a implementováním metody `actions` vznikne diskrétní proces. Implementací rozhraní `jDisco.Condition` vytvořím stavovou podmínku na kterou bude proces čekat ve volaní metody `waitUntil`. Dále balík obsahuje další třídy pro zjednodušení popisu modelu, také několik numerických metod (různé `Monitory`). Nebudu zde dělat jejich výčet. Dokumentaci této knihovny sem přidal[2] k programové dokumentaci své práce. Více tedy tam.

4.3.1 Řízení simulace

Simulace je řízena na pozadí zvoleným `Monitorem`, který je zodpovědný za to, že se:

1. stav modelu mění spojitě mezi událostmi
2. diskrétní události provedou ve správný čas
3. provede sběr informací o chování modelu

Spojité procesy pracují paralelně a jsou synchronizovány s diskrétními, které jsou pomocí kalendáře prováděny kvaziparalelně. V jednom okamžiku simulace běží tedy buď jeden diskrétní proces nebo `Monitor` řídí výpočet několika spojitých. Proces je javové vlákno vzdávající se procesoru těsně před zavoláním `Object.wait` na sebe, zavolá `Object.notify` procesu následujícím v kalendáři.

[1] SimPack, JDEVS, pak několik stejnojmenných JSimů a JavaSimů
[2] resp. se vygeneruje s programovou dokumentací

Kapitola 5

Návrh a implementace

Tato kapitola je zaměřena na stručný popis implementovaných součástí. Podrobnosti obsahuje programová dokumentace a zdrojové soubory. Nejprve je popsána celková struktura zdrojových kódů a poté rozvedeny jednotlivé součásti, především ty nejdůležitější.

5.1 Rozdělení do modulů – balíků

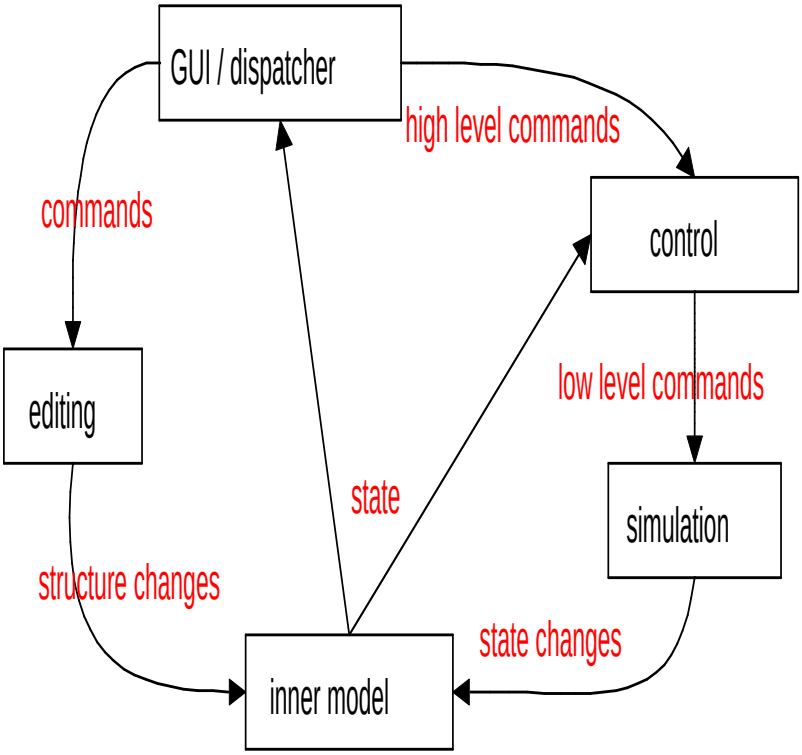
Úplně nejprve je třeba si uvědomit a mít na paměti celkovou strukturu aplikace, i když z ní zatím budou implementovány některé části. Mohlo by se stát, že něco naimplementuji a pak to budu pracně předělávat[1], protože to nepůjde spojit. Na obrázku 5.1 jsou ve vyšší abstrakci znázorněny základní funkční moduly aplikace a komunikaci mezi nimy, tak jak jsem předpokládal na začátku návrhu. Program má pracovat ve dvou módech: editace a simulace. Zde jsem se rozhodl, že bude lepší vytvořit si obálku pro přístup k datovému modelu. Módy mají společné operace, ale také ty, které jsou možné jen v tom určitém módu. Toto tedy zapouzdřuje rozhraní[2] `Context` a jeho podrozhraní. Jeho zatím jediná implementace `DefaultContext` na první pohled vypadá jako jako vševědoucí třída či jako odkladna neumístitelných funkčností. Faktem ale je, že jsou zde většinou soustředěny metody, propojující různé části programu. Pro základ projektu však nepotřebuji mít zatím dvě různé implementace, postačí dbát na oddělování pomocí rozhraní. Ale umožňuje to, že simulační kontext nebude jen v paměti, ale může být sdílen mezi aplikacemi například po síti nebo v databázi.

Mezi moduly (balíky) vznikají závislosti, tj. kód jednoho balíku se odkazuje na jiný. Zvlášť nepříjemné jsou cyklické, které nejsou znakem dobrého návrhu, pokud vznikají neřízeně ve velkém množství. Je-li snaha vytvářet moduly co nejvíc nezávislé, bude kód znovupoužitelnější. Na obrázku 5.2 jsou znázorněny stanovené závislosti modulů v současné implementaci. Předpokládám, že čtenáři z toho vyplynou i tranzitivní. Této zásady jsem držel až na nějaké výjimky, zaněž v budoucnu pravděpodobně zaplatím. Zpětná závislost dat na kontextu je ale řešena přes rozhraní. Navíc třeba třída `AbstractPath` je spíše základem pro balík řízení.

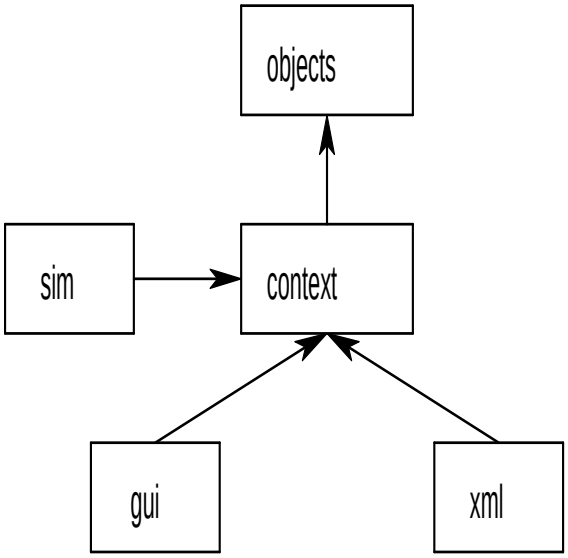
Všechny balíky z obrázku 5.2 jsou také závislé na balíku `util`, ten už je závislý jen na standardních knihovnách. Zejména na kolekcích, které rozšiřuje o další potřebné datové

[1] během návrhu nelze mít na mysli všechny detaily, obzvlášť tam kde se mění požadavky – tak se sem tam provede menší předělávka

[2] Java interfaces – v sekci 4.2



Obrázek 5.1: Základní struktura aplikace a tok dat



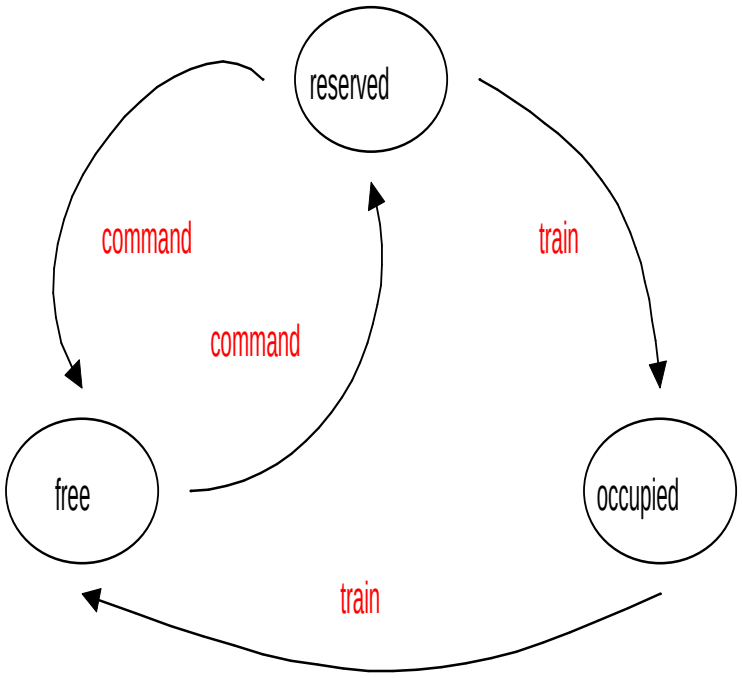
Obrázek 5.2: Graf bezprostředních závislostí balíků

struktury, jako například neorientovaný graf. O těchto strukturách více na straně 11. V následujících podkapitolách popíši další balíky jednotlivě.

5.2 Síť a datový model

V této sekci je blíže popsána vnitřní reprezentace sítě. [13] Částečně jsem se inspiroval také v [4].

Z pohledu dispečera je to neorientovaný graf $G = (U, H)$. Ke každé dvojici objektů $(uzel1, uzel2)$ kde $uzel1 \neq uzel2$ je přiřazen maximálně jeden objekt hrany. Tento graf (především jeho uzly) je umístěn ve dvourozměrném prostoru s celočíselnými indexy. Tento prostor je jakýsi pseudorastr – matice objektů. Hrana je rozdělena na části a ty vyplňují volné buňky na přímce[3] mezi nimy. V každé buňce se samozřejmě může nacházet nejvýše jeden objekt. Uzly jsou na hrany propojeny pomocí určení okolí buňky (při editaci je nalezena), které zapouzdřuje výčtový typ `Cell.Segment`. Každá buňka má množinu segmentů[4] , a pouze do tohoto okolí lze ke každé dvojici $(segment, uzel)$ lze připojit maximálně jednu hranu. Z pohledu vlaku se blok, představovaný hranou, může skládat z více oddílů, mezi nimiž jsou dispečerem neovlivnitelná (automatická) návěstidla. Dále existuje tzv. `TrackFacility` – je to nejmenší i více-oddílová jednotka, ve které se může nacházet maximálně jeden vlak. Na obrázku 5.3 jsou znázorněny stavy koleje a co může vyvolat



Obrázek 5.3: Životní cyklus `TrackFacility`

jejich přechod. Tyto stavy jsou diskrétní a vyjádřil jsem je pomocí výčtového typu. Mělo by být jasné, že vlak může vjet pouze na rezervovanou kolej, také že rezervovat nelze pokud již je rezervovaná či obsazená. Rezervuje ji příkaz dispečera a obsazuje resp. uvolňuje vlak.

[3] Bresenhamův algoritmus
[4] slouží i k vykreslování

5.2.1 Prvky sítě

- Uzly: `NodeCell` – předpokládám, že v každém uzlu je čidlo, ovlivňující obsazení napojených hran. Jeho činnost je simulována jízdou vlaku (viz 5.3.2) Kromě výhybky jsou následující uzly orientované.

  Výhybka je nastavena buď na hlavní směr nebo do odbočky

  Hlavní návěst měnící se signál

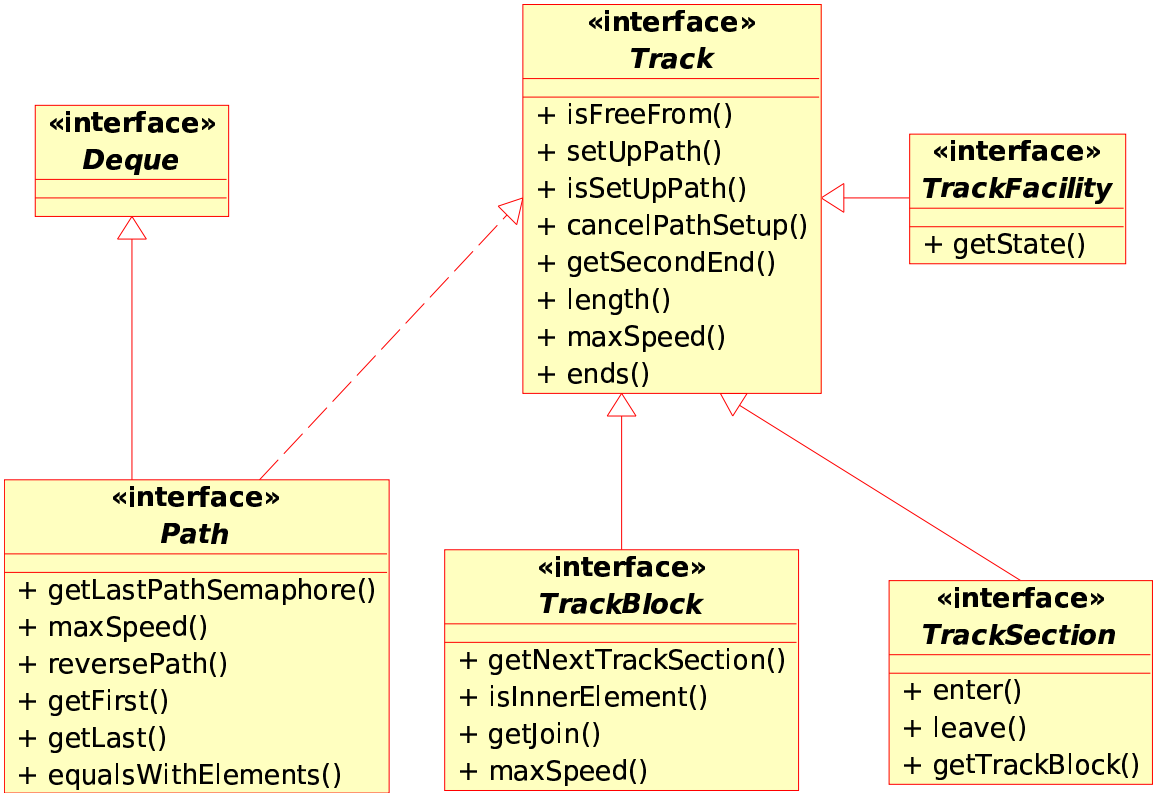
  Vstupně-výstupní bod představuje externí kolejový systém. Nebo můžou být jako rozšíření propojeny dva uvnitř modelu bez simulačního Workeru – podpora mimoúrovňových křížení.

  Zakončení koleje konstantní návěst „Stůj“

- Hrany: `TrackBlock` Obrázek 5.4 znázorňuje odvození bloku od dráhy. Jednu z možných hran představuje jednoduchá kolej `SimpleTrackBlock`. Je to `TrackFacility` s jedním oddílem. Dalšími hranami mohou být mezistaniční úseky:

  Automatický blok má více oddílů – viz strana 21

  Poloautomatický blok facility, 1 hlavní oddíl + 2 předstaniční



Obrázek 5.4: Diagram odvozenin dráhy

5.2.2 Dynamická konfigurace a cesty v síti

Rezervované koleje a dovolující signály představují jednu či více tzv. „postavených vlakových cest“. Tato cesta se jízdou vlaku postupně rozpadá a může se za určitých podmínek měnit. Formálně je to cesta z teorie grafů tj. posloupnost uzlů a neobsazených hran, ale s ohledem na konfiguraci prvků – konkrétněji:

- počáteční a koncový uzel je návěstidlo ve směru cesty či vstupně-výstupní bod
- výhybky propojují hrany (oba sousedy z posloupnosti)
- návěstidla ve směru musí dovolovat jízdu, až na poslední, které ji zakazuje
- protisměrná návěstidla: hlavní zakazují (dále v nedoimplementovných – oddílová jsou vypnutá a na předzvěsti nemá cesta vliv)

Tato funkčnost[5] je implementována v `AbstractPath`. Na obrázku 5.4 je také znázorněna návaznost cesty na dráhu.

5.2.3 Datový soubor

Pro ukládání a načítání dat jsem navrhl tzv. `ContextFactory`. Zvolil jsem XML s validací pomocí XML Schema (viz 4.1). Současný datový formát je v základní podobě a představuje skládání prvků sítě na nejnižší uživatelem upravitelné úrovni. Představuje statický popis sítě stanice. Jednotivé uzly a hrany jsou obyčejně vyjmenovány jako podelementy kořenového elementu. K rozlišení typů prvků používám mj. výčtové typy. Nejenže šetří místo a jsou jednoduše rozšiřitelné, ale snadno se i načítají a zapisují.

5.3 Simulace

Tato sekce popisuje průběh simulace – neformálním popisem a pomocí vybraných abstraktních modelů nejdůležitějších procesů.

Proč kombinovaná simulace?

Poloha vlaku se dá pojmout jak diskrétně - tj. které kolejové obvody obsazuje. Také je však třeba znát jeho přesnější polohu, rychlost a zrychlení v každém okamžiku. Míra přesnosti je nastavitelná. Umožňuje více možností a lépe se mění. Například lze změnit rovnici v popisu pohybu.

5.3.1 Vstupně-výstupní body – `InOut`

Představuje vstup a výstup do kolejiště. Jeho chování, jehož základem je fronta vlaků, popisuje `InOutWorker`. V jednoduchých modelech lze do ní vkládat přímo generátorem, ve složitějších to musí provést vyšší řídící logika – dispečer. Sám o sobě umí povolit cestu k nejbližšímu návěstidlu orientovanému ve směru jízdy, pokud výhybky k nějakému cestu propojují[6] a zároveň je nějaký vlak ve frontě a zároveň jsou na ní volné všechny koleje. V budoucnosti může představovat připojení externího kolejiště. Momentálně uvažuji modely o dvou `InOut`. Když propojím pouze dva tyto `InOut` jednou kolejí, rozpoutá se mezi nimi

[5] pracující s hlavními návěstidly, protože ostatní jsou uvnitř bloku
[6] zde se je možné vstup odclonit

o tu kolej konkurenční boj. Kdo dřív přijde na řadu, může vpustit vlak do systému. Nemůže se však stát, že to provedou oba najednou nebo pustí vlaky proti sobě.

5.3.2 Jízda vlaku

Vlak čeká ve frontě vstupně-výstupního bodu na povolení k jízdě. Po jeho získání se spustí podproces `Train.Front`, který při tom, když narazí na uzel (`PathSeparator`), případně změní parametry podle něj a obsadí oddíl před ním. Až vyjede vlak celý spustí se `Train.Tail`. Ten při průjezdu každým uzlem uvolní oddíl za ním.

Každé návěstidlo poskytuje informaci o povolené rychlosti za ním a o signálu na následujícím návěstidle, pokud ovšem nesvítí signál „Stůj“. V tomto případě by měl vlak dobrzdit před ním a čekat na signál umožňující jízdu. Pro zjednodušení předpokládám, že stav návěstidel může být zjišťován pouze v diskrétním stavu, když vlak dojede k návěstidlu, protože stanovit, kdy může zaregistrovat změnu signálu je v reálu neurčité narozdíl od programu.

Při průjezdu kolem návěstidla začíná měnit rychlost v závislosti na signálu. Tzv. „rozjezd na výstrahu“, kdy se vlak u prvního návěstidla začíná rozjíždět a před následujícím zastaví, je ošetřena zvlášť. Podprocesy končí v koncových vstupně-výstupních bodech `InOut`. Zde jsem zjednodušil model předpokladem, že největší délka vlaku musí být přeci menší než nejmenší možná délka kolejového bloku mezi dvěma `InOut`. Takovéhle podobné okrajové případy nemá cenu jinak ošetřovat, než prohlásit model za chybný. Detekce této chyby vyžaduje hledání minimálních cest.

Formálně je to kombinovaný proces (viz sekce 3.2.3). Diskrétní část čeká na stavové podmínky dojezdu k semaforu a nastavuje zrychlení. Tuto interakci popisuje Petriho síť na obrázku 3.2, ta slouží však jen pro názornost. Simulace není implementována přímo pomocí PS. Zvolil jsem jednouchý spojitý model, protože je zhlediska odladění diskrétních interakcí předvídatelnější. Základní pohybové rovnice 5.1 a 5.2 pro zrychlení a, rychlost v a dráhu s:

$$v = \int a \, \mathrm{d}t \tag{5.1}$$

$$s = \int v \, \mathrm{d}t \tag{5.2}$$

Výpočet zrychlení a v rovnici 5.3 z počáteční rychlosti v_0, cílové rychlosti v_1 a dráhy s pro zrychlování:

$$\begin{aligned} \Delta v &= v_1 - v_0 \\ a &= \frac{\Delta v(v_1 + v_0)}{2s} \end{aligned} \tag{5.3}$$

5.4 Příklad – Výhybna

Pro lepší demonstraci jak funguje simulační model jsem vytvořil „výhybnu“ řízenou na základě jednoduchých pravidel. Ke každému ovlivnitelnému návěstidlu je na pevno přiřazena alespoň jedna vytvořená cesta, která vede na další kolej buď s možností odstávky či opuštění systému, tedy z určitého pohledu představuje jednu nedělitelnou operaci přesunu. Vlaky jsou vytvářeny s jednoduchým jízdním řádem (dva údaje odkud, kam – `InOut`) pomocí generátoru s exponenciálním rozložením doby mezi jednotlivými generováními. Dříve než jsou

před vloženy do fronty `InOut`, mělo by se předejít zahlcení kolejiště. Řídící proces iterativně prochází znalostmi o systému:

1. zapomene vlaky, které už projely systémem
2. z vlaků, které čekají na vstup vybere tolik, aby byly celkem v systému nejvýše dva a udělí jim souhlas (aktivováním procesu)
3. projde všechny kolejové bloky
   (a) ve kterých se nachází vlak nebo je rezervovaná cesta, zjistí jeho resp. její směr – návěstidlo ke kterému je vlak veden
   (b) pokud je z tohoto návěstidla volná cesta (předem uložená) postaví ji

Obrázek 5.5: Kolejové schéma výhybny

Ke schématu na obrázku 5.5: úsečky jsou koleje, jejich spojení jsou výhybky, • je `InOut` a ▶ je návěstidlo. Po spuštění simulace se na standardní výstup programu vypisují po řádkách zprávy:

1. jízda vlaku, spojitý vzorek:

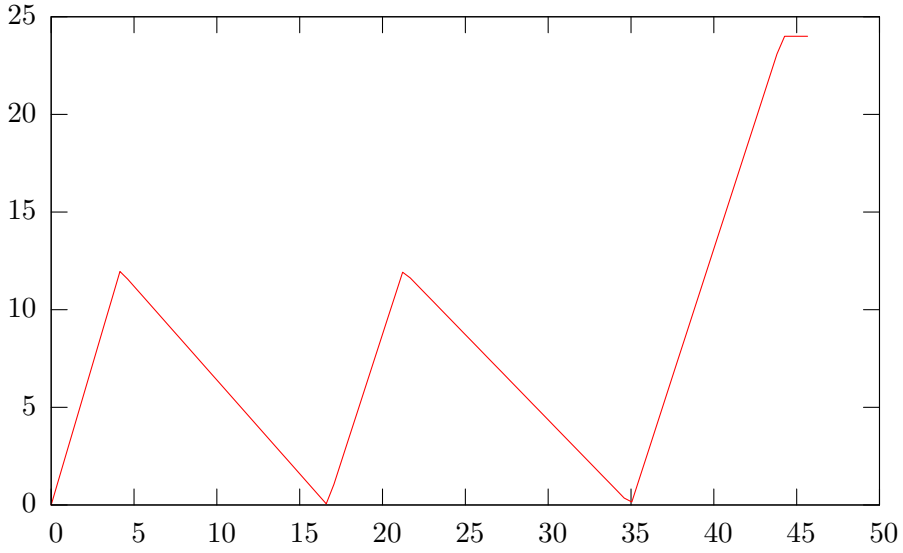
   ```
   čas objekt zrychlení rychlost dráha kolej_od kolej_do vzd_návěstidlo
   ```

2. diskrétní událost prvku:

   ```
   čas objekt zpráva
   ```

`dráha` je celková dráha ujetá vlakem, `kolej_od` kolej na které se nachází začátek vlaku, `kolej_do` kolej na které se nachází konec vlaku, `vzd_návěstidlo` vzdálenost k nejbližšímu návěstidlu.

V grafu na obrázku 5.6 je znázorněno, jak vlak v modelu reaguje na signály během události dojezdu k návěstidlu. Zde je zrovna vidět jak řídící logika nestačí povolit cestu do následujícího bloku včas, a vlak se mezi semafory rozjíždí a zastavuje. V sekci 5.3.2 je vysvětleno chování vlaku.



Obrázek 5.6: Graf rychlosti vlaku při průjezdu výhybnou

Kapitola 6

Závěr

V této kapitole jsou popsána možná rozšíření programu a zhodnocení celé práce.

6.1 Náměty na rozšíření

Nápadů na to jak rozšířit současnou aplikaci je plno, ale zde se zaměřím spíše na ty, pro které je současná implementace přímo navržena, nebo z ní vyplývají a jsou v nejbližší době realizovatelné.

Cíle výsledné aplikace

Tyto cíle jsem vytyčil z důvodu, že nechci, aby naimplementovaný kód byl vytvořen jen pro jeden účel. Tímto bych chtěl nějak vyjádřit možnosti dalšího rozvoje celé aplikace. Na mou práci mohou navázat a uskutečnit je jiní studenti.

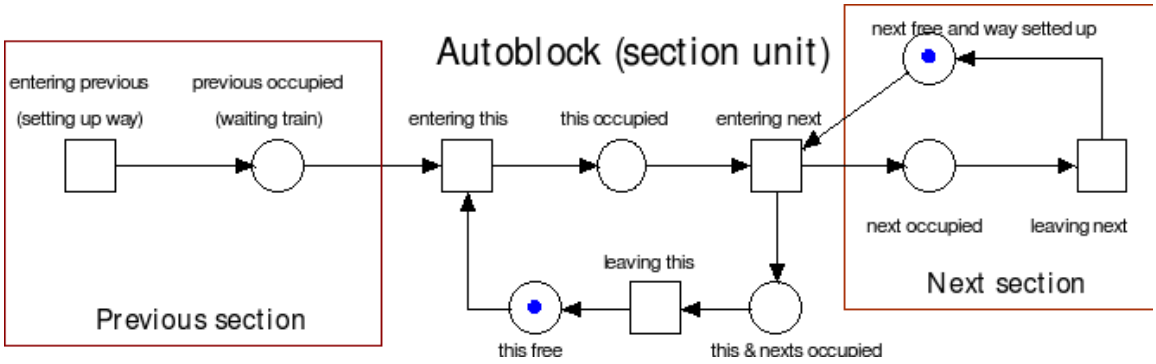
- grafický editor a simulátor řízení železniční sítě v jedné aplikaci, zaměření na funkčnost, minimalizace klikání
- platformní nezávislost – alespoň na úrovni zdrojových kódů
- formát dat, který umožňuje vytvářet šablony prvků kompozicí z jiných

Fyzikální model jízdy

Pro současný popis jízdy existuje i analytické řešení. Popis pomocí diferenciální rovnice ale umožňuje její změnu za složitější, která bude věrněji modelovat další možné případy. Dráhu vlaku (hlavně v mezistaničních úsecích) bych pojal jako křivku v trojrozměrném prostoru – tj. další vlastnost hrany. Při tom je třeba rozložit si současnou jedinou sílu udělující výsledné zrychlení na několik složek – nakloněná rovina v kopcích, tření, tažná síla motoru, ta by mohla být řízena například fuzzy regulátorem. Můžu pak uvažovat i poruchy, při které je síla motoru nulová.

Automatický blok

Automatický blok (AB) je pevně postavená, automaticky obnovovaná, cesta pro více vlaků. Může to být napevno `TrackBlock` poskládaný z několika oddílů `SimpleTrack`. AB jako celek má dipečerem nastavený směr. Jednotlivé oddíly okamžitě rezervují cestu v tom směru, když dojde k jejich uvolnění. Na obrázku 6.1 je znázorněno chování oddílu AB. AB by také mělo



Obrázek 6.1: C/E Petriho síť oddílu v autobloku

být možné vytvořit z postavené cesty. Dále s mezistaničními úseky souvisí i použití dalších druhů návěstí (záleží kde budou v síti umístěny):

Oddílová návěst v autobloku

Předzvěst pomocí konstantní návěsti s „Volno“

Ukládání stavu simulace

Diskrétní proces je rozkouskovaný na události příkazy, jako je čekání, vedoucí ke ztrátě aktivity. Pokud nějakým univerzálním způsobem vytvořím možnost pamatovat stav ve kterém je každý proces zrovna pasivní mezi událostmi a definuji transformaci z počátku do tohoto stavu, můžu tuto informaci uložit společně s dalšími vlastnostmi při serializaci. Proces by mohl být vnitřně reprezentován jako automat či Petriho síť a jeho resp. její stav by se uložil a obnovil. U spojitých proměnných je třeba ukládat `jDisco.Variable`. Je možné, že kvůli univerzálnímu řešení, bude vyžadovat úpravy v knihovně *jDisco* pro podporu serializace. Stav potomků `LoopProcess`, který uvnitř iterace nepřichází do pasivního stavu, je možné již ukládat.

Vizualizace pomocí celulárních automatů, Real-time

Čtvercová síť modelu přímo předurčuje použít pro řízení vykreslování celulární automaty. Snahou je minimalizovat množství překreslované plochy a zbytečnou výpočetní zátěž, způsobenou procházením stavů všech buněk, omezením pouze na malé množství buněk některých typů a u ostatních budu vycházet z předpokladu, že pokud se změní jedna buňka, tak pravděpodobnost změny v buňkách v okolí[1] se nastaví na maximum a pak klesá s časem. Dále bude vhodné vytvořit `Monitor` se synchronizací reálného času s modelovým. Ten by měl lépe detekovat a vhodně vkládat „odmlky“ mezi spojitý výpočet. V současnosti to ošetřuji diskrétním procesem, který pro každou sekundu modelového času, po kterou čeká tj. simulují se jiné procesy, spočítá uběhlý čas a vloží přímo potřebnou pauzu javového vlákna. Kvaziparalelismus zajistí, že se pozdrží výpočet celé simulace. Je to tedy vložení jakýchsi synchronizačních impulsů, které je však třeba naplánovat v lepší okamžik.

[1] neprázdné buňky

Rozhraní do modelu pro řízení dispečerem

Navázat je možné na současnou formalizaci cest, doimplementovat jejich vyhledávání. Vytvořit balíček rutin a funkcí zastřešující přístup do modelu pro řízení z několika úrovní. Na to také navazuje dodělat příkazy z GUI rozhraní. Vhodné bude zformalizovat posloupnost příkazů jako jazyk, který se dá případně optimalizovat.

6.2 Zhodnocení dosažených výsledků

Program simuluje pohyb několika vlaků v síti současně. Každý vlak se pohybuje podle během simulace stanovené cesty – reaguje na signály návěstidel. Do obsazených kolejí nemůže vjet další vlak. Stanovených minimálních cílů bylo tedy dosaženo, což demonstruje příklad. Složitější síť je rozumně předveditelná až v případě doimplementování řídícího modulu a vizualizace simulace, kdy kontrolu musí převzít člověk. Validita tohoto simulačního modelu by šla ověřit sbíráním statistik o skutečných vlakových sítích a srovnáním s modelem.

Vhodné rysy jazyků resp. nástrojů nakonec převažují, přestože některé konstrukce v nich byly obtížněji vyjádřitelné. Pokud jsem měl prověřit jakou zátěž je simulace na operační systém, vždy se hodnoty během spuštěné instance virtuálního stroje[2] pohybovaly na mém přístroji kolem těchto hodnot a nijak výrazně se neměnily: sdílená paměť – 10 MB, kód a data ve fyzické paměti – 25 MB, množství virtuální paměti – 213 MB, výkon CPU 91 % (toto není pokus o sofistikovanou výkonostní analýzu, jenom jsem sledoval, zda si program nevynucuje zbytečně moc prostředků).

Z vlastní iniciativy jsem si vyzkoušel možnost pracovat na takovémto projektu a ponořit se do problematiky analýzy a modelování komplexních systémů, což zahrnuje zabývat se případovými studiemi technických zařízení. Dále jsem si rozšířil obzor, o to jak zakomponovat simulaci, a zdokonalil se v návrhu a programování aplikace. Problém jsem měl zejména ze začátku s odhadem časové náročnosti prací. Samotné kódování šlo poměrně rychle, ale na ladění jsem v odhadech času pozapomněl. Raději testuji průběžně, chyba se potom snadněji hledá. U rozsáhlých projektů je však třeba ještě vytvářet testovací skripty. V oblasti návrhu je stále se co učit a více využít návrhových vzorů a hlavně se držet jejich zásad. Ale musím poznamenat, že doba strávená studováním a návrhem systému byla delší než doba programování, kterou bych chtěl v pokračování na výstavbě této aplikace ještě zkrátit o vyvarování se chybám, z kterých jsem se dopustil.

[2] Spustil jsem výhybnu bez synchronizace s reálným časem na 15 minut

Literatura

[1] Vyhláška Ministerstva dopravy, kterou se vydává dopravní řád drah. 1995.
URL http://www.mvcr.cz/sbirka/

[2] XML Schema Part 1: Structures Second Edition. web, 2004.
URL http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[3] JDK 6 Documentation. web, 2006.
URL http://java.sun.com/javase/6/docs/

[4] FALKNER, A.; FLEISCHANDERL, G.: Configuration requirements from railway interlocking stations. Technická zpráva, Siemens Austria, 2001.

[5] HAROLD, E. R.; MEANS, W. S.: *XML v kostce*. Computer Press, 2002.

[6] HELSGAUN, K.: jDisco – a Java package for combined discrete and continuous simulation. Technická zpráva, Department of Computer Science, Roskilde University, 2001.
URL http://www.akira.ruc.dk/~keld/research/JDISCO/

[7] HEROUT, P.: *Java - bohatství knihoven*. Kopp, České Budějovice, 2003.

[8] NOVOSAD, P.: Výukový nástroj pro práci s C/E Petriho sítěmi. In *Pedagogický software 2006*, Zemědělská Fakulta, Jihočeská Univerzita, 2006, ISBN 80-85645-56-4, s. 247–249.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8103

[9] PECINOVSKÝ, R.: *Java 5.0 – Novinky jazyka a upgrade aplikací*. Computer Press Brno, 2005.

[10] REKTORYS, K.; kol.: *Přehled užité matematiky I. - II.* Prometheus, 2000.

[11] RÁBOVÁ, Z.; ČEŠKA, M.; ZENDULKA, J.; aj.: *Modelování a simulace*. VUT FEI, druhé vydání.

[12] SPELL, B.: *Java Programujeme profesionálně*. Computer Press Praha, 2002.

[13] WRÓBLEWSKI, P.: *Algoritmy – Datové struktury a programovací techniky*. Computer Press Brno, 2004.

Seznam použitých zkratek

AB Automatický blok – traťové zabezpečovací zařízení rozdělené na prostorové oddíly

DOM Document Object Model – více v [5]

GCC GNU Compiler Collection – http://www.gnu.org/software/gcc/

JOP Jednotné obslužné pracoviště – řídící zařízení umožňující kontrolovat několik stanic najednou

M&S Modelování a simulace – viz sekce 3.2

PS Petriho síť – viz sekce 3.2

SAX Simple API for XML – více v [5]

SIMLIB SIMLIB/C++ – http://www.fit.vutbr.cz/~peringer/SIMLIB/

STL Standard Template Library – knihovní funkce jazyka C++

XSLT eXtensible Stylesheet Language Transformations – funkcionální programovací jazyk v XML pro specifikaci převodu vstupního XML na výstupní formát (více v [5])