VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

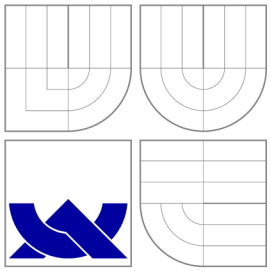
ŘÍZENÍ PROCESŮ S DYNAMICKOU OPTIMALIZACÍ ROZVRHU ZDROJŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

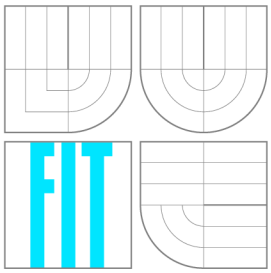
AUTOR PRÁCE
AUTHOR

Bc. JAN ŠINKORA

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ŘÍZENÍ PROCESŮ S DYNAMICKOU OPTIMALIZACÍ ROZVRHU ZDROJŮ

SOFTWARE TOOLS FOR DYNAMIC PLANNING AND SCHEDULING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN ŠINKORA

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.

BRNO 2012

Abstrakt

Práce se zabývá mezioborovou problematikou na pomezí informačních technologií a optimalizace procesů. Jsou zde využity a rozšířeny dříve navržené postupy pro modelování projektů a zdrojů pomocí objektově orientovaných Petriho sítí. Dále se rozebírají možnosti použití genetických algoritmů pro optimalizaci rozvrhů zdrojů, které určují jejich přiřazení k jednotlivým aktivitám v dynamických systémech. Je popsána třída rozvrhovacích problémů s omezenými zdroji a způsob, jakým lze tyto projekty implementovat. Také je ukázáno, jak lze vytvořit složitější model výroby inspirovaný reálnými výrobními procesy. Dále je navržen řídící agent, který sleduje běžící výrobní systém a umožňuje jeho dynamickou optimalizaci. Celý systém je implementován v prostředí Squeak Smalltalk za využití nástroje PNtalk, který je experimentální implementací objektově orientovaných sítí.

Abstract

This project pursues issues on the border of information technologies and process optimization. Previously published concepts of modeling projects and shared resources with object-oriented Petri nets are presented and further expanded. The possibilites of the use of genetic algorithms for dynamic realtime optimization of the resource schedules are explored. The resource constrained project sheduling problem is presented and it is shown, how instances of the problem can be implemented. A more complex model that is inspired by real production systems is then created. Next, a control agent, which monitors a running production system and allows for it's dynamic optimization is designed. The whole system is implemented in the Squeak Smalltalk environment with the use of the tool PNtalk, which is an experimental implementation of the object oriented Petri nets paradigm.

Klíčová slova

Rozvrhování, RCPSP, Dynamické systémy, Petriho sítě, OOPN, Squeak, Smalltalk, PNtalk, Genetické algoritmy

Keywords

Scheduling, RCPSP, Dynamic systems, Petri nets, OOPN, Squeak, Smalltalk, PNtalk, Genetic algorithms

Citace

Jan Šinkora: Řízení procesů s dynamickou optimalizací rozvrhu zdrojů, diplomová práce, Brno, FIT VUT v Brně, 2012

Řízení procesů s dynamickou optimalizací rozvrhu zdrojů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Vladimíra Janouška

.....
Jan Šinkora
23.5.2012

Poděkování

Děkuji doc. Ing. Vladimíru Janouškovi, Ph.D. za vedení při zpracování této diplomové práce, Ing. Radku Kočímu, Ph.D. za spolupráci při hledání chyb a ladění a Ing. Milanu Hrnčířovi za potřebná data a odbornou konzultaci při sestavování výrobních modelů.

Obsah

1 Úvod **3**

2 Rozvrhování **5**
2.1 Klasifikace rozvrhovacích problémů . . . 5
2.2 Dynamické rozvrhovací problémy . . . 5
2.3 Rozvrhovací problém projektů s omezenými zdroji . . . 6
2.3.1 Rozšíření RCPSP problému . . . 7
2.3.2 Shrnutí rozvrhovacích problémů . . . 7
2.4 Evoluční algoritmy . . . 7
2.5 Kódování rozvrhů pro účely evolučních algoritmů . . . 8
2.5.1 Genetické operátory pro rozvrhovací chromozom . . . 9
2.6 Analýza efektivnosti popsaného kódování . . . 9
2.7 Volba parametrů GA . . . 10
2.8 Fitness funkce . . . 11

3 Objektově orientované Petriho sítě **12**
3.1 Petriho sítě . . . 12
3.2 Objektová orientace v Petriho sítích . . . 12
3.2.1 Synchronní porty . . . 13
3.2.2 Základní konstrukce při vytváření tříd . . . 14
3.3 Nástroj PNtalk . . . 15
3.3.1 Repozitář . . . 16
3.3.2 Jazyk PNtalk . . . 16
3.3.3 Simulátor . . . 16
3.3.4 Metaprotokol . . . 16
3.4 Modelování procesů a zdrojů pomocí OOPN . . . 18
3.4.1 Projektové portfolio . . . 19
3.4.2 Projekty . . . 19
3.4.3 Zdroje . . . 20
3.5 Modelování portfolia . . . 23
3.5.1 Portfolio a zdroje konkrétního systému . . . 23
3.5.2 Knihovna PSPLIB . . . 24
3.5.3 Model inspirovaný reálným problémem . . . 27

4 Návrh a implementace systému **32**
4.1 Balíček RTScheduling-System: Hlavní třídy systému . . . 32
4.1.1 Změny modelu reálného prostředí . . . 33
4.1.2 Realizace změn v modelu prostředí . . . 34

4.1.3 Logování průběhu experimentu . . . 34
4.2 Balíček RTScheduling-Scenarios: Scénáře a události . . . 34
4.3 Balíček RTScheduling-Control: Řídící systém . . . 34
4.3.1 Vnitřní model prostředí . . . 35
4.3.2 Proces optimalizace . . . 35
4.3.3 Implementace řídícího systému . . . 36
4.3.4 Protokol komunikace s optimalizátorem . . . 36
4.3.5 Externí optimalizátor . . . 37

5 Testování a vyhodnocení **38**
5.1 Simulace PNtalku . . . 38
5.2 Efektivita optimalizačního procesu . . . 38
5.2.1 Porovnání s referenčním řešením z knihovny PSPLIB . . . 38
5.2.2 Model inspirovaný reálnou výrobou . . . 40
5.2.3 Demonstrace scénáře s více událostmi . . . 40
5.2.4 Demonstrace portfolia s více projekty . . . 41
5.3 Problémy při experimentech . . . 42

6 Závěr **43**

A Obsah přiloženého média **48**

B Konfigurace a spuštění **49**

C Detailní popis výrobních postupů **50**

Kapitola 1

Úvod

Dynamický systém, jehož části vyžadují omezené sdílené zdroje, vyžaduje pro efektivní běh koordinované přiřazování těchto zdrojů k aktivitám. Rozvrh, který přiřazení zdrojů řídí, je tedy také dynamický a je třeba ho při změnách systému optimalizovat na základě daných kritérií. Funkční a kvalitní řídící systém umožňuje neustálý vývoj řízeného systému bez potřeby přerušení jeho běhu. Takové řízení může být vhodné například pro výrobní systémy, kdy manuální přepočet rozvrhu při nečekané změně (nová objednávka atd.) způsobí díky trvání či nedokonalosti výsledku ztráty. Tato diplomová práce se věnuje právě problematice tvorby dynamického rozvrhu pro systémy běžící v reálném čase. Práce staví na již publikovaných teoretických článcích popisujících koncepty, které doposud nebyly implementovány. Cílem práce je tyto koncepty ověřit, rozšířit je do reálně použitelné podoby, implementovat, otestovat a vyhodnotit jejich použitelnost.

V druhé kapitole je představen problém rozvrhování v souvislosti s dynamickými systémy. Je zde formálně popsán rozvrhovací problém s omezenými zdroji, jehož varianty lze nalézt v průmyslové výrobě a je tedy vhodné na něm principy dynamického rozvrhování prezentovat. Jsou ukázány způsoby, jak lze tento problém rozšířit, aby se více blížil reálným procesům. Dále je ukázán princip využití evolučních algoritmů při optimalizaci rozvrhu pro tento problém a nakonec je navržen konkrétní algoritmus pro optimalizaci rozvrhů, který dovoluje optimalizaci rozšířeného rozvrhovacího problému.

Třetí kapitola se věnuje modelovacímu paradigmatu nazývanému objektově orientované Petriho sítě (OOPN z anglického Object Oriented Petri Nets) a nástroji PNtalk, který je experimentální implementací tohoto paradigmatu v prostředí Squeak Smalltalk. Je zde ukázán princip, jak lze pomocí OOPN modelovat projektové portfolio, které může reprezentovat například nějaký výrobní systém, kde výrobky korespondují s dokončením jednotlivých projektů. PNtalk umožňuje vytvářet kopie simulací a tyto je možné použít za běhu reálného systému pro výpočet funkce fitness pro genetické algoritmy. Dále je zde představen způsob, jakým lze problémy s omezenými zdroji z knihovny PSPLIB konvertovat do navrženého systému portfolií. Kapitola také představuje složitější portfolio modelující výrobní proces inspirovaný reálnými průmyslovými postupy.

Ve čtvrté kapitole je popsán návrh řídícího systému spolu s popisem jeho implementace. Jsou identifikovány různé dynamické změny, které mohou v modelu běžícího portfolia nastat. Součástí je samotný řídící agent, který má za úkol při těchto změnách optimalizovat rozvrhy zdrojů pomocí genetického algoritmu. Tento agent využívá k samotnému běhu genetického algoritmu externí aplikaci napsanou v jazyce C++ s využitím široce používané knihovny GAlib. Kapitola popisuje také způsob, jakým spolu optimalizátor a řídící agent komunikují a postupy, které využívají metaprotokol PNtalku k programovému přístupu

k simulacím.

Pátá kapitola popisuje simulační experimenty, které byly na implementovaném systému provedeny. Jsou zde popsány problémy, s jakými se systém potýká a je vyhodnocena časová náročnost výpočtu simulace PNtalku. Dále je popsán experiment ukazující scénář s více dynamickými změnami. Nakonec jsou identifikována slabá místa systému a následně navrženy způsoby, jakými by bylo možné celou aplikaci vylepšit.

Kapitola 2

Rozvrhování

Rozvrhování je možné volně popsat jako přiřazení sdílených zdrojů v čase k aktivitám, které tyto zdroje vyžadují ke svému provedení [21]. Následuje ustanovení základních termínů, které s rozvrhováním souvisí:

- **Úloha** je složena z aktivit, které je potřeba vykonat buď nezávisle, nebo v určitém pořadí.
- **Aktivita** je dále nedělitelná činnost, která vyžaduje ke svému běhu zdroje. Může mít definovanou dobu trvání, která je ovlivnitelná použitými zdroji.
- **Zdroj** lze popsat jako zařízení, které je třeba k vykonání nějaké aktivity, ale může jít také o elektřinu či pracovníka. Zdroje jsou sdílené mezi aktivitami a mohou mít různá omezení, například na počet obsluhovaných aktivit. O zdroji se mluví přímo jako o *stroji*, pokud to dává v dané úloze smysl.

Sestavení rozvrhu je obvykle prováděno s ohledem na řadu omezujících podmínek a s cílem dosáhnout co nejlepšího výsledku – jedná se tedy o optimalizační problém. Omezující podmínky mohou být klasifikovány jako měkké a tvrdé. Nedodržení měkké podmínky vede pouze k penalizaci účelové funkce, kdežto u tvrdé podmínky se stává celé řešení nepřípustným. Účelová funkce je specifická pro daný problém. U výrobních procesů může jít například o minimalizaci ztráty způsobené pozdním dodáním [19].

2.1 Klasifikace rozvrhovacích problémů

Pokud jsou všechny aktivity a zdroje předem známé a neměnné, nazýváme rozvrhování problémem *statickým*, jinak mluvíme o *dynamickém problému*. Dále můžeme označit problém jako *deterministický*, pokud jsou u jednotlivých aktivit a zdrojů dané a neměnné všechny parametry, jako například doba zpracování. Problém je *nedeterministický*, pokud jsou některé tyto parametry neznámé či nejisté.

U rozvrhování je také nutné rozlišit, zda je povoleno zpracování aktivity přerušit a později obnovit (*preemptivní* rozvrhování) či je nutné, aby aktivita došla do konce, než je možné zdroje uvolnit pro jiné aktivity (*nepreemptivní* rozvrhování) [16].

2.2 Dynamické rozvrhovací problémy

Dynamický rozvrhovací problém se vyznačuje tím, že některé parametry zdrojů a aktivit jsou předem neznámé či nejisté, popřípadě ani úlohy a jejich rozložení na aktivity nejsou

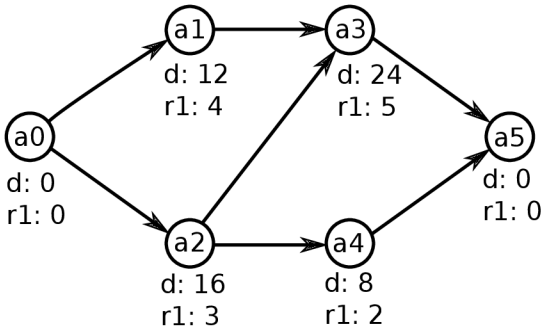
pevné a neměnné. Řešení takového problému je potenciálně nekonečný proces, pokud uvažujeme, že úlohy do systému mohou stále přibývat. Algoritmus, který takovýto problém řeší, musí být schopen reagovat na různé změny oproti původním předpokladům. Jde například o změny parametrů zdrojů nebo nedodržení předem odhadnutých časů potřebných pro vykonání aktivit. Řídící systém musí rozpoznat potřebu přepočítání rozvrhů a adaptovat je tak, aby došlo při změně parametrů k co nejmenší ztrátě z hlediska účelové funkce. Velké množství výrobních systémů spadá právě do této kategorie, dynamické chování vyplývá ze situací jako je výpadek stroje, onemocnění pracovníka či příchod nové objednávky, jejíž zpracování vyžaduje již přidělené zdroje.

2.3 Rozvrhovací problém projektů s omezenými zdroji

Zde je definována nejjednodušší podoba rozvrhovacího problému projektů s omezenými zdroji (SMRCPSP – z anglického *Single Mode Resource Constrained Project Scheduling Problem*), která byla představena v [15]. Pokud se dále v textu zmíní RCPSP, je myšlena právě tato varianta.

Mějme množinu zdrojů $\mathbb{R} = r_1, ...r_R$. Každý zdroj r_j je omezený svou kapacitou c_j. Zdroje jsou obnovitelné, což znamená, že po dokončení aktivity je obnovena kapacita zdroje, kterou si aktivita vyžádala.

Problém typu RCPSP sestává z dílčích aktivit o velikosti A: $\mathbf{A} = \{a_1, ..., a_A\}$. Každé aktivitě a_i je přiřazena délka jejího trvání $d_i \in \mathbb{R}_0^+$ a aktivitu nelze přerušit, pokud už jednou začala. Dále pro každou aktivitu existuje precedenční množina $\mathbf{P}_i \subseteq \mathbf{A}$. Její význam je takový, že každá aktivita z precedenční množiny $\mathbf{P}_i$ musí být dokončena, než je možné započít aktivitu a_i. Kromě aktivit v množině $\mathbf{A}$ jsou použity ještě dvě pomocné aktivity – a_0 a a_{A+1}, které značí započetí a ukončení projektu. Aktivita a_0 má prázdnou precedenční množinu a vyskytuje se v precedenční množině všech počátečních aktivit a naopak a_{A+1} má v precedenční množině všechny koncové aktivity a sama není v precedenční množině žádné aktivity. Z precedenční množiny a množiny aktivit lze odvodit acyklický orientovaný graf, který reprezentuje *síť aktivit* problému. Každá aktivita a_i vyžaduje $0 \geq req_{ij} \leq c_j$ jednotek zdroje r_j a kapacita zdroje nesmí být v žádném okamžiku přečerpána. Ukázka sítě aktivit jednoduchého problému typu RCPSP s jediným zdrojem je na ilustraci 2.1.



Ilustrace 2.1: Jednoduchý RCPSP projekt s jedním zdrojem a čtyřmi základními aktivitami.

Řešením problému typu RCPSP je vektor $t = (t_1, t_2, ...t_A), t_i \in \mathbb{R}_0^+$, který každé aktivitě přiřazuje její startovní čas. Tento vektor lze nazvat *rozvrhem* aktivit v čase. Řešení samozřejmě musí splňovat omezující podmínky kapacit zdrojů. V případě, že nepočítáme s prostoji mezi vykonáním následných aktivit, tzn. mezi koncem jedné a startem další ak-

Řešení	Doba běhu
$t_1 = (a_1, a_2, a_3, a_4)$	60
$t_2 = (a_1, a_2, a_4, a_3)$	60
$t_3 = (a_2, a_1, a_3, a_4)$	60
$t_4 = (a_2, a_1, a_4, a_3)$	52
$t_5 = (a_2, a_4, a_1, a_3)$	52

Tabulka 2.1: Možná řešení projektu z ilustrace 2.1, pokud má zdroj kapacitu 6. Nejlepší řešení jsou t_4 a t_5, která jsou co se týče běhu totožná, protože aktivity a_1 a a_4 mohou v obou běžet současně.

tivity nebude při dostupných zdrojích žádné čekání, lze redukovat rozvrh na jednoduché uspořádání aktivit pro jednotlivé zdroje. Podle tohoto uspořádání pak zdroje rozhodují, v jakém pořadí budou jednotlivým aktivitám přiřazovány.

Pokud má v projektu z ilustrace 2.1 zdroj kapacitu 6, existuje 5 možných řešení ukázaných v tabulce 2.1.

2.3.1 Rozšíření RCPSP problému

Kromě základní varianty RCPSP problému bylo představeno několik variant. Obecně lze problém jakkoli rozšířit. Pro demonstraci zde bude kromě základní definice uvažováno následující rozšíření: Je dána množina skupin zdrojů **G**, která je rozkladem množiny **R** takovým, že zdroje ve stejné skupině plní stejnou funkci, tedy jsou použitelné pro jeden typ práce. Aktivity poté nevyžadují pro své provedení jednotlivé zdroje, ale žádají si o libovolné zdroje z určitých skupin zdrojů. V kapitole 3 bude předvedeno, jakým způsobem lze dále přizpůsobovat RCPSP při řešení konkrétních problémů.

2.3.2 Shrnutí rozvrhovacích problémů

S problémy typu RCPSP a jejich variantami se lze setkat obecně v jakémkoli řízení procesů, například při plánování nejrůznějších výrobních procesů. RCPSP je složitější variantou tzv. Job-shop problému, který je NP-složitý, pokud je v systému uvažováno 3 a více zdrojů a účelová funkce je minimalizace času mezi začátkem první a koncem poslední aktivity [8]. Proto je vhodné hledat pro hledání dobrých rozvrhů heuristické metody.

2.4 Evoluční algoritmy

Evoluční algoritmy jsou stochastické optimalizační techniky založené na principech přirozeného výběru. Jedním z nejznámějších zástupců evolučních algoritmů je třída *genetických algoritmů* (dále GA). Tyto pracují s kolekcí kandidátních řešení, nazývanou *populace*, která se vyvíjí v iteracích nazývaných *generace*. Jednotlivá řešení jsou zakódována do *chromozomů*. Každý jedinec z populace je ohodnocen *fitness* funkcí, která určuje, jak dobré je toto řešení v kontextu daného problému. Z kandidátů jsou s přihlédnutím na jejich fitness stochasticky vybírání jedinci, kteří jsou poté použiti pro vytvoření další generace. Je přitom využíváno metod křížení (vytvoření nového kandidáta ze dvou či více rodičů) a mutace (změna v chromozomu, která není inspirována žádným z rodičů). Po určitém počtu generací, nebo pokud je dosaženo dostatečné konvergence v populaci, je vybrán nejlepší

kandidát z aktuální generace, který reprezentuje nějaké suboptimální řešení problému. Nakolik je toto suboptimum blízké optimálnímu řešení závisí na mnoha faktorech. Lepších výsledků je dosaženo například zvolením vhodných metod křížení a mutací, nebo dobrou volbou způsobu generování počáteční populace [5].

Ostatní typy evolučních algoritmů jsou na první pohled velice podobné genetickým algoritmům, ale liší se například v použitých datových strukturách pro kódování kandidátů, nebo například v pořadí jednotlivých kroků základního postupu – například třída *evolučních strategií* nejdříve pozmění aktuální populaci a až později je z ní vytvořena další generace.

Vhodnost evolučních algoritmů pro řešení rozvrhovacích problémů vyplývá z toho, že je velice složité řešit tyto problémy analyticky. Ačkoli genetický algoritmus negarantuje optimální řešení, je možné dosáhnout relativně dobrého řešení v daném čase [4].

2.5 Kódování rozvrhů pro účely evolučních algoritmů

Základní způsob kódování rozvrhu do chromozomu je představen v [18]. V této sekci bude popsán způsob kódování, který z tohoto vychází a rozšiřuje jej o výše zmíněné skupiny zdrojů. Nejprve bude definován rozvrh problému.

Mějme konečnou množinu aktivit $\mathbf{A}$, konečnou množinu zdrojů $\mathbf{R}$ a konečnou množinu skupin zdrojů $\mathbf{G}$ dle výše uvedených definic. Rozvrh problému je pak množinou rozvrhů pro každou skupinu:

$$\mathbf{S} = \{s_1, s_2, ..., s_G\}. \tag{2.1}$$

Každý rozvrh skupiny obsahuje množinu rozvrhů pro jednotlivé zdroje patřící do skupiny:

$$s_i = \{s_{i1}, s_{i2}, ...s_{iQ}\}, Q = |g_i|, \tag{2.2}$$

$$s_{ij} \in S(O), O \subseteq \mathbf{A_i}, \tag{2.3}$$

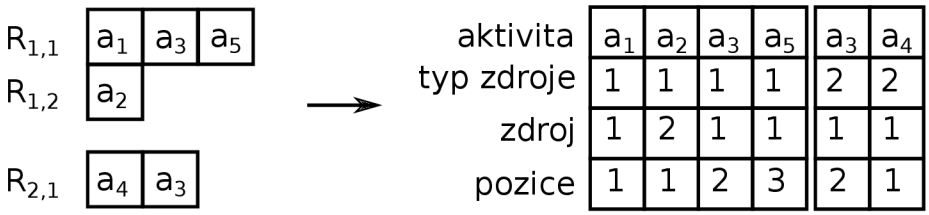
kde $S(X)$ značí množinu všech permutací množiny X a $\mathbf{A_i}$ je podmnožina aktivit, které vyžadují zdroj ze skupiny i.

Nutnou, avšak nedostačující podmínkou pro přípustnost rozvrhu je fakt, aby se každá aktivita objevovala v jednom z rozvrhů skupin zdrojů, o které aktivita žádá. Rozvrh, který tuto podmínku splňuje, může být nepřípustný kvůli zaseknutí způsobenému v případě, že aktivita, která následuje po jiné, je na sdíleném zdroji naplánována dříve. Za povšimnutí stojí fakt, že chromozom neobsahuje informaci o čase, ve kterém je aktivita započata. Časové údaje vyplývají ze simulace, která na základě genomu probíhá.

Při návrhu kódování rozvrhu do chromozomu je vhodné uvážit, jakým způsobem budou fungovat genetické operátory. Ty by měly mít schopnost při křížení zachovat na potomcích nějaké vlastnosti z obou rodičů a při mutaci změnit jakoukoli vlastnost. Při návrhu se vychází z toho, že v každé skupině musí všechny aktivity zůstat, protože jinak by se na ně při přiřazování zdrojů nikdy nedostalo. Chromozom se tedy bude skládat z menších úseků, které se při vývoji populací nemohou ovlivňovat. V rámci rozvrhu jedné skupiny zdrojů je jakákoli změna jakousi dvourozměrnou permutací – aktivita se může přesunout na jinou pozici v rozvrhu daného zdroje, nebo na jiný zdroj ze stejné skupiny.

Přestože je rozvrh skupiny dvourozměrný, lze jej realizovat jednorozměrně aniž by byly možnosti jeho vývoje omezené. Navržený chromozom je lineárním seznamem genů. Každý gen svou pozicí v chromozomu určuje, jakou konkrétní aktivitu popisuje. To znamená, že gen se přes permutační charakteristiku problému nikdy nepohybuje v chromozomu. Každý gen obsahuje tři údaje: identifikaci skupiny zdrojů, identifikaci konkrétního zdroje a unikátní

pozici aktivity v rozvrhu tohoto zdroje. Informace o tom, o jakou aktivitu jde, je skrytě zakódovaná v pozici genu a není nutné ji explicitně uchovávat, protože se v průběhu algoritmu nemění. Pouze z důvodu zjednodušení kódovacího mechanismu jsou aktivity patřící do stejné skupiny zdrojů shlukovány u sebe. Na ilustraci 2.2 lze vidět ukázkový rozvrh a jeho kódování.



Ilustrace 2.2: Ukázka, jak může vypadat rozvrh pro tři rozvrhy ve dvou skupinách. Vlevo: rozvrhy zdrojů. Vpravo: Výsledný genom.

Z pohledu kódování je v dynamickém systému možné rozdělit události, které jsou spjaty s touto dynamikou, do dvou tříd: události *částečné* a *totální*. Částečné události mění parametry zdrojů a aktivit, tedy například dobu zpracování. Nejde tedy o změnu na úrovni kódování chromozomu. Totální události naopak vyžadují překódování – může jít o ztrátu či přidání zdroje nebo celé úlohy.

2.5.1 Genetické operátory pro rozvrhovací chromozom

Genetické operátory jsou díky vhodnému návrhu kódování velice jednoduché. Mutace aktivitě změní pozici či id zdroje a křížení lze použít klasické jednobodové. Při těchto operacích ale může dojít k narušení unikátnosti pozicí aktivit v rozvrhu konkrétního zdroje, kdy se více aktivit ocitne na stejné pozici, nebo budou některé pozice scházet. Zejména při křížení je nutné pozice opravit tak, aby výsledný chromozom co nejlépe vyjadřoval vlastnosti získané z obou rodičů. Proto je v genech uchováván ještě druhý údaj o původní pozici. Při opravování chromozomu jsou pak procházeny aktivity pro každý rozvrh a jejich pozice jsou přečíslovány tak, aby byly unikátní a začínaly od 1. Při konfliktech je přihlíženo právě k této původní pozici – přednost má ten, který byl před operací ve svém původním rozvrhu plánovaný dříve. Pokud i původní pozice genů byla stejná, má přednost první z genů.

2.6 Analýza efektivnosti popsaného kódování

Popsané kódování pracuje s rozvrhy pro jednotlivé zdroje. To znamená, že aktivita, která alokuje více zdrojů, se objevuje v genomu vícekrát. Vzhledem ke způsobu, jakým pracují genetické operátory, může nastat případ, kdy bude při alokaci dvou zdrojů paralelními aktivitami jeden plánován pro první a následně druhou aktivitu a druhý v opačném pořadí. Čím více zdrojů alokují jednotlivé aktivity, tím větší je pak pravděpodobnost, že nastane zaseknutí na úrovni paralelních aktivit.

Oprava zaseknutí rozvrhů na úrovni aktivit

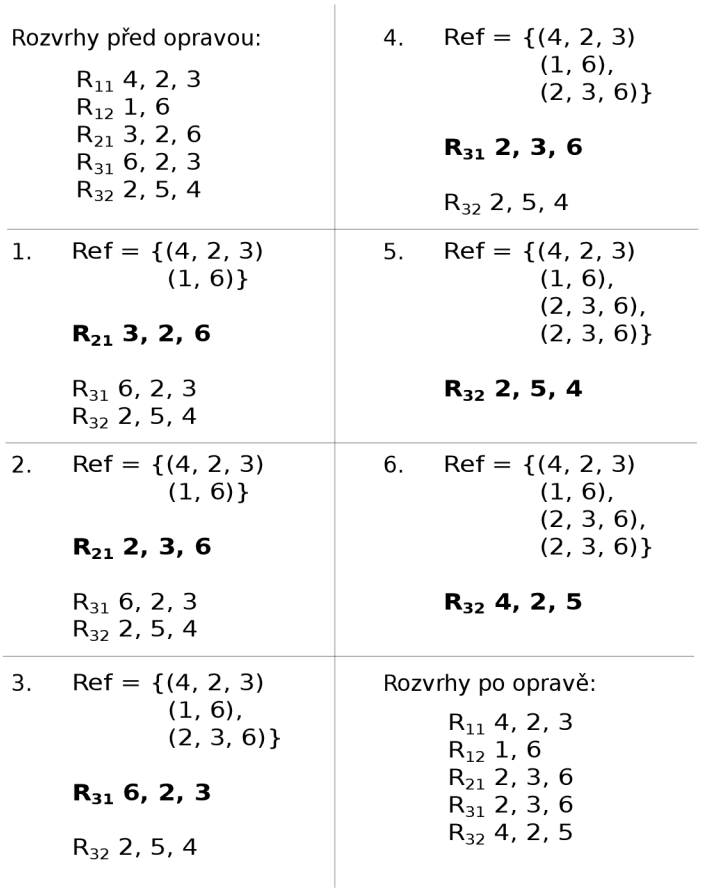
Rozvrhy získávané z GA je možné opravit tak, aby neobsahovaly tato zaseknutí. Opravný algoritmus využívá skutečnosti, že každá aktivita je v jedné skupině zdrojů pouze jednou. Algoritmus postupuje následovně:

1. Rozvrhy zdrojů v první skupině jsou umístěny do *referenční množiny* Ref. Pořadí aktivit v seznamech umístěných v Ref určují částečné uspořádání na nějaké podmnožině všech aktivit v systému.

2. Aktivity na zdrojích druhé skupiny jsou seřazeny podle částečného uspořádání definovaného seznamy v Ref. Poté jsou tyto opravené seznamy přidané do Ref.

3. Po průchodu všech skupin platí, že pokud je aktivita a před aktivitou b v některém z rozvrhů, je toto pořadí dodrženo i ve všech ostatních rozvrzích.

Tento algoritmus neopraví zaseknutí druhého typu, způsobené porušením precedence aktivit, ale i částečná redukce vadných kandidátů zlepší výkonnost genetického algoritmu. Ukázka běhu algoritmu je na ilustraci 2.3.



Ilustrace 2.3: Ukázka běhu algoritmu pro opravu zaseknutí na úrovni aktivit.

2.7 Volba parametrů GA

Při použití genetických algoritmů je vhodné přizpůsobit některé parametry řešenému problému. Základními parametry bývají velikost populace, počet generací a pravděpodobnost mutace. GA vždy konvergují při nekonečném počtu generací k optimálnímu výsledku, pokud je nenulová pravděpodobnost mutace a pokud je využito elitářství[1] [20]. To je ovšem pouze

[1]Elitářství: určitá část populace s nejlepšími fitness hodnotami je přímo převedena do další generace.

teoretická úvaha. Pro nalezení dostatečně dobrého výsledku je nutné určit velikost populace a počet generací tak, aby bylo výpočetně únosné tyto experimenty provádět. Odhad velikosti populace byl empiricky určen například v [2] jako hodnota v intervalu $< \log_2 N, 2log_2 N >$, kde N je velikost prohledávaného stavového prostoru, přičemž permutačních problémů je velikost stavového prostoru $n!$, kde n je počet genů.

Při řešení problému pomocí GA s kódováním popsaným výše je ale ve stavovém prostoru veliké množství nepřípustných řešení, která vedou k zaseknutí simulace. Je tedy nutné použít spíše větší populaci, aby při počátečním generování vznikla alespoň nějaká přípustná řešení, jejichž potomci mohou dále konvergovat k lepším výsledkům.

2.8 Fitness funkce

Obecně lze v optimalizačním procesu uvažovat více kritérií. V rozvrhovacích problémech lze uvést cíle jako *minimální čas dokončení všech úloh*, *minimální průměrné zpoždění úlohy* nebo v případě just-in-time[2] rozvrhování *minimální průměrnou odchylku od požadovaného termínu* [14]. Pro použití genetických algoritmů je vhodné vícekriteriální optimalizaci redukovat na jedinou fitness funkci. Pokud označíme množinu cílů jako $G = \{g_1, g_2, ..., g_n\}$, lze definovat souhrnnou fitness funkci pro ohodnocení kandidátního řešení jako:

$$fitness = \sum_{i=0}^{n} w_i g_i,$$

kde w_i je relativní důležitost cíle g_i vůči ostatním cílům. Při použití výše zmíněného kódování rozvrhu lze hledat například rozvrh s co nejmenší dobou trvání procesů nebo s co nejmenším zpožděním oproti danému termínu dodání, ale i pro ostatní typy cílů by bylo možné přidat ke genu aktivity ještě další údaj: časové zpoždění před započetím aktivity.

[2] Just-in-time se kromě zpoždění snaží minimalizovat i dobu uskladnění výrobků.

Kapitola 3

Objektově orientované Petriho sítě

Kombinaci Petriho sítí a objektové orientace nazýváme Objektově orientované Petriho sítě. Nejdříve bude stručně představen formalismus P/T *Petriho sítí* [17]. Objektově orientovaná varianta vychází z pokročilejších formalismů a tak následující sekce spíše slouží k seznámení se základními myšlenkami modelování a simulace Petriho sítí.

3.1 Petriho sítě

Petriho sítě jsou matematickým formalismem, použitelným pro studium systémů. Jsou vhodné pro modelování systémů, které lze charakterizovat jako asynchronní, distribuované, paralelní, nedeterministické nebo stochastické. Výhodou Petriho sítí je jejich srozumitelná a přehledná grafická reprezentace.

Petriho síť je složená z grafu N a počátečního značení M_0. Graf N je orientovaný, vážený a bipartitní. Obsahuje dva typy uzlů nazývané *místa* a *přechody*. Hrana může vést pouze od místa k přechodu, nebo od přechodu k místu. Místa mají definovanou maximální kapacitu značek. Do míst lze umístit značky a hrany mohou být značeny kladnými váhami. U hrany, která vede z místa do přechodu značí váha počet značek, které musí být v místě přítomny, aby byl přechod proveditelný. Provedení přechodu je však umožněno až při splnění této podmínky na všech hranách, které do něj vedou a také pokud mají výstupní místa přechodu dostatečnou volnou kapacitu. Provedením přechodu jsou značky ze vstupních míst vyjmuty. Váhy na hranách, které vedou z přechodu do místa, značí počet značek, které jsou do těchto míst po provedení přechodu vloženy. Nedeterminismus plyne z faktu, že v případě, kdy je možno provést více přechodů, není definováno pořadí, v jakém k tomu musí dojít.

3.2 Objektová orientace v Petriho sítích

Jak je ukázáno v [9], do Petriho sítí lze zavést objektovou orientaci. Výchozím principem jsou funkcionální Petriho sítě. Každá funkce je definována šablonou sítě včetně počátečního značení. Jsou vyčleněna speciální místa pro uložení argumentů a místo nazvané *return*, kam je uložena výsledná hodnota. Akce $y := f(x_1, x_2, ..., x_n)$ v přechodu sítě je vyhodnocena ve třech fázích:

1. Je provedena vstupní fáze volajícího přechodu, tedy jsou získány značky ze vstupních míst přechodu. Je vytvořena instance sítě funkce, doplněno počáteční značení a do patřičných míst sítě funkce jsou vloženy argumenty.

2. Instance sítě je spuštěna paralelně k volající síti a je simulována, dokud se nedostane do místa *return* výsledek.

3. Výsledné značky z místa *return* jsou uloženy do proměnné y, instance sítě je zrušena a následně je provedena výstupní část volajícího přechodu (doplnění značek do výstupních míst).

Stejně jako je síť funkce chápána jako šablona pro instanci funkce, lze třídu modelovat jako šablonu pro vytvoření objektu (instance třídy). Metody objektu jsou sítě, jejichž šablony jsou definované v rámci třídy. Instance sítí metod jsou vytvářeny v reakci na zprávu, kterou objekt obdržel. Místa v objektové síti modelují instanční proměnné a metody na ně tedy mohou navazovat své přechody. Přechody objektové sítě umožňují implicitní chování objektu. Lze vytvářet podtřídy tříd, předefinovávat v nich přechody a místa a přidávat nové metody, místa a přechody.

Značky v místech mohou být obecně jakékoli objekty (tedy i další OOPN sítě) a seznamy objektů. Váha na hraně je značena pomocí tzv. multimnožinového konstruktoru ve tvaru $\{n_1`\ x_1, n_1`\ x_2, ..., n_m`\ x_m\}$ kde n_i je kladné číslo či proměnná, která obsahuje kladné číslo, a x_i může být buď primitivní objekt (např. číslo či řetězec), proměnná nebo seznam $(e_1, e_2, ..., e_j)$, kde e_k může být primitivní objekt, proměnná nebo seznam. Multimnožinové konstruktory se používají i pro definici počátečního značení v místech.

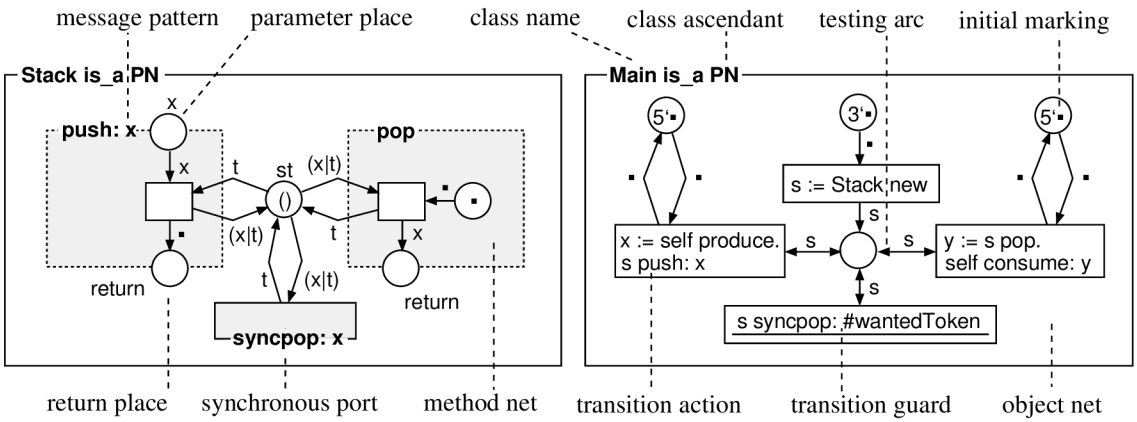
Pro přechod je definována *stráž*. Kromě nutnosti splnění typu objektů na vstupních hranách objektu je možné ve stráži specifikovat další podmínky, které mohou být buď výrazy složené z volání metod na primitivních objektech jako například $x > 0$, nebo může jít o testování *synchronního portu* libovolné sítě, které sémanticky odpovídá také splnění určité podmínky. Dále existují u přechodů *akce*, které mohou obsahovat zasílání zpráv objektům včetně přiřazování návratových hodnot do proměnných (ty lze pak použít na výstupních hranách).

Existuje také možnost použít tzv. *testovací* hranu, která stejně jako vstupní hrana vyžaduje přítomnost určitých značek ve vstupních místech a umožňuje navazování proměnných na tyto značky, ale značky z míst neodstraňuje. O testovací hraně lze uvažovat jako o oboustranné hraně a je tak také graficky znázorňována.

3.2.1 Synchronní porty

Synchronní port je prostředkem pro synchronizovanou interakci mezi objekty. Je podobný přechodu v tom, že má vstupní a výstupní podmínky. Pokud je port proveditelný, je při zpracování stráže proveden a podmínka je splněna. Speciální případ synchronního portu, který neovlivňuje stav objektu, se nazývá *predikát* a jeho variantou je *negativní predikát*, což je port, který je vyhodnocen jako pravdivý, pokud naopak provést nelze. Stejně jako metody mohou mít synchronní porty definované parametry, nejde ovšem o sítě a tak nejsou hodnoty těchto parametrů uloženy do žádných míst. Pokud je ovšem jako parametr předána volná proměnná, může dojít v rámci provedení přechodu k jejímu navázání na objekt z některého ze vstupních míst synchronního portu. Hlavní rozdíl mezi synchronním portem a metodou spočívá v tom, že synchronní port je použitelný pro testování podobně jako u vstupních podmínek – nejprve se otestuje, zda je proveditelný a až pokud všechny podmínky úspěšně projdou, je atomicky proveden.

Ilustrace 3.1 ukazuje všechny základní elementy OOPN.



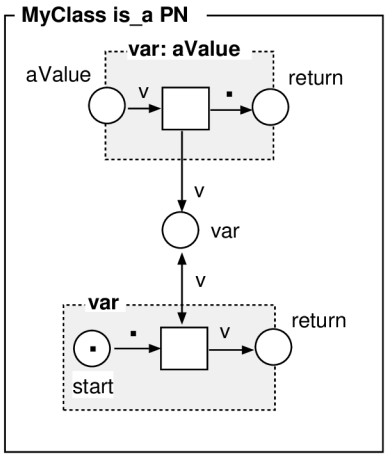
Ilustrace 3.1: Implementace zásobníku a ukázka jeho použití. Ilustrace demonstruje základní komponenty OOPN [13].

3.2.2 Základní konstrukce při vytváření tříd

Nyní bude představen způsob, jak lze modelovat v OOPN některé základní konstrukce, které se běžné objevují při vytváření tříd v objektově orientovaném programování.

Přístup k instančním proměnným

Mezi základní metody objektů patří obvykle metody přistupující k instančním proměnným. Toto platí i u OOPN – instanční proměnné jsou místa v sítích a občas je vhodné obsah těchto míst zpřístupnit pro jiné objekty. Ukázka nastavovacích a čtecích metod (anglicky *setter* a *getter*) je na ilustraci 3.2.

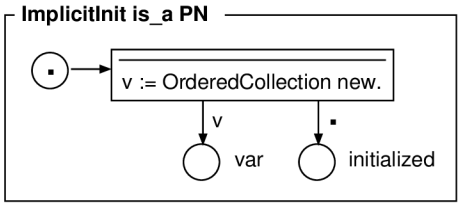


Ilustrace 3.2: Jednoduché metody pro nastavení a získání hodnoty instanční proměnné. Horní metoda je setter, spodní je getter.

Inicializace

Pokud je potřeba při vytváření instance třídy inicializovat obsah instančních proměnných na složitější objekty, než pouhé symboly či čísla, lze to provést pomocí přechodů v objektové

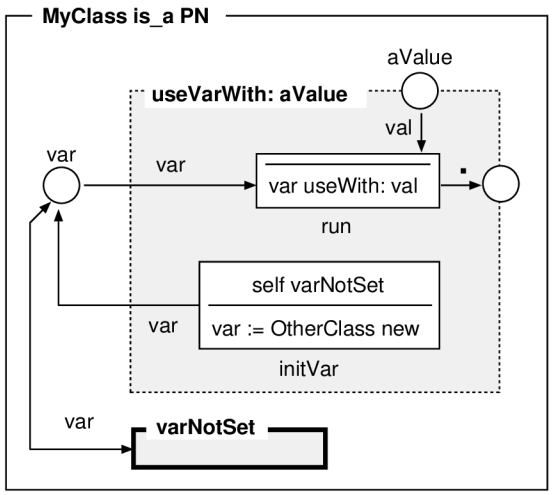
síti. Tento princip je ukázán na ilustraci 3.3.



Ilustrace 3.3: Implicitní inicializace instančních proměnných. Ostatní chování může být podmíněné buď přítomností objektu v místě `var`, pokud objekt potřebuje, nebo přítomností značky v místě `initialized`.

Odložená inicializace

Na principu odložené inicializace lze ukázat, jak se dá pracovat s negativními predikáty. Odložená inicializace vytváří potřebné objekty až ve chvíli, kdy jsou poprvé potřeba. Je tedy nutné zjistit, zda už objekt existuje a poté ho použít, nebo ho před použitím vytvořit. Metoda, která používá odloženou inicializaci, je zobrazena na ilustraci 3.4.



Ilustrace 3.4: Odložená inicializace. Přechod `run` je povolen až ve chvíli, kdy je k dispozici instanční proměnná `var`. Pokud tato k dispozici není, tak nejdříve proběhne přechod `initVar`. Port `varNotSet` je negativní predikát. Až po provedení `run` je naplněno místo `return` a metoda končí.

3.3 Nástroj PNtalk

Implementací výše popsaného formalismu OOPN je nástroj PNtalk, který byl představený v [10]. Nástroj je implementován v prostředí Squeak Smalltalk. Umožňuje modelování tříd a jejich metod pomocí vlastního jazyka a následnou simulaci takových modelů. Pokud mají objekty spíše programový než modelující charakter, dá se na simulaci pohlížet jako na prostředí, ve kterém tyto objekty žijí – simulace musí běžet, aby mohly objekty fungovat

a reagovat na zprávy. Nástroj umožňuje v případě potřeby synchronizaci simulace s reálným, popřípadě proporcionálním časem, nebo může běžet zcela bez synchronizace.

PNtalk je vyvíjen jako akademický nástroj a jako takový je z větší části neotestovaný, což znamená, že při modelování složitějších systémů je nutné spolupracovat s autory. Nevýhodou použití takového systému je možnost, že bude nutné čekat na opravu nějaké chyby, ale naopak výhodou je fakt, že pokud systém nově požadovanou funkcionalitu neobsahuje, existuje reálná možnost ji přidat.

3.3.1 Repozitář

Systém PNtalk využívá hierarchickou strukturu `MyRepository` pro uchovávání tříd. V této struktuře se objekty identifikují svou cestou podobnou běžným souborovým systémům. Třídy PNtalku mohou pracovat kromě Smalltalk tříd také se všemi třídami, které jsou ve stejné složce v repozitáři. Existuje také grafické uživatelské rozhraní pro práci s repozitářem, které mimo jiné i umožňuje inspekci aktuálně vytvořených simulačních experimentů.

Uživatelské rozhraní repozitáře umožňuje uživateli přidávat, odstraňovat, přejmenovávat, kopírovat a vkládat nové třídy. Stejné operace lze provádět s metodami tříd. U všech struktur, které jsou definovány jako sítě PNtalku, lze také otevřít grafický editor, jehož vývoj zatím není dokončen a lze jej zatím použít pouze k zobrazování sítí.

3.3.2 Jazyk PNtalk

Jazyk PNtalku umožňuje deklarativně popsat sítě tříd a jejich metod a v těchto sítích místa, přechody, synchronní porty a negativní predikáty. Jediný opravdu procedurální popis kódu, který se v PNtalku vyskytuje, je série akcí, které jsou provedeny před výstupní fází přechodu. Ilustrace 3.5 obsahuje ukázky kódu třídy a metody.

Každá síť je definována pomocí zdrojového kódu. Při uložení je tato textová podoba kompilována do objektů Smalltalku, které reprezentují třídy a jsou už přímo použitelné pro simulaci.

3.3.3 Simulátor

Systém PNtalk je zapouzdřený v komponentě jiného systému nazvaného SmallDEVS, který slouží k modelování dle formalismu DEVS a následným simulacím. Samotný PNtalk simulátor postupuje po atomických krocích. Jeden krok je vykonáním jedné akce z bloku `action` ve všech běžících přechodech sítě s tím, že první akce je vykonána zároveň s navázáním vstupních proměnných a naopak poslední akce je vykonána spolu s umístěním značek do výstupních míst. Simulaci je tedy možné zastavit i během vykonávání jednoho přechodu, s čímž je potřeba při úpravách simulace za běhu počítat.

Každá instance sítě (ať už jde o síť objektu či o jeho metodu) běží paralelně ve vlastním procesu. Komunikace těchto procesů je zprostředkována zasíláním zpráv. Žádné dva procesy tedy nemohou přímo ovlivňovat své stavy.

3.3.4 Metaprotokol

Pro dynamickou práci se sítěmi PNtalku je k dispozici metaprotokol popsaný v [12], který existuje na úrovni mezi Smalltalkem a PNtalkem a poskytuje třídy Smalltalku, které popisují a umožňují měnit běžící instance sítí vytvořených v PNtalku. Zde budou popsány některé třídy metaprotokolu.

```
class MyClass is_a PN
    place events()
    trans fireEvent
        precond events(1‘#e)
        action {
            Transcript show: ’event fired’.
            Transcript cr.
        }

    sync hasEvents
        cond events(1‘#e)

    inhibitor hasNoEvents
        cond events(1‘#e)

    method addEvent
        place start(1‘#e)
        place return()
        trans run
            precond start(1‘#e)
            postcond events(1‘#e), return(1‘#e)

class OtherClass
    place myCls()
    place start(1‘#e)
    trans initCls
        precond start(1‘#e)
        action {
            cls := MyClass new.
        }
        postcond myCls(1‘cls)

    trans addEvent
        cond myCls(1‘cls)
        guard {
            cls hasNoEvents.
        }
        action {
            cls addEvent.
        }
```

Ilustrace 3.5: Ukázka zdrojového kódu definujícího dvě třídy. První obsahuje synchronní port, negativní predikát a metodu. Druhá využívá negativní predikát instance první třídy ve stráži přechodu.

Třída PNCompiledClass

Každá instance třídy `PNCompiledClass` reprezentuje jednu třídu PNtalku. Každá třída obsahuje jednu instanci třídy `PNCompiledObjectNet`, popisující objektovou síť, která obsahuje implicitní chování objektu spolu se synchronními porty. Dále může třída obsahovat několik instancí `PNCompiledNet`, které popisují metody třídy. Protože PNtalk podporuje dědičnost, je ve třídě také uložená informace o její supertřídě.

Třída PNObject

Instance `PNObject` reprezentují OOPN objekty, které jsou instancemi OOPN tříd. Obsahují jednu instanci `PNProcess`, která reprezentuje instanci objektové sítě, a může dále obsahovat několik dalších instancí stejného typu, které jsou sítěmi aktuálně spuštěných metod objektu. Metaprotokol umožňuje přes tyto procesy přistupovat například k jednotlivým místům sítí a jejich obsahům.

Třída PNPlace

`PNPlace` popisuje místa v sítích OOPN. Objekty v místech jsou uloženy ve struktuře typu `Dictionary`. Samotné objekty jsou použity jako klíče slovníku a hodnota u každého klíče určuje kvantitu tohoto objektu v místě[1]. Slovníky spoléhají při ukládání a následnému vyhledávání objektů na jejich `hash` hodnotu. Protože změnou objektu může dojít ke změně `hash` hodnoty, je nutné objekt před jeho změnou z místa vyjmout pomocí metody `take:mult:` a po změně jej opětovně vložit pomocí `add:mult:`.

Proxy objekty

Objekty PNtalku spolu komunikují skrze zprávy, které jsou charakteristické pro svět simulace PNtalku. Aby mohly být metody objektů PNtalku volány i z vnějších objektů Smalltalku, existují proxy třídy, které dědí ze třídy `PNtalkProxy` a zařizují překlad volání metody ze Smalltalku na doménovou metodu PNtalku. Pokud nějaký vnější objekt požádá o instanci objektu PNtalku, dostane instanci `PNtalkObjectProxy`. Toto volání je odchyceno v objektu metodou `doesNotUnderstand:`, která vytvoří doménovou zprávu a zašle ji zapouzdřenému objektu. Po návratu z PNtalk metody deleguje proxy objekt výsledek volajícímu objektu. Volání metody PNtalku je tedy logicky blokující. Při zastavené simulaci zůstane tedy Smalltalk proces, ze kterého byla metoda volána, zastavený. Proxy objekt poskytuje pomocí metody `reify` přístup k instanci `PNObject`, čímž je umožněna manipulace objektu pomocí metaprotokolu. Kromě toho pracuje Proxy objekt i obráceně, tedy poskytuje objektům PNtalku volat zprávy na objektech Smalltalku.

3.4 Modelování procesů a zdrojů pomocí OOPN

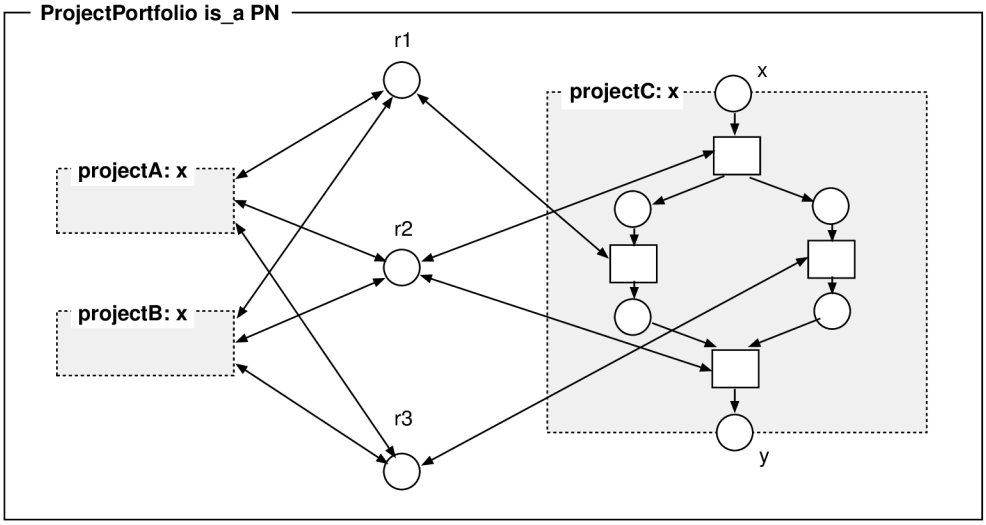
Základní koncepty modelování procesů a zdrojů jsou naznačeny v [13] a budou zde popsány a rozšířeny tak, aby bylo možné je implementovat. Sítě, které jsou v této kapitole uvedeny, jsou velice blízkou abstrakcí skutečně implementovaných sítí. Proto tato sekce slouží i jako programová dokumentace ke všem částem přiloženého systému, které jsou implementované v PNtalku. Z důvodu složitosti některých sítí jsou zde tyto zobrazeny ve zjednodušené

[1] Multimnožinový zápis $2`\#e$ je tedy v místě reprezentován jako $\#e \rightarrow 2$

formě, která neobsahuje všechny přechody a místa, ale přesto vyjadřuje všechny důležité aspekty chování.

3.4.1 Projektové portfolio

Projektovým portfoliem je myšlen soubor procesů (projektů, úloh), jejichž aktivity sdílejí zdroje. Portfolio je modelováno jako OOPN třída, jejíž metody popisují jednotlivé projekty a místa uchovávají objekty zdrojů. Spuštění projektu je provedeno zasláním patřičné zprávy objektu portfolia. V souladu s principy OOPN lze projekt spustit vícekrát a všechny projekty mohou běžet paralelně. Zde je vidět modelovací síla OOPN: nejde o jeden konkrétní model projektu, je totiž možné libovolně přidávat jakékoli metody projektů, které pouze využívají již přítomné zdroje. Portfolio obsahuje ještě několik dalších mechanismů, které budou popsány až v kontextu zdrojů s rozvrhem. Nástin základní sítě portfolia je na ilustraci 3.6.



Ilustrace 3.6: Základní princip projektového portfolia. Struktura projektů A a B je schovaná [13].

3.4.2 Projekty

Projekty[2] jsou modelovány jako OOPN metody třídy projektového portfolia. Jejich přechody reprezentují aktivity, které testují dostupnost potřebných zdrojů pomocí testovacích podmínek a stráží. Použití zdrojů začíná zjištěním, zda je konkrétní aktivita pro potřebné zdroje naplánovaná. K tomu jsou ideálním nástrojem synchronní porty. Po úspěšné alokaci dojde k čekání pomocí systémové metody `hold`, kterou znají všechny instance PN tříd. Tuto metodu lze volat explicitně v těle aktivity, nebo může být skrytá v metodě uvolnění zdroje – záleží na konkrétním typu zdroje. Nakonec je nutné všechny zdroje uvolnit zavoláním patřičných metod. Vzhledem ke způsobu, jakým jsou zdroje modelovány, je aktivita provedena ve chvíli, kdy jsou pro ni všechny zdroje úspěšně alokovány a následně dokončena po uplynutí určitého času (trvání aktivity). Metoda projektu může mít definované různé

[2]Termíny projekt a proces jsou zaměnitelné podle toho, jestli je na problém nahlíženo z pohledu projektového plánování nebo z plánování výrobních procesů.

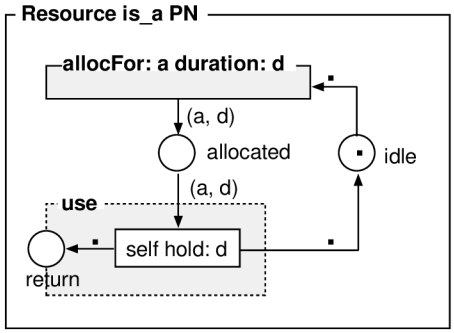
parametry, které mohou mít význam například pro optimalizaci rozvrhů. Příkladem takového parametru je termín požadovaného dokončení projektu. Návratová hodnota pak může obsahovat informaci o tom, zda se projekt stihl a pokud ne, tak jaké měl zpoždění.

3.4.3 Zdroje

Zdroje jsou třídy OOPN a jejich instance jsou v objektu projektového portfolia umístěny do míst. Každé takové místo v sobě drží všechny zdroje ze stejné skupiny zdrojů. Myšlenka, na které je systém zdrojů postaven, je taková, že každý zdroj je potřeba před použitím alokovat a po použití uvolnit. K alokaci se přímo nabízí využití synchronních portů, protože pokud zdroj nebude úspěšně alokován, tak aktivita jednoduše stojí a čeká na jeho uvolnění. Uvolnění zdroje může být poté provedeno klasicky metodou.

Základní zdroj

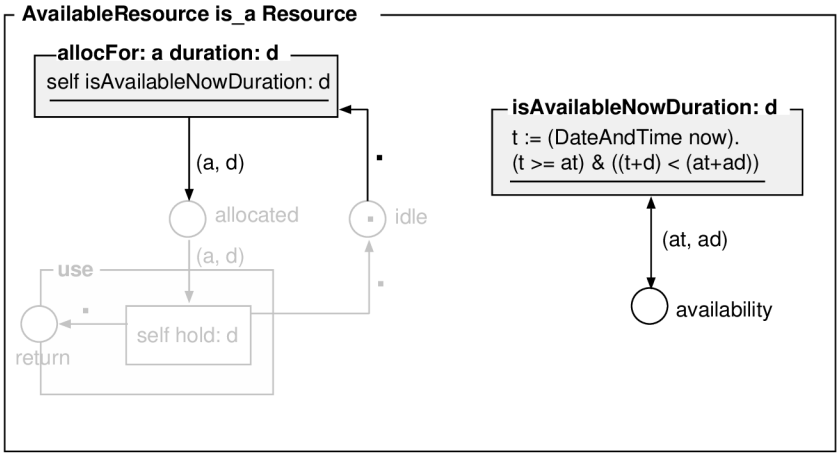
Koncept zdroje bude ukázán na jednoduché PN třídě, která je nazvána `Resource` a je zobrazena na ilustraci 3.7. Obsahuje synchronní port `allocFor:duration:` proveditelný pouze pokud je zdroj k dispozici (značka v místě `idle`). Zdroj si uloží do místa `allocated` dvojici s informací o tom, jaká aktivita ho chce využívat a na jak dlouho[3]. Dále zdroj definuje metodu `use`, která samotnou aktivitu vykoná, což je modelováno jako čekání pomocí volání metody `hold:`. Po uplynutí dané doby je zdroj použitelný další aktivitou. Zdroj, který je dostupný pouze v určitých časových intervalech, lze modelovat navázáním místa se seznamem těchto intervalů na synchronním portu `allocFor:duration:`, jak je vidět na ilustraci 3.8. Kromě této základní funkcionality obsahuje třída ještě důležitou metodu `initParent:name:`, která je volána ihned po vytvoření instance a slouží k nastavení odkazu na instanci portfolia a k nastavení názvu zdroje, který může sloužit k logování či ladění modelu.



Ilustrace 3.7: Základní model zdroje [13].

Takto implementovaný zdroj je přidělován aktivitám, které si o něj žádají, v pořadí požadavků (v případě shodného času požadavku je rozhodování nedeterministické). Pro účely rozvrhování je nutné třídu zdroje rozšířit o rozvrh. S tím souvisí nutnost odlišovat aktivity různých projektů. Dalším nedostatkem této třídy je skryté čekání uvnitř metody `use`, protože neumožňuje aktivitě efektivně využít více zdrojů najednou, aniž by docházelo k opakovanému sériovému čekání ve více zdrojích.

[3] V takto modelovaném systém určuje dobu zpracování aktivita, nikoli zdroj. Bylo by možné toto modelovat i jinak, například by doba zpracování mohla být spočtena z náročnosti aktivity a efektivity zdroje.



Ilustrace 3.8: Model zdroje s omezenou dostupností. Ve stráži portu `isAvailableNowDuration:` je použita deklarace proměnné pouze pro přehlednost – v kódu by se muselo volání `(DateAndTime now)` vložit přímo do složeného podmínkového výrazu, protože ve stráži nelze přiřazovat do proměnných [13].

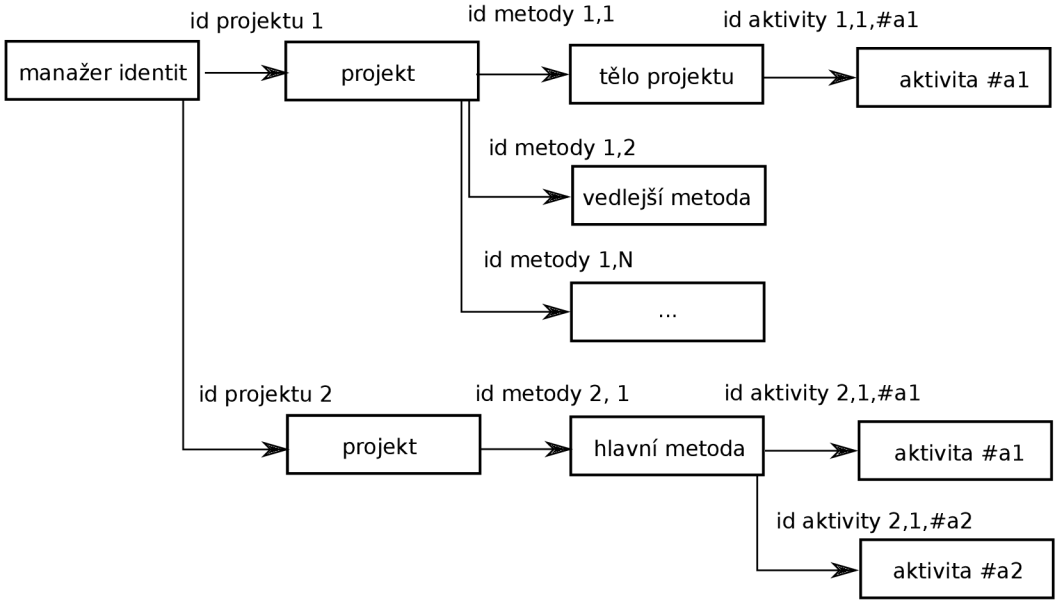
Identifikace aktivit

Protože může být spuštěno více instancí stejného projektu díky možnosti vícenásobného zavolání metody, je třeba odlišovat nejen jednotlivé aktivity, ale i do které instance projektu aktivita patří. Každý projekt tedy hned po inicializaci získá unikátní ID. Protože je u některých projektů žádoucí zanořovat se do dalších metod (například při potřebě vyrobit 4 identické součástky, než může hlavní projekt pokračovat), získá i každá takováto metoda včetně té hlavní ještě další unikátní identifikátor v rámci tohoto projektu. Každou aktivitu v systému je pak možné identifikovat pomocí id projektu, id metody v rámci projektu a id aktivity v rámci metody. Id projektu a metody je pro všechny aktivity konkrétní metody neměnné, spojuje se tedy do jednoho identifikátoru (dále *id metody*). Princip je zobrazen v ilustraci 3.9.

Zdroj s rozvrhem

Zdroj s rozvrhem je nazván `ScheduledResource`. Původní koncept zdroje nepočítal s id metod a proto musel být přepracován. Zde bude uvedena již tato přepracovaná verze. Zdroj musí umět dvě základní funkce: umožnění alokace pouze té aktivitě, která je jako příští v rozvrhu, a naopak možnost natrénovat rozvrh v případě, že není jisté, jaké aktivity si o zdroj budou při běhu říkat. Také je vhodné, aby přepínání těchto módů funkce bylo skryté z pohledu implementace jednotlivých metod projektů. Spolu s nutností identifikovat aktivity i podle id metody vzniká tedy nový alokační port `allocForObject:activity:`, jehož novým prvním parametrem je právě id metody a který ztrácí parametr `duration:`, protože čekání již nebude provádět implicitně zdroj. Port kontroluje veškeré náležitosti ohledně dostupnosti zdroje, ale samotnou kontrolu, zda je aktivita další v rozvrhu, zprostředkovává rodičovský objekt – instance projektového portfolia – pomocí synchronního portu `isScheduledObject:activity:inResource:`.

Tento port v projektovém portfoliu si vyžádá na vstupní hraně instanci plánovače z místa `scheduler`. Existují dvě varianty plánovačů, klasický a trénovací. Oba plánovače mají stejně nazvaný synchronní port `isScheduledObject:activity:inResource:`,



Ilustrace 3.9: Schéma přiřazování id k jednotlivým aktivitám běžících projektů.

který se ale chová rozdílně. U klasického plánovače se zavolá zpátky na zdroji z parametru `inResource:` synchronní port `isScheduledObject:activity:`, který už rovnou kontroluje, zda je aktivita naplánovaná v rozvrhu. U trénovacího plánovače synchronní port naplní místo `trainActivity` informacemi o žádající aktivitě. Z tohoto místa je pak tato informace vzata přechodem `train` a je vložena do trénovacího rozvrhu. Při trénování tedy alokace z pohledu rozvrhu není nikdy blokována a projekt proběhne naivně se zdroji přiřazovanými v tom pořadí, v jakém si o ně aktivity říkají. Právě aktivní plánovač je uložený v síti portfolia v místě `scheduler` a ten, který není používaný, je v místě `otherScheduler`. Při přepínání z jednoho módu do druhého jsou tyto plánovače vyměněny.

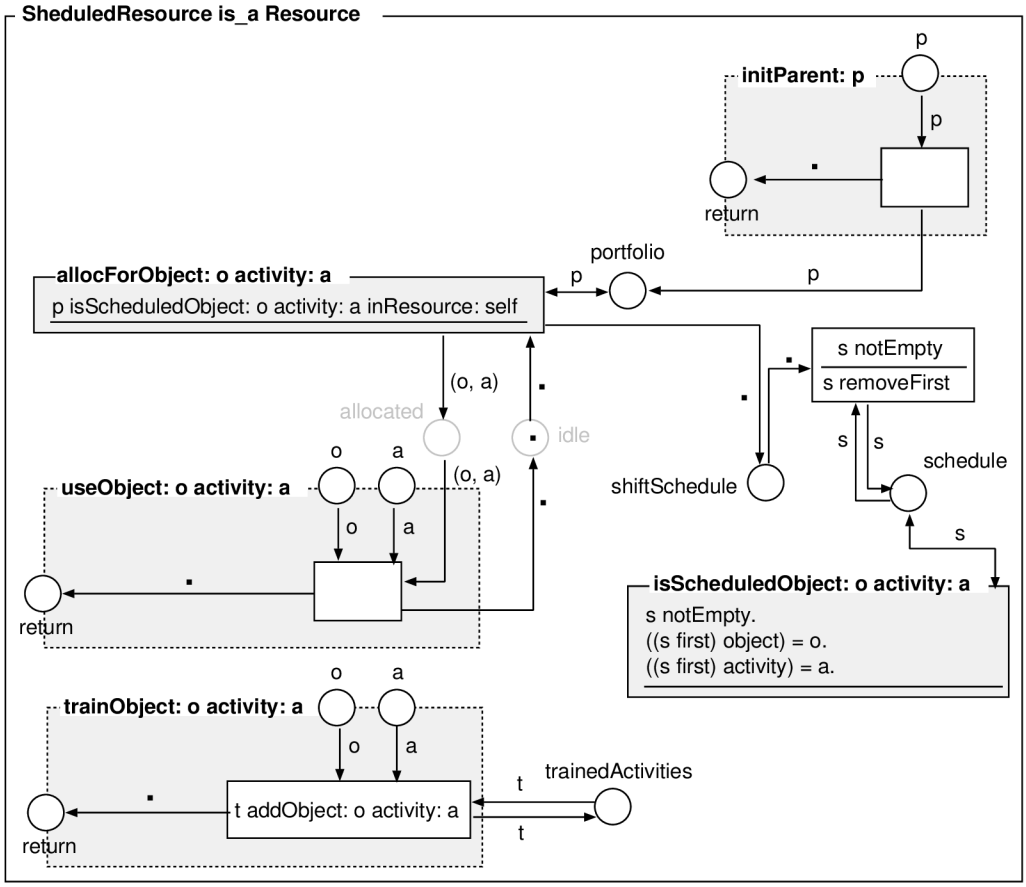
Zdroji je přidán rozvrh ve formě místa nazvaného `schedule`. Toto místo obsahuje kolekci typu `ScheduleCollection`, která dědí z `OrderedCollection` a přidává pouze jednu metodu sloužící k ukládání informací o aktivitě včetně id metody. Důvod pro vznik této jednoduché metody je zjednodušení kódu implementujícího trénování rozvrhů v PNtalk třídách. Koncept trénování dává smysl v kontextu optimalizace a je dále vysvětlen v kapitole 4.

Protože se alokace provádí v synchronních portech, kdy není možné vykonávat žádné akce, nelze přímo při alokaci zařídit odstranění aktivity z rozvrhu. Proto je při provedení portu umístěna značka do místa `shiftSchedule`. Ve zdroji je pak přechod, který při existenci této značky kolekci rozvrhu obslouží.

Uvolnění zdroje probíhá v metodě `useObject:activity:`, která nahrazuje původní `use`. To, že je opět předávaná aktivita, umožní zdroji zároveň obsluhovat více aktivit – při uvolnění jsou uvolněny ve zdroji pouze ty jeho části, které byly přiřazeny této konkrétní aktivitě. Z těla metody je též odstraněno čekání, o které se bude nyní starat přechod aktivity.

I když je aktivita plánovaná vždy pouze na jeden zdroj ze skupiny zdrojů, při testování přechodu jsou oproti synchronnímu portu alokace otestovány všechny zdroje v daném místě, takže pokud je aktivita na některém naplánovaná, je tento zdroj nalezen a použit. Ukázka zdrojů s rozvrhem je na ilustraci 3.10, plánovače jsou zobrazeny na ilustraci 3.11.

Rozvrh u zdroje může způsobit zaseknutí simulace, pokud poruší relaci precedence u ak-



Ilustrace 3.10: Model zdroje s rozvrhem. Port `isScheduledObject:activity:inResource:` buď volá obratem port `isScheduledObject:activity:`, nebo se stará o natrénování aktivit (podle toho, jaký plánovač zrovna portfolio používá). Metoda `initParent:` je ve skutečnosti implementovaná v rodičovské třídě, ale dává smysl až v kontextu rozvrhů, proto je zobrazena zde.

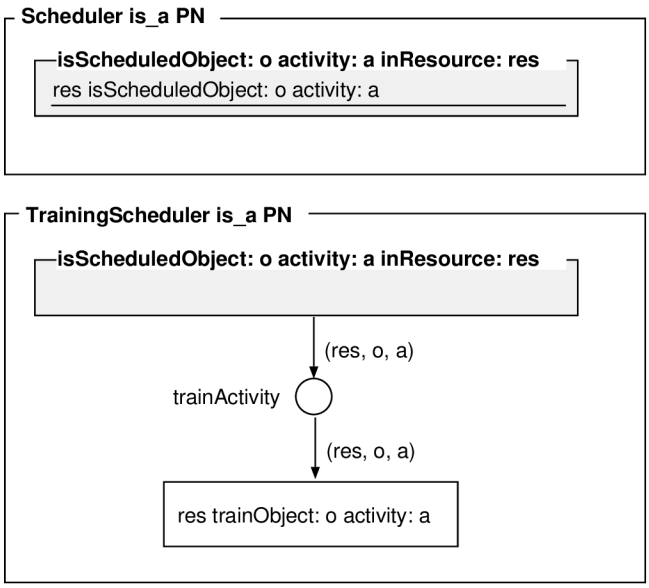
tivit. Zjištění zaseknutí bude je řešeno dále v sekci 4.1, ve vyhodnocení fitness je zaseknutí považováno za nejhorší možný případ a takový jedinec je z populace automaticky vyřazen.

3.5 Modelování portfolia

V této sekci bude nejprve popsán způsob, jak lze pomocí výše popsaných PNtalk tříd namodelovat konkrétní projektové portfolio. Poté budou představeny instance problémů typu RCPSP z knihovny PSPLIB a jejich automatizovatelný převod do PNtalku. Nakonec bude ukázáno, jak lze modelovat složitější systém inspirovaný reálnou výrobou.

3.5.1 Portfolio a zdroje konkrétního systému

Konkrétní portfolio vznikne jako podtřída třídy `ProjectPortfolio`. Musí obsahovat místa pro držení zdrojů a metodu `setup`, která instance zdrojů vytvoří a vloží je do těchto míst vloží při inicializaci zdroje. Dále definuje samotné metody reprezentující jednotlivé projekty, které v tomto portfoliu mohou být spuštěny.



Ilustrace 3.11: Třídy plánovačů. Nahoře je plánovač pro normální běh, dole plánovač pro trénování. Oba spolupracují s třídou `ScheduledResource` zobrazené na ilustraci 3.10.

Zdroje lze implementovat jako podtřídy třídy `ScheduledResource`. Pokud zdroj vyžaduje inicializaci nějakých vlastností, jako například nastavení maximální kapacity, obsahuje metodu ve stylu `initParent:name:param1:param2:...`, která vnitřně volá rodičovskou metodu `initParent:name:`. Alokační synchronní port je také specifický, pokud každá aktivita zdroj alokuje jiným způsobem (například obsazuje jen určitou kapacitu zdroje). Toto se promítne v portu přidáním parametrů: `allocForObject:activity:param1:param2:...`. Je možné a v mnoha případech nutné informaci o tom ukládat do místa `allocated` nebo nějaké jeho alternativy. Pokud je alokační port změněný, je potřeba změnit i tělo uvolňovací metody tak, aby zabrané části zdroje opět uvolnila. Jako ukázka slouží implementace zdroje s kapacitou popsaná dále.

3.5.2 Knihovna PSPLIB

Zkratka PSPLIB pochází z anglického Project Scheduling Problem LIBrary. Obsahuje instance různých variant RCPSP problémů, včetně protokolů o jejich optimálních či heuristických řešení včetně doby potřebné pro vyřešení. Instance problémů mají standardní tvar, který lze programově zpracovat. Kromě problémů v knihovně je zde k nalezení i program *ProGen*, který slouží ke generování těchto problémů. Knihovna i generující ProGen jsou dostupné na [1].

ProGen

ProGen je nástroj pro generování problémů různých variant typu RCPSP. Zde bude popsán formát generovaného výstupu u těch problémů, jejichž aktivity mají pouze jeden mód běhu[4] a tedy vyhovují definici SMRCPSP. Projekty tohoto obsahují jediný typ zdroje, a to obnovitelný zdroj s omezenou kapacitou.

[4]Vícemodální aktivity mohou mít definováno více rozdílných skupin zdrojů, na kterých mohou běžet – při běhu je vybráno, která z těchto skupin bude použita.

Výstup programu

Výstupní soubory jsou v jednoduchém textovém formátu, kde dílčí informace jsou odděleny řádky a hodnoty v řádcích potom libovolným počtem mezer. První řádek definuje počet aktivit A a počet zdrojů R. Druhý řádek postupně pro každý zdroj definuje jeho kapacitu. Každý další řádek potom popisuje jednu aktivitu. První hodnota určuje dobu trvání aktivity, dalších R hodnot postupně určuje, jakou kapacitu daného zdroje aktivita potřebuje (0 znamená, že zdroj není používán). Poté následuje číslo P určující, pro kolik aktivit je tato aktivita předchůdcem dle relace precedence. Nakonec je vypsáno P čísel identifikující tyto následné aktivity. První aktivita je vždy startovní a nevyžaduje žádné zdroje, pouze inicializuje první opravdové aktivity projektu. Poslední aktivita je sběrná a má podobný charakter jako první. Nástin formátu souboru je na ilustraci 3.12.

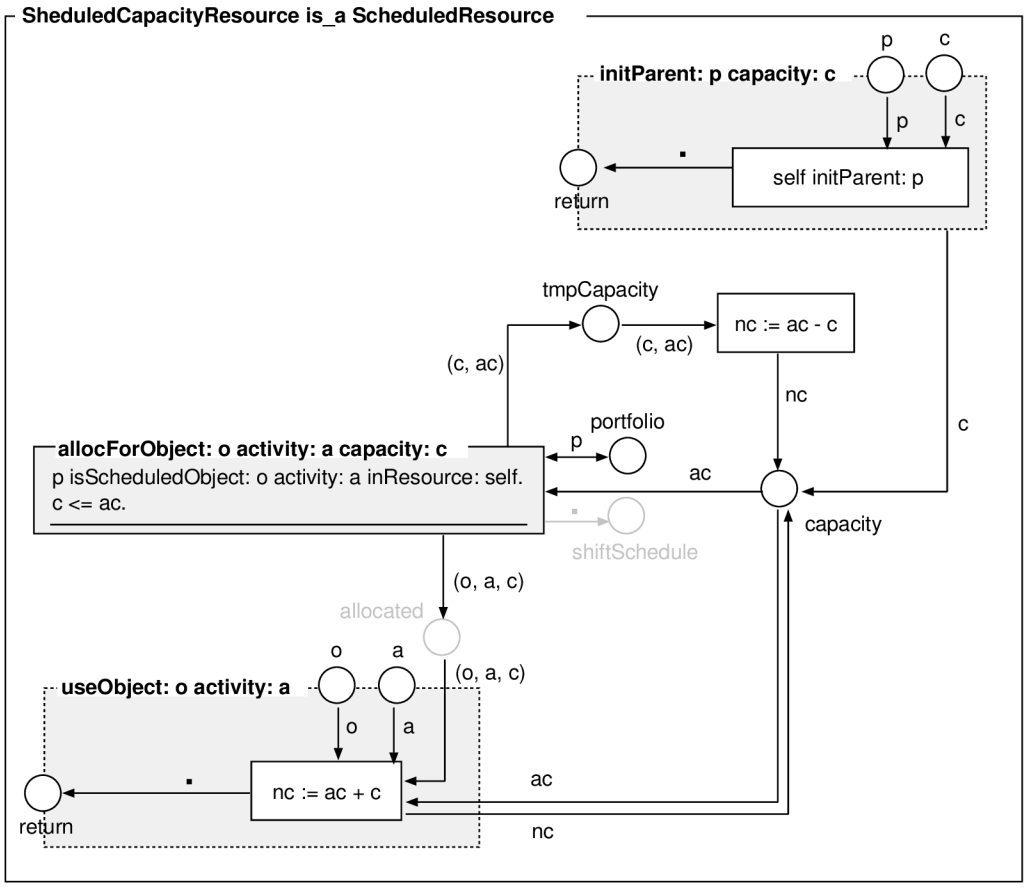
```
6       1
6
0       0       2       2       3
6       4       1       4
8       3       2       4       5
12      5       1       6
4       2       1       6
0       0       0
```

Ilustrace 3.12: Formát, v jakém jsou generovány problémy typu SMRCPSP programem ProGen. Zde je popis problému z ilustrace 2.1.

Zdroj s omezenou kapacitou

Instance tohoto zdroje má definovanou libovolnou kladnou číselnou kapacitu. Při alokaci zdroje aktivita určí, jaké množství prostředků bude po zdroji požadovat. Při uvolnění se alokované prostředky opět navrací zdroji k dalšímu použití.

První změnou v síti zdroje je přidání místa `capacity`, které drží informaci o aktuální kapacitě a nahrazuje funkci místa `idle` ze základního zdroje. Alokační synchronní port získává nový parametr a vzniká tak `allocForObject:activity:capacity:`. V portu se testuje, zda má zdroj požadovanou kapacitu – pokud ano, tak je kapacita ze místa odebrána a spolu s požadovanými prostředky uložena do místa `tmpCapacity`. Důvodem k tomuto postupu je nemožnost provádění akci v synchronním portu, takže je třeba po alokaci v normálním přechodu kapacitu snížit a vrátit novou hodnotu do místa `capacity`. Do místa `allocated` se ukládá kromě identity aktivity i informace o tom, kolik prostředků si aktivita zažádala, aby bylo možné kapacitu po vykonání aktivity obnovit. Tento proces je prováděný v upraveném těle metody `useObject:activity:`. Zdroj s kapacitou může využívat libovolné množství aktivit, přičemž limitem je pouze definovaná kapacita. Síť zdroje je naznačená na ilustraci 3.13.



Ilustrace 3.13: Třída zdroje s kapacitou. Jsou zobrazeny pouze změněné metody a porty, děděné informace jsou skryté.

Převod do PNtalku

Programový převod ze souborů tohoto formátů je přímočarý. Obsahuje šablony kódu, které tvoří kostru PNtalk třídy, potažmo jejich metod – jde například inicializační přechody nastavující id metody. V objektu projektového portfolia jsou vytvořena místa pro držení zdrojů pojmenovaná podle pořadí $r1, r2, ..., rR$. Metoda `setup` pro každý zdroj zavolá jeho inicializaci a vloží jej do patřičného místa.

Poté je vytvořen kód metody nazvané `projectWithName:withDueDate:withDeadline:`, která reprezentuje samotný projekt. Metoda obsahuje inicializaci id metody a ukládá startovní čas spolu s inicializačními parametry. Následně je dle informací o relaci precedence u jednotlivých aktivit vytvořena samotná síť. Každá aktivita až na sběrné (první a poslední) má vstupní a výstupní místo, přičemž do výstupního místa umísťuje vždy jednu značku `#e` na znamení svého dokončení. Podle toho, kolik má aktivita přímých předchůdců, je nastaveno, jaké množství značek se musí objevit ve vstupním místě, aby mohla aktivita začít. Tento přístup k řešení relace precedence je zvolen, protože varianta, kdy by každá aktivita měla své vlastní výstupní místo, obsahuje mnohem více míst a plní naprosto stejný účel. Síť totiž nikdy na vstup nedostane více než jednu značku – jiné spuštění stejného projektu vytváří novou instanci metody. Poslední aktivita při výstupu aktivuje přechod, který vygeneruje návratovou hodnotu projektu: seznam, který obsahuje informace o názvu pro-

jektu, čas jeho trvání a informace uvedené jako argumenty při volání metody. Zároveň uloží tyto informace i do místa `projectResults` v objektové síti portfolia. Průběh projektových aktivit je následující: aktivita získá zdroje, alokuje je, čeká danou dobu trvání, vypíše svůj název, zdroj uvolní a umístí značku `#e` do patřičných vstupních míst dalších aktivit.

Implementace převodního algoritmu

Převodní algoritmus je implementován ve třídě `SMCPLoader`, konkrétně ve třídní metodě `openAndLoad`. Po zavolání je otevřen dialog, ve kterém uživatel najde vstupní soubor. Poté je dotázán na požadovaný název PNtalk třídy. Po vytvoření potřebných zdrojových kódů tříd a metod je vše za pomocí metod tříd `MyRepository` a `PNCompiledClass` přidáno do systému PNtalk tříd a připraveno k použití.

3.5.3 Model inspirovaný reálným problémem

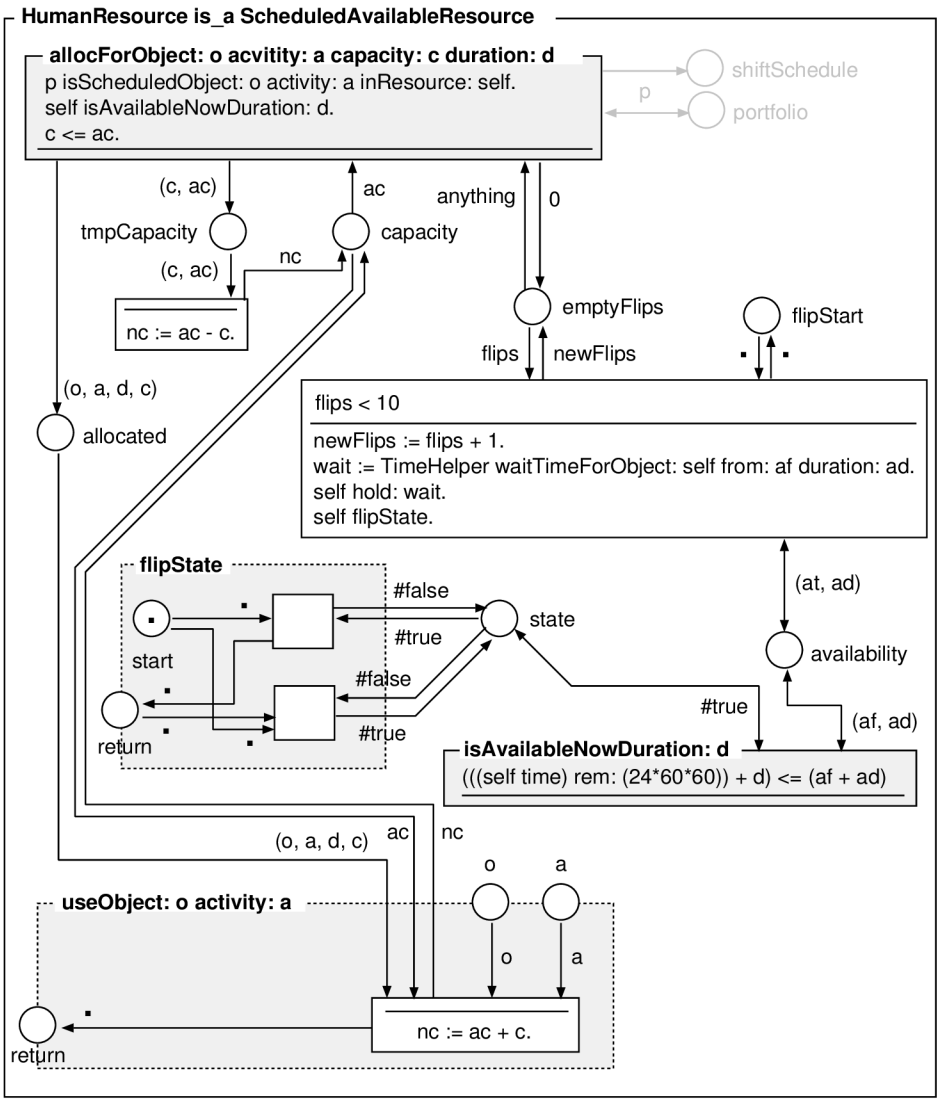
Nyní bude představen způsob, jakým lze vytvářet složitější modely na ukázkovém modelu výrobního systému, který je inspirován pracovními postupy reálného výrobního procesu součástek pro ofsetové tiskárny vyráběné firmou KBA-Grafitec s.r.o. V modelu se produkují dva různé výrobky: brzda archu papíru (dále *brzda archu*) a jednotka sloužící k osvětlení a k přítlaku archů ventilátory (dále *osvětlení*). Součástí jejich výrobního procesu je jak výroba základních součástek, tak následná montáž.

Jednotlivé meziprodukty, potřebné pro sestavení finálního výrobku, jsou rozděleny na aktivity. Každá aktivita má definované pracoviště, potřebné mezivýrobky a materiály, počet pracovníků, kteří na jsou při aktivitě přítomni a čas, jak dlouho aktivita trvá. Zde lze identifikovat zdroje v modelu: jde o pracovníky a pracoviště. Mezivýrobky jsou značky v konkrétních místech, umístěny na základě běhu dané části výrobního procesu. Všechny zmíněné zdroje budou odvozeny od dříve naznačeného modelu zdroje s rozvrhem, tedy od třídy `ScheduledResource`.

Lidé jsou modelování jako zdroje s intervaly a rozvrhem. Oproti výše popsanému konceptu zdroje s dostupností je zapotřebí změny – dostupnost lidí je totiž definovaná v intervalech během každého dne. V modelu lidského zdroje je také použit koncept zdroje s kapacitou, čímž je umožněna práce jednoho člověka na dvou různých aktivitách. Kapacita je reálné číslo v intervalu $(0, 1)$ a určuje, jakou část své pozornosti může pracovník momentálně poskytnout další aktivitě (1 značí, že pracovník momentálně nedělá nic, 0 znamená plné obsazení). Model lidského zdroje je označen jako *HumanResource* a je ukázán na ilustraci 3.14.

Většina pracovišť je definována jako již zmíněné zdroje s rozvrhem, u kterých však není dostupnost limitovaná časovými intervaly, protože stroje na těchto pracovištích jsou schopny pracovat kdykoli, pokud mají k dispozici lidskou obsluhu. Protože je kontrolováno obvykle více výrobků najednou, stejně jako materiály je možno kvůli úspoře ze skladu vydávat najednou pro více činností, je u aktivit, které vyžadují jako zdroj kontrolní stanoviště nebo výdej materiálu, obvykle modelována pouze poloviční potřebná kapacita lidského zdroje. Existují však ještě další odlišné zdroje, které budou nyní popsány.

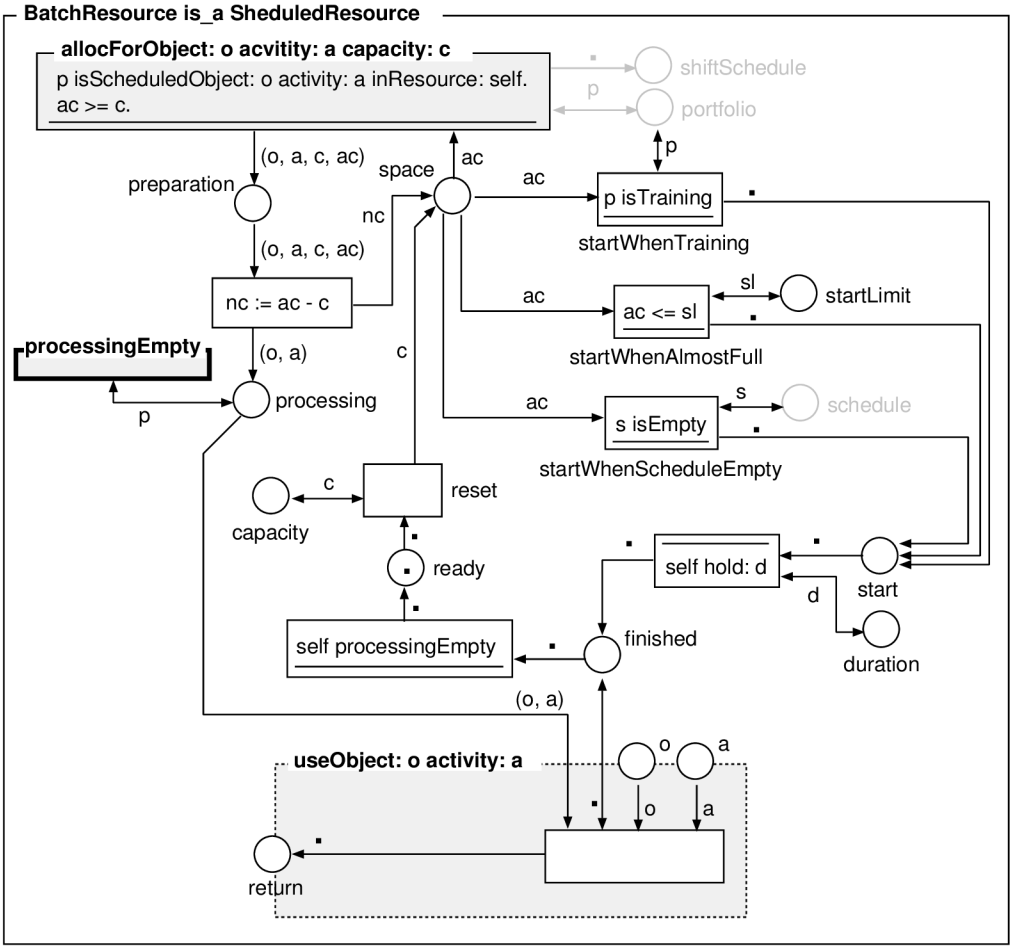
Některé typy povrchových úprav (např. alkalická oxidace) se provádí po *dávkách* součástek omezených maximální kapacitou zařízení a trvají pevně stanovenou dobu. Rozvrh zdroje určuje, v jakém pořadí se do něj vkládají součástky. Při dostatečném naplnění kapacity nebo při prázdném rozvrhu je proces spuštěn. Způsob alokace zdroje pro aktivitu je podobný jako u lidského zdroje – jen počáteční hodnota není 1, ale je třeba ji inicializovat podle parametrů pracoviště. Obsah metody `use` je upraven tak, aby reflektoval výše



Ilustrace 3.14: Model lidského zdroje.

zmíněný princip spouštění procesu, jak je zobrazeno na ilustraci 3.15. Důležitou změnou v konceptu tohoto zdroje je skrytí čekání do zdroje, protože o čekání je rozhodováno na základě obsahu zdroje, nikoli v aktivitě. Aktivita pak čeká na dokončení běhu zdroje při volání metody `useObject:activity:`. Tento typ zdroje bude dále označován jako dávkový – *BatchResource*.

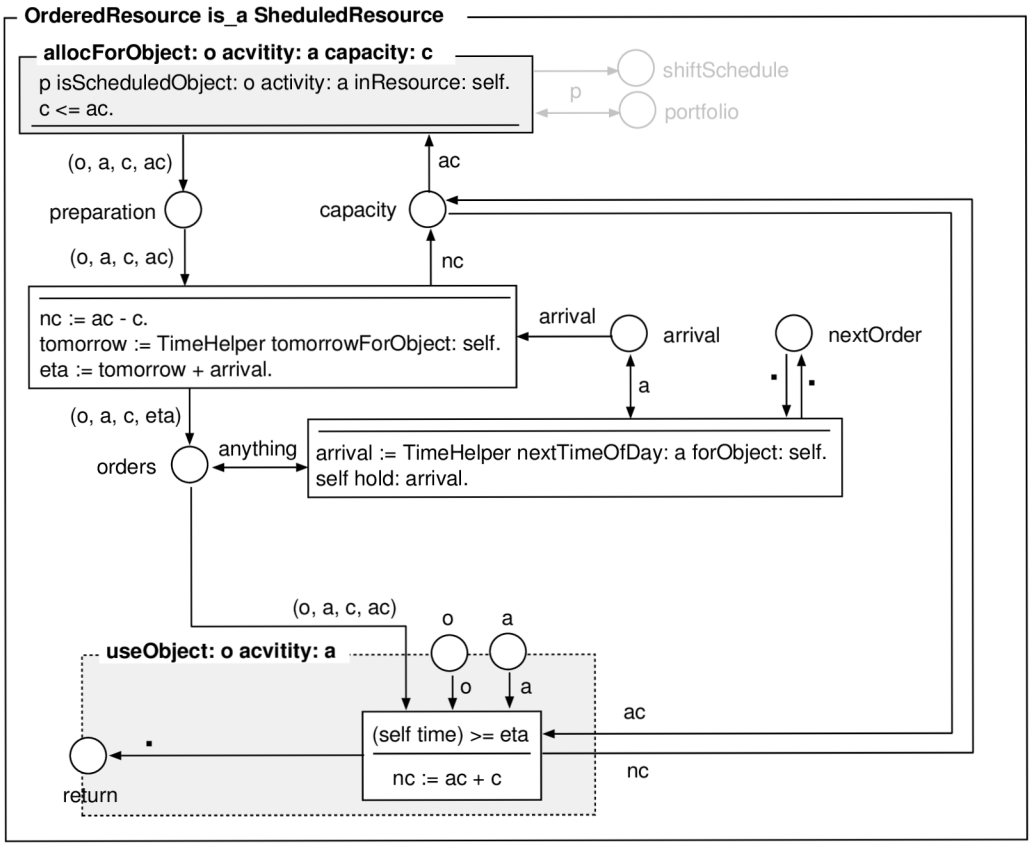
Některé aktivity, jako například řezání laserem, se provádí externě. To znamená, že ve chvíli, kdy se aktivita započne, je řezání objednáno a je dokončeno až druhý den v blíže specifikovanou dobu. Opět je zde místo *idle* nahrazenou kapacitou, která značí počet objednávek, které je poskytovatel schopen do dalšího dne dodat. Druhá změna je v metodě `useObject:activity:`, kde se již nečeká po nějakou dobu, ale na konkrétní okamžik v čase. Model externího zdroje je naznačen na ilustraci 3.16 a bude dále nazýván *OrderedResource*.



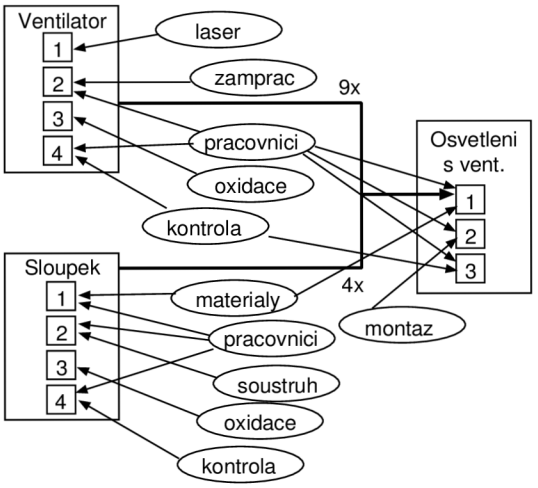
Ilustrace 3.15: Model dávkového zdroje. Objekt je třeba inicializovat pomocí zavolání metody `initParent:name:capacity:duration:`, která není na ilustraci zobrazena. Objekt vykazuje implicitní chování, kterým se vykonává samotné dávkové zpracování. Za povšimnutí stojí chování v přechodech označených jako `start...`, které obstarávají spuštění zpracování v různých situacích.

Modelované zdroje a pracovní postupy

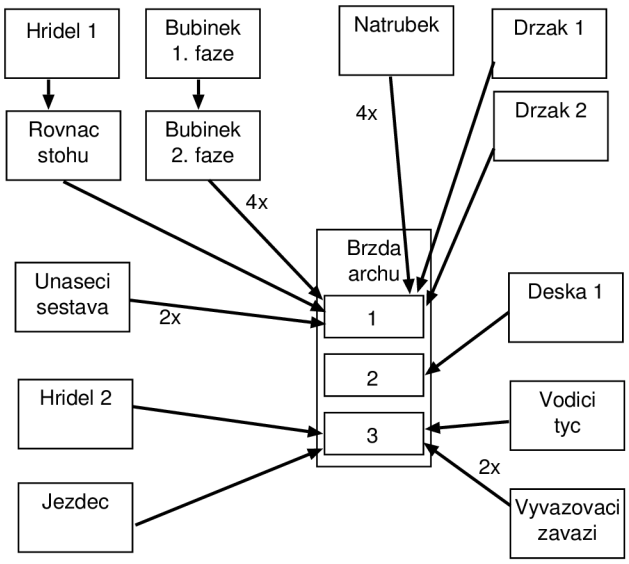
Pro přehlednost jsou všechny konkrétní typy zdrojů a kompletní pracovní postupy včetně podrobností uvedeny v příloze C. Ilustrace 3.17 a 3.18 ukazují abstraktní grafovou podobu modelů výrobních procesů pro oba modelované výrobky, nejde však o OOPN z úsporných důvodů. Na ilustraci 3.19 je ukázka části modelu výrobního procesu jednoho z výrobků v OOPN.



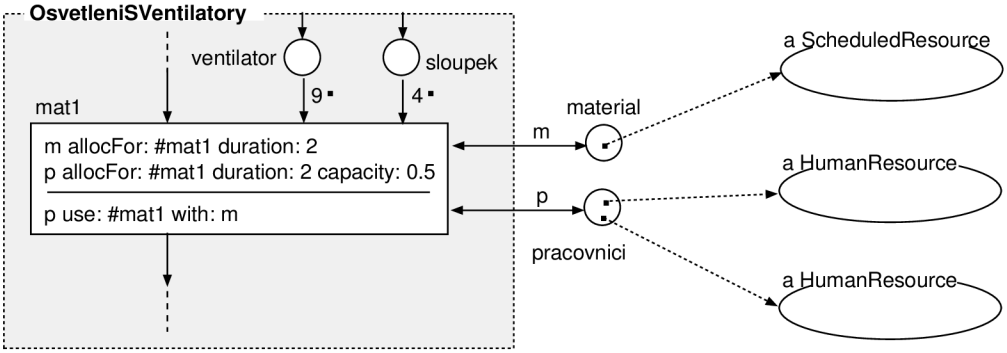
Ilustrace 3.16: Model externího zdroje (na objednávku)



Ilustrace 3.17: Ukázka závislosti jednotlivých fází výrobních procesů produktu *Osvětlení s ventilátory* na zdrojích a jiných fázích. Tenké čáry ukazují potřebné zdroje v k jednotlivým aktivitám, tlusté čáry představují závislosti na dříve vyrobených meziproduktech



Ilustrace 3.18: Ukázka závislosti fází montáže produktu *Brzda archu* na dalších meziproduktech. Pro přehlednost zde nejsou zobrazeny zdroje.

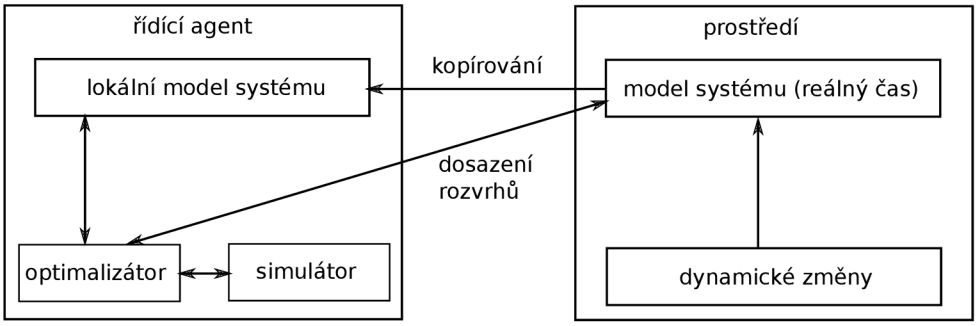


Ilustrace 3.19: Část OOPN modelu výrobku *Osvětlení s ventilátory*, která ukazuje použití více zdrojů a zároveň závislost na dokončení meziproduktů. Metoda `use: with:` je pouze paralelním voláním `use:` na obou zdrojích (aby nedocházelo k dvojitému čekání).

Kapitola 4

Návrh a implementace systému

Cílem této kapitoly je představit systém pro demonstraci dynamického rozvrhování a plánování. Systém obsahuje projektové portfolio implementované v PNtalku dle výše zmíněných principů, na kterém je spuštěno určité množství projektů a které je simulováno v reálném čase. Na tento model jsou napojeni dva autonomní agenti. První je označen jako *řídící systém* a plní výše popsanou optimalizační funkci. Druhý agent mění parametry simulovaného modelu a modeluje tedy dynamické chování reálného systému – tento agent je tedy součástí prostředí a řídícímu agentovi je skrytý. Celý systém až na model je implementovaný v prostředí Squeak Smalltalk za využití klasických Smalltalk tříd v balíčcích `RTScheduling-*`. Náčrt systému je na ilustraci 4.1.



Ilustrace 4.1: Nástin celého navrženého systému.

4.1 Balíček RTScheduling-System: Hlavní třídy systému

Tento balíček obsahuje obalující třídu systému – `RTSystem`. Ta obsahuje simulaci prostředí, agenta provádějícího dynamické změny, řídící systém a zároveň slouží k ovládání celého systému. U řídícího agenta zpřístupňuje nastavení fitness funkce pomocí metody `objectiveFunction:`, agentovi dynamických změn deleguje nastavení scénáře změn pomocí metody `playScenario:`. Také obsahuje metody na nastavení parametrů genetického algoritmu.

Třída `RTSimulationWrapper` slouží k obsluze simulace. Kromě ovládacích metod aktivně při běhu simulace v pravidelných intervalech sleduje, zda se simulace dostala do stavu, kdy není možné provést žádný přechod. To může znamenat buď situaci, kdy neběží žádný projekt, nebo kdy došlo k zaseknutí vlivem např. nepřípustných rozvrhů. Existuje

zde metoda `whenSuspendedDo:`, které je předán blok kódu, který má na zastavení simulace reagovat. Blok musí přijmout jediný parametr, který po zavolání obsahuje informace o běhu dokončených projektů v případě, že všechno proběhlo v pořádku, nebo symbol `#deadlock` pokud došlo k zaseknutí. Tento blok je důležitý při optimalizaci, používá se totiž k výpočtu fitness.

4.1.1 Změny modelu reálného prostředí

Součástí modelu prostředí je agent modelující zásahy do reálného systému, které řídící systém nepředvídá. K těmto zásahům je použito metaprotokolu PNtalku. Agent je implementován jako součást třídy *RTSimulationWrapper*. Zásahy do modelu mohou být několika typů, které zde budou rozepsány.

Změna rozvrhů

Navržený optimalizační proces využívá genetických algoritmů a nenalézá tedy zaručeně optimální řešení. Je proto žádoucí, aby měl způsobilý operátor systému možnost zasáhnout do sestavených rozvrhů. Toto je provedeno nahrazením značek reprezentujících rozvrhy na místech `scheduledActivities` v sítích zdrojů. Tato změna je odborným zásahem, který nevyžaduje další optimalizaci a není tedy třeba na ni v systému reagovat.

Přidání či odstranění zdroje

Přidání nového zdroje je pouze otázkou zavedení nového objektu do patřičného místa v síti projektového portfolia a odebrání z tohoto místa je stejně snadné. Je ale nutné ošetřit případ, kdy je zdroj momentálně používán nějakou aktivitou. Protože je model navržený jako nepreemptivní, je tuto situaci možné vyřešit zakázáním odnímání zdrojů ve chvíli, kdy jsou využívány.

Přidání a zrušení objednávky

Přidání objednávky je realizováno zavoláním patřičné met(šablony projektu) projektového portfolia. Zrušení objednávky sestává ze zrušení instance sítě projektu a z dealokace zdrojů, které měl projekt v době zrušení přiřazené. Dealokace je provedena pomocí metaprotokolu PNtalku a jde o vymazání značek z místa `allocated` a případné doplnění značky do místa `idle`, nebo alternativní zásah sloužící k vyčištění zdroje. Jelikož je tento proces velice specifický pro každý typ implementovaného zdroje a systém nemusí tušit, jak zdroje v projektu použité fungují, nebude možnost zrušení projektu v dynamických změnách použita.

Přidání a odstranění projektů

Sítě projektů spolu s potřebnými zdroji jednotlivých aktivit jsou výsledkem práce experta a jsou tedy pro zde představovaný systém předem definované. Řídící agent musí být schopný reagovat i na příchod nového *typu* projektu. Přidání typu projektu je přidáním nové metody, což může být zařízeno opět pomocí metaprotokolu (např. kompilací zdrojového kódu metody do běžící instance portfolia). Pro jednoduchost lze tuto operaci modelovat jednoduše tak, že projekty, které v systému nemají být, nebudou spuštěny – řídící systém sám od sebe metody nespouští, nebude pro něj tedy existence těchto metod nijak viditelná.

4.1.2 Realizace změn v modelu prostředí

Agent dynamických změn má předem definovaný scénář složený z dynamických změn a časových prodlev, po jakých mají tyto změny být provedeny. Simulátor samotný neobsahuje možnost napojit nějakou uživatelskou funkcionalitu na změnu v čase, proto musí být oznamování o dosažení určitého času simulace součástí modelu. Při nastavování scénáře pomocí metody `playScenario:` je na modelu portfolia zavolána pro každou plánovanou změnu PNtalk metoda `addChange:`, která přidá do místa `nextEvent` objekt typu `RTEvent`. Ten obsahuje údaje o časové prodlevě, jakou má model čekat před provedením akce, a samotnou akci ve formě bloku Smalltalk kódu, který dostává jako parametry model portfolia a objekt řízení simulace – může tedy libovolně se zastavenou simulací a s modelem pracovat. Ovládání simulace je po uplynutí dané doby informováno zavoláním metody `eventFired:at:` přičemž druhý argument poskytuje informaci o aktuálním čase v modelu. Simulace se v reakci na událost zastaví a následně je proveden kód změny.

4.1.3 Logování průběhu experimentu

Důležité události jsou ukládány do logu systému. Ten je realizován ve třídě `RTLogger` a uložené záznamy obsahují informace o reálném čase, o simulačním čase (pokud se záznam týká simulace) a samotný popis události. Z těchto logů lze později vyčíst informace potřebné pro vyhodnocení experimentu. Logovací objekt umožňuje filtrací a výpis události na `Transcript` a uložení logu do souboru.

4.2 Balíček RTScheduling-Scenarios: Scénáře a události

Tento balíček obsahuje dvě důležité třídy: `RTScenario`, tedy třídu scénáře, ze kterého se odvíjejí konkrétní scénáře pro modely, a `RTEvent`, jejíž instance jsou dynamické události v modelu. Každá událost je obsahuje informace o prodlevě před spuštěním, krátký popis a nakonec blok kódu, který bere jako parametry objekt modelu a objekt ovládání simulace. Na objektu modelu může provést potřebný zásah.

Každý scénář je instancí třídy `RTScenario`. Po inicializaci je nutné nastavit, pro jaký model je scénář definovaný pomocí metody `structClassName:`. Dále lze již přidávat jednotlivé události pomocí metod `at:describe:do:`, potažmo `at:do:`, pokud není třeba textový popis. Tyto metody se samy starají o vytváření instancí `RTEvent`.

Pokud je vhodné scénář uchovat pro opětovné použití, lze toho docílit jednoduše vytvořením podtřídy `RTScenario` a následné vytvoření instanční metody `setup`. Tato metoda je volána automaticky při inicializaci instance a může například obsahovat popis jednotlivých událostí.

4.3 Balíček RTScheduling-Control: Řídící systém

Samotný řídící systém má mít za úkol monitorovat reálný systém a dynamicky upravovat rozvrhy zdrojů se snahou o optimalizaci běhu systému na základě definované hodnotící funkce. Řídící agent musí být na výše popsaný model prostředí napojen takovým způsobem, aby mohl reagovat na případné změny a přepočítané rozvrhy pak vkládat na patřičná místa v síti modelu.

4.3.1 Vnitřní model prostředí

Řídící agent musí mít povědomí o reálném systému a jeho stavu, čehož je dosaženo udržováním vnitřního modelu monitorovaného systému. Tento model je inkrementálně upravován na základě změn v reálném systému díky využití metaprotokolu PNtalku. Pohyb značek po sítích může být sledován pomocí dotazů na obsahy míst. Při změně struktury v síti je tato změna promítnuta i do metaobjektů, které síť popisují a je tedy možné takovouto změnu zaregistrovat a replikovat ji ve vnitřním modelu systému. Agent má tedy v případě potřeby k dispozici aktuální stav reálného systému včetně aktivních rozvrhů.

Jelikož je už ale v systému zavedený mechanismus upozorňování na časy dynamických změn, je celý proces sledování stavu zbytečný. Mnohem jednodušší je totiž při dynamické změně zkopírovat celou simulaci, čímž je získán naprosto aktuální model.

4.3.2 Proces optimalizace

Při jakékoli z výše popsaných dynamických změn je nutné nebo přinejmenším vhodné přepočítat a nahradit aktuální rozvrhy zdrojů. Nejde vždy jen o optimalizaci, ale o nutný proces doplnění aktivit do rozvrhů, protože při většině změn se aktuální rozvrh stává nedostatečným. Příkladem je změna, která požaduje vytvoření nového projektu, jehož aktivity tedy v rozvrzích vůbec nefigurují.

Trénování rozvrhů

Při rozběhnutých projektech v různých fázích běhu je problematické zjistit, jaké jsou v běžícím systému aktivity a jaké zdroje si budou žádat. Nejjednodušším způsobem získání této informace je simulace, ve které nejsou aktivity omezené žádnými rozvrhy. Mechanismus získání takovýchto rozvrhů byl popsán v kapitole 3 v sekci 3.4. Po doběhnutí simulace lze tyto rozvrhy považovat za triviální řešení problému, které je vždy přípustné.

Optimalizační období

Nejprve je určena doba potřebná pro proběhnutí optimalizačních algoritmů. Ta se odvozuje od potřebného počtu evaluací kandidátních řešení a od doby, jakou trvá simulace jednoho běhu projektu s rozvrhem získaným při trénování.

Poté je vnitřní model systému simulován v simulačním (neproporcionálním) čase po tuto dobu a následně je simulace zastavena a stav vnitřního modelu uložen. Pro běh systému po určitou dobu je použit stejný mechanismus jako u dynamických změn – do modelu je nahrána událost s patřičnou prodlevou a po uplynutí je simulace v rámci reakce na modelem zaslanou zprávu zastavena. Tento zastavený model bude dále nazýván jako *optimalizační*. Na tomto novém modelu jsou znovu natrénovány rozvrhy, tentokrát již pro účely GA. Z tohoto plyne, že odhad doby běhu GA je nadsazený, protože rozvrhy pro systém nějakou dobu v budoucnosti může být menší, pokud během této doby nějaké aktivity proběhly.

Rozvrhy jednotlivých zdrojů jsou sesbírány a zakódovány do chromozomu dle popisu v sekci 2.5.

Samotný genetický algoritmus je implementován jako externí program napsaný v jazyce C++ za použití knihovny GAlib [22], který komunikuje se zbytkem systému přes klasický TCP socket. Externí aplikace bude popsaná dále v této sekci.

Po získání kandidátních řešení z genetického algoritmu je třeba je dekódovat a ohodnotit fitness funkcí. Agent podle chromozomu dosazuje aktivity do rozvrhů žádaných zdrojů

v nové kopii simulačního modelu. Následně se provádí simulace v simulačním čase. Po doběhnutí agent získá ze simulace data použitelná ve fitness funkci (doba běhu, zpoždění jednotlivých projektů, atd.) a ohodnotí kandidáta. Genetický algoritmus poté pokračuje.

Pro urychlení optimalizace se výsledky fitness k jednotlivým rozvrhům v průběhu jedné optimalizace uchovávají. Pokud přijde požadavek na výpočet již předem počítané fitness, je možné simulaci přeskočit a vrátit uloženou hodnotu. Po daném počtu generací je proces ukončen. Ve chvíli, kdy model prostředí dosáhne stejného času jako simulační model, je prostředí zastaveno, nejlepší kandidát je nahrán do modelu prostředí a následně optimalizovaný systém pokračuje v běhu.

4.3.3 Implementace řídícího systému

Balíček `RTScheduling-Control` zapouzdřuje dvě důležité třídy: `RTController`, což je samotný řídící systém a `GAConnector`, který zaopatřuje komunikaci s optimalizačním programem.

`RTController` slouží k samotnému řízení, čímž je myšlena optimalizace rozvrhů při dynamických změnách. Optimalizace začíná voláním metody `startAndConnectToOptimizer`. Tato spustí pomocí balíčku `CommandShell` optimalizační aplikaci a poté dá instrukce instanci `GAConnector`, aby otevřela naslouchací TCP socket na portu `31337`. Konektor po spojení s optimalizátorem zavolá na instanci `RTController` metodu `runOptimization`, která už provádí samotnou obsluhu optimalizátoru.

4.3.4 Protokol komunikace s optimalizátorem

Optimalizátor a řídící systém si vyměňují jednoduché zprávy v následujícím formátu:

```
!<4 znaky definující typ zprávy>#[velikost dat v bytech#data#]
```

Po spojení zasílá řídící systém zprávu typu `init`, která v datové části obsahuje parametry pro genetický algoritmus: počet generací, velikost populace a zda jde o minimalizaci či maximalizaci. Poté odesílá řízení další zprávu, tentokrát typu `base`, obsahující serializovaný natrénovaný rozvrh. Tento je použitý k inicializaci genetického algoritmu. Poté očekává řízení zprávy typu `eval`, které obsahují kandidátní rozvrhy. Na tyto rozvrhy odpovídá zprávou typu `fitn` s datovou částí obsahující výslednou fitness po odsimulování modelu. Jakmile obdrží zprávu typu `best`, ze které přečte nejlepší rozvrh, ukončí spojení a může pokračovat aplikací rozvrhů do míst v modelu prostředí. Po dokončení optimalizace je optimalizační model smazán.

Třída `GAConnector` slouží k udržení spojení a samotné komunikaci s optimalizátorem. Stará se o vytváření a čtení zpráv a informuje řízení o případných chybách ve spojení. Řízení se s chybami při spojení nebo s nedodržením protokolu ze strany optimalizátoru vypořádává zrušením celého optimalizačního sezení.

Balíček obsahuje ještě několik pomocných tříd:

- `ConnectorError` a `ProtocolError`, které slouží k odchytávání chyb v jednotlivých krocích řízení.
- `GASerializer`, který slouží k serializaci rozvrhů, které vytvoří kontrola při sběru z modelu a naopak k deserializaci kandidátů při výpočtu fitness funkce.

Formát při serializaci je také velice jednoduchý. Skládá se ze dvou částí. První definuje strukturu rozvrhů, tzn. kolik je typů zdrojů a kolik je zdrojů a aktivit pro každý typ. Jde

o seznam dvojic oddělených čárkou: `RN-AN`. Velikost tohoto seznamu určuje počet typů, `RN` je počet zdrojů pro daný typ a `AN` je počet aktivit. Druhá část už pak definuje natrénované rozvrhy jako seznam trojic reprezentující samotné geny ve tvaru: `TID-RID-POS`, kde `TID` je ID typu/skupiny zdrojů, `RID` je ID zdroje v dané skupině a `POS` je pozice v rozvrhu. Obě části jsou oddělené znakem `|`.

4.3.5 Externí optimalizátor

Optimalizátor byl implementován jako externí program v jazyce C++ za použití knihovny GAlib, která obsahuje všechny potřebné algoritmy a struktury potřebné pro definici a řešení výše definovaného optimalizačního problému rozvrhování. Po spuštění se program pokusí spojit s řízením pomocí klientského TCP socketu na port `31337`. Po ustavení komunikace získá natrénovaný rozvrh a použije jej ke generování počáteční populace.

Genom pro rozvrhování je implementovaný ve třídě `ScheduleGenome`. Samotné geny jsou naplněné struktury `tScheduleGene` obsahující informace o id skupiny zdroje, id zdroje, aktuální pozici a předchozí pozici (bližší popis v sekci 2.5).

Inicializace probíhá tak, že je pro velkou část populace proveden určitý počet mutací na tomto základním rozvrhu, u malé části se generuje náhodná permutace aktivit pro všechny skupiny a nakonec malé množství zůstává nezměněné, aby bylo v populaci zaručeně nějaké množství přípustných rozvrhů – optimalizace tedy končí přinejhorším triviálním přípustným řešením. Mutace je implementována jako vyměnění pozic a zdrojů dvou aktivit z jedné, náhodně zvolené skupiny zdrojů. Křížení je klasické jednobodové. Po mutaci i po křížení je chromozom opraven tak, aby byly pozice aktivit v rozvrzích unikátní.

Při výpočtu fitness je kandidát serializován a odeslán přes socket řídícímu agentovi. Po obdržení hodnoty fitness genetický algoritmus pokračuje, dokud neproběhne předem definovaný počet generací. Po odeslání nejlepšího kandidáta se optimalizátor ukončuje. Pokud dojde při komunikaci k nějaké chybě, dojde také k ukončení – řízení se samo postará o opětovné spuštění v případě potřeby.

Kapitola 5

Testování a vyhodnocení

V této kapitole bude popsáno testování implementovaného systému a projektových modelů. Dále bude porovnána efektivita popsaného způsobu optimalizace vůči již existujícím řešením instancí problémů typu RCPSP. Poté bude vyhodnocen celek co se týče použitelnosti při reálném nasazení a nakonec budou diskutována možná vylepšení a budou načrtnuty alternativní možnosti přístupu k celé problematice.

5.1 Simulace PNtalku

Prvním důležitým testem je zjištění rychlosti simulace navržených modelů v systému PNtalk a závislosti této rychlosti na počtu současně běžících projektů. Od této rychlosti se totiž přímo odvozuje doba potřebná pro proces optimalizace. Testování probíhalo jako opakované spouštění simulace modelu s narůstajícím počtem projektů. Běžící projekty byly identickými instancemi projektu ukázaného na ilustraci 2.1. Běh simulace probíhal na již předem dosazeném rozvrhu, aby model zaručené doběhl do konce všech projektů. Výsledný graf je na ilustraci 5.1.

Z tohoto grafu lze odvodit, že doba simulace roste exponenciálně vzhledem k počtu spuštěných projektů. To přeneseně odpovídá závislosti na počtu přechodů, u kterých je nutné v každém kroku simulace ověřit, zda jsou proveditelné. Jelikož testování synchronních portů je proces, který je těžko optimalizovatelný a protože navržené modely zdrojů jsou na synchronních portech silně založené, je tato závislost očekávatelná.

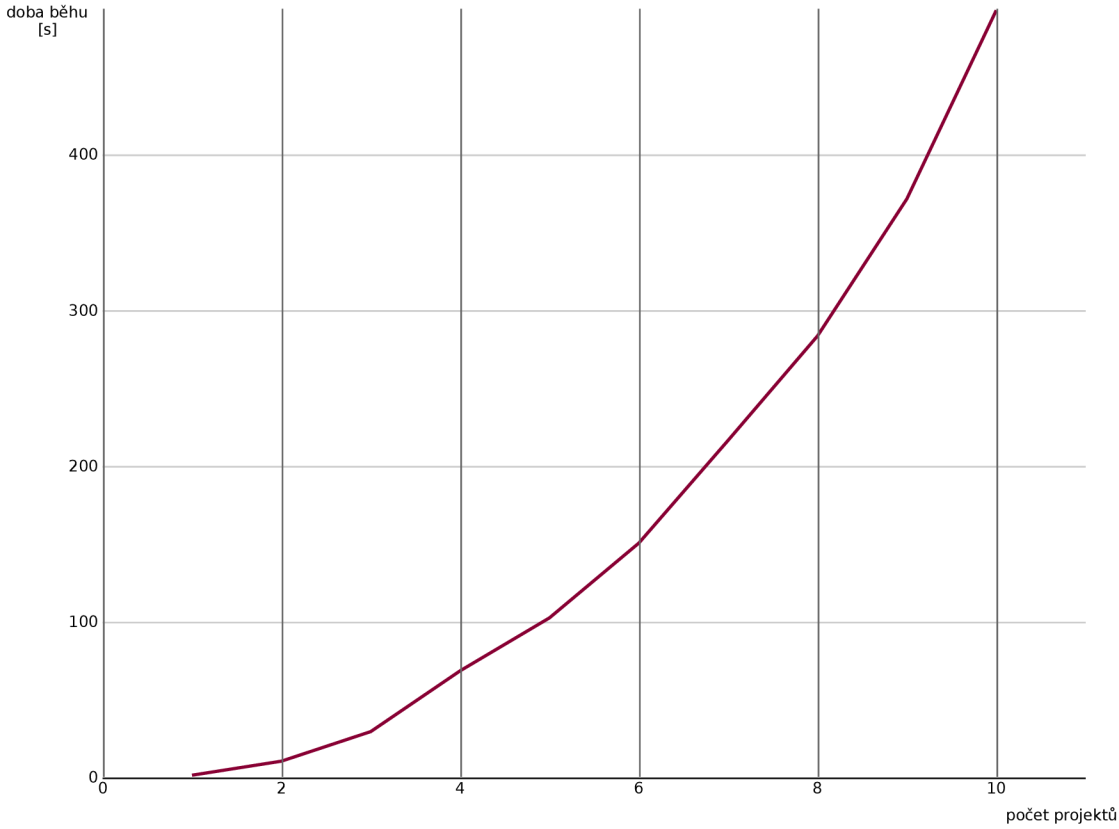
5.2 Efektivita optimalizačního procesu

Vzhledem k tomu, že i běh jedné instance projektu trvá na běžném počítači řádově sekundy, je jasné, že optimalizace využívající k vyšetření fitness simulaci PNtalku není příliš vhodná pro reálné použití. Zde budou provedeny experimenty, které mají toto tvrzení podložit.

5.2.1 Porovnání s referenčním řešením z knihovny PSPLIB

Pro srovnání byla spuštěna optimalizační úloha nad jedním základním projektem vzatým ze sady SMRCPSP úloh získané z knihovny PSPLIB. Šlo konkrétně o projekt *j305_4*[1], který byl nejdříve převeden do formátu generovaného programem ProGen a následně konvertován do OOPN postupem popsaným v sekci 3.5. Tento projekt dokáže referenční řešení z roku

[1] j305_4: Sada problémů j30, pátý problém, čtvrtá instance.



Ilustrace 5.1: Závislost doby běhu simulace modelu na počtu současně spuštěných projektů.

1995 na počítači s procesorem taktovaným na 25MHz vyřešit optimálně za 34.63s [6]. Zde navržený optimalizační proces řeší stejný problém na procesoru o taktu 2GHz 75 minut[2]. Řešení navíc není optimální – na nalezení optima by bylo potřeba velikost populace a počet generací ještě zvýšit a trvání by se tedy ještě prodloužilo.

Z logu bylo také zjištěno, že ze všech kandidátních řešení, u kterých GA vyhodnocoval fitness, bylo pouhých 9 přípustných a 212 nepřípustných. Nepřípustná řešení plynou z porušení relace precedence při mutacích a křížení. Tato vada bude diskutována v závěru v rámci shrnutí možného budoucího vývoje projektu. Parametry a výsledek experimentu jsou uvedeny v tabulce 5.1, log experimentu je na přiloženém mediu, viz přílohu A.

U problémů z knihovny PSPLIB není specifikováno mapování doby trvání aktivit na reálný čas, protože jde o problémy teoretického rázu. Přesto porovnání doby trvání optimalizace vede k závěru, že zdaleka nejde o efektivní proces.

Výhodou použití OOPN oproti klasickým řešením problému z PSPLIB je naopak možnost pustit více projektů najednou, projekty ani nemusí být spuštěny ve stejný čas, nebo může jít úplně o jiné projekty, které pouze sdílí zdroje. Navržený systém je mnohem obecnější a tím pádem poskytuje více možností při modelování optimalizovatelného systému.

[2]Může ale jít až o dobu delší než jeden den v závislosti na diverzitě a velikosti populace, což přispívá ke snížení efektivity vyrovnávací paměti fitness hodnot, ale zároveň zvyšuje šance na nalezení lepších řešení.

třída portfolia	SMCP54
třída scénáře	SMCP54OneRun
velikost populace	100
počet generací	10
fitness funkce	minimalizace času dokončení všech projektů
trvání při naivním rozvrhu	106
trvání při nalezeném rozvrhu	82
trvání při optimálním řešení	63

Tabulka 5.1: Popis experimentu s projektem ze sady SMRCPSP problémů z knihovny PSPLIB (pátý problém, čtvrtá instance).

5.2.2 Model inspirovaný reálnou výrobou

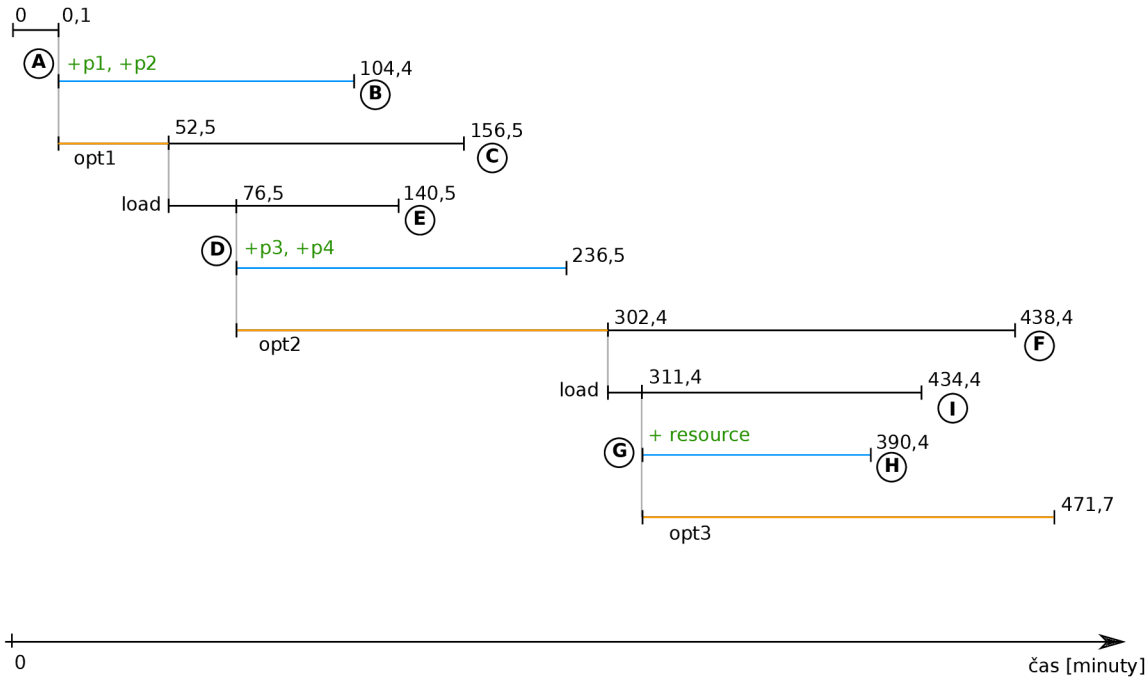
U modelu inspirovaného reálnou výrobou byly získány doby trvání jednotlivých aktivit v reálném čase, je tedy teoreticky možné ověřit použitelnost navrženého systému v prostředí bližšímu realitě. Bohužel, i menší část modelu, popisující výrobu osvětlení s ventilátory (více viz příloha C), není možné odsimulovat v rozumném čase. Simulace v simulačním čase jednoho běhu trénování rozvrhů na tomto modelu byla po čtyřech hodinách ukončena z důvodu zaseknutí z důvodu chyby v PNtalku. Dá se očekávat, že i kdyby byl dosazený validní rozvrh a tato chyba nenastala, tak běh modelu s rozvrhem by trval ještě déle, protože pokud není model v trénovacím režimu, musí ověřovat i rozvrhy a vede tedy k složitějšímu testování proveditelnosti přechodů. V PNtalku tedy lze korektně definovat složitější modely zdrojů, ale jejich simulace již není příliš prakticky proveditelná.

Nízká rychlost simulace modelu plyne z jeho vysoké složitosti. Modely zdrojů, které jsou zde používány, jsou výrazně sofistikovanější, než jednoduché zdroje s kapacitou používané v problémech typu SMRCPSP. Nízká výpočetní síla simulace OOPN je tedy vyvážena vysokou vyjadřovací schopností.

5.2.3 Demonstrace scénáře s více událostmi

V tomto experimentu byl spuštěn model portfolia s jednoduchým projektem ukázaným v sekci 2.3. Pro systém v prostředí byla doba trvání uvažována v minutách. Scénář obsahuje následující tři události, přičemž první dvě spouští v časovém rozestupu dva a dva projekty a třetí poté přidává nový zdroj. Při návrhu projektu byla vytvořena následující hypotéza: Zdroj je kapacitou natolik omezený, že běh prvních dvou projektů bude trvat déle, než doba potřebná pro optimalizaci rozvrhů při přidání dvou dalších projektů. Při druhé události tedy bude spuštěna optimalizace pro aktivity více než dvou projektů – půjde o celé dva nové projekty spolu s několika nedoběhlými aktivitami z prvních dvou projektů. Krátce po dokončení druhé optimalizace je přidán nový zdroj, přičemž první je zahlcen natolik, že opět optimalizace bude trvat kratší dobu, než trvá doběhnutí projektů. Po třetí optimalizaci již bude zbývat relativně málo aktivit, což spolu s přidaným zdrojem umožní rychlé dokončení projektů.

Průběh experimentu s popisem způsobu jeho zobrazení je na ilustraci 5.2. Z vývoje je jasné, že optimalizace je natolik pomalá, že by bylo výhodnější nechat projekty běžet s naivním rozvrhem. Z rozdílu bodů **C** a **E**, potažmo **F** a **I** je vidět, že optimalizace je funkční, tedy nalézá lepší, než naivní řešení, protože po optimalizaci je doba dokončení kratší. Po přidání nového zdroje je odhad doby potřebné pro optimalizaci delší, než odhad zbývají-



Ilustrace 5.2: Vizualizovaný průběh experimentu demonstrujícího scénář s více událostmi. Čas je na ose X, posun na ose Y směrem dolů značí vývoj systému na základě událostí. Osa X není lineární, pro znázornění je totiž důležitá pouze následnost událostí. Zelený text vždy značí novou dynamickou událost, *+pN* je zavolání projektu a *+resource* je přidání nového zdroje. V bodě **A** jsou přidány dva projekty. Modrá čára značí, jak dlouho by projekty běžely při použití naivního rozvrhu – první dva projekty by tedy skončily v bodě **B**. Oranžová čára značí dobu vymezenou pro optimalizaci a její černé pokračování je pouze orientační a ukazuje, jak dlouho by projekty běžely s naivním rozvrhem po dokončení optimalizace. Průběhy značené *load* jsou po nahrání rozvrhů, které vyšly z optimalizace. Bod **E** ukazuje, jak dlouho by projekty běžely po nahrání nového rozvrhu, pokud by nedošlo k žádné události. Bod **D** opět přidává dva projekty. V bodě **G** je přidán nový zdroj.

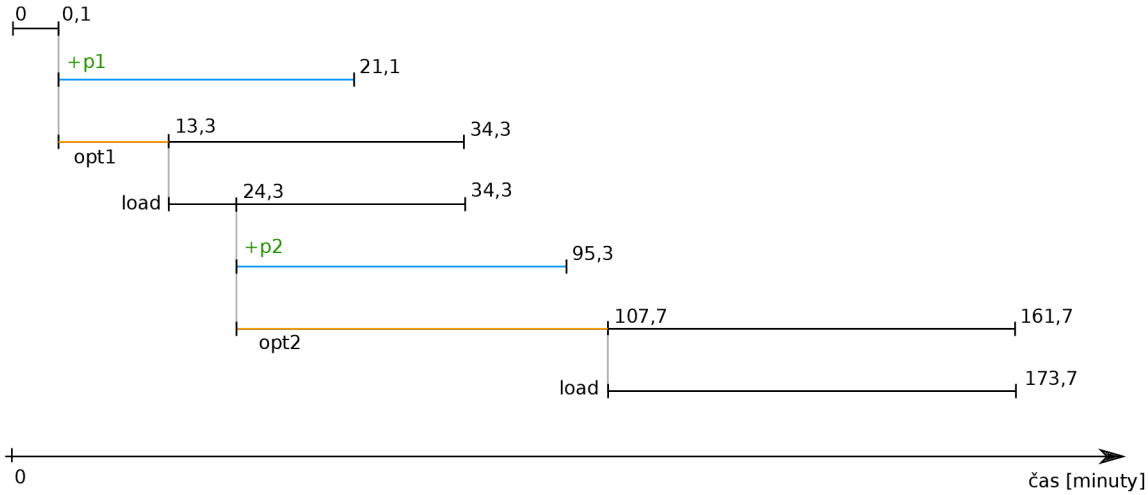
cího času pro doběhnutí celého modelu prostředí. Obě hodnoty jsou opravdu jen odhady, protože při nahrávání rozvrhů, potažmo při jiné manipulaci s modely, se simulační čas, který může být synchronizovaný s reálným časem, posouvá stále dopředu. Ve skutečnosti v tomto případě po simulačním období zůstaly v systému nějaké nedoběhnuté aktivity, ale bylo jich extrémně málo a tato poslední optimalizace tedy nebyla vůbec podstatná. Za povšimnutí stojí fakt, že naivní rozvrh po přidání nového zdroje dopadl opět mnohem lépe, než optimalizované řešení.

Z poznatků o rostoucí době potřebné pro dokončení simulace PNtalku v závislosti na počtu paralelních projektech a z doby potřebné pro simulaci i tohoto jednoduchého projektu plyne, že každý složitější projekt bude ještě více časově náročný.

5.2.4 Demonstrace portfolia s více projekty

Cílem tohoto experimentu je prezentovat, jakým způsobem lze v jednom systému implementovat více typů procesů, které sdílí pouze zdroje. Portfolio je kombinací první z instancí, které generuje program ProGen a instance problému typu SMRCPSP z knihovny PSPLIB

s pojmenováním j305_4. Každá z instancí je implementovaná ve vlastní metodě. Zdroje mají kapacitu nastavenou na hodnotu nižší, než je součet příslušných kapacit v obou instancích, aby bylo docíleno ještě více omezeného prostředí. Scénář spočívá ve dvou událostech. Spuštění jednoho z projektů proběhne ihned na začátku, druhý projekt je spuštěn chvíli po dokončení optimalizace rozvrhů pro první projekt. Průběh je znázorněn na ilustraci 5.3 a lze z něj vyčíst stejné poznatky jako z předešlého experimentu: optimalizace je vysoce neefektivní. V tomto případě navíc optimalizační proces nenajde ani v jednom případě lepší než triviální řešení.



Ilustrace 5.3: Průběh druhého experimentu. Obě optimalizace dopadnou stejně jako kdyby byl použit naivní rozvrh a jde tedy pouze o zdržení vývoje celého systému. U druhé optimalizace se neshodují časy odhadu dokončení při použití naivního rozvrhu a při použití výsledku optimalizace. Tento časový posun vznikl v následku manipulace s kopírovanými modely, protože ve skutečnosti jde o stejný rozvrh.

5.3 Problémy při experimentech

Protože je PNtalk experimentální akademický nástroj, který se neustále vyvíjí a není řádně otestovaný, mohou při spouštění simulací nastat různé problémy. Některé podstatné části systému, například kopírování zastavené simulace, byly vyvíjeny až paralelně s touto prací. Velká část problémů byla odladěna, ale i tak občas při operaci s PNtalk objekty dojde k nějaké chybě. Tyto chyby nastávají náhodně, protože k nim dochází v závislosti na stavu simulace při kopírování, což je závislé na reálném čase potřebném pro průběh jedné simulace. Tento čas je pokaždé jiný, protože je závislý na zdrojích, které výpočtu simulace Squeak přidělí. Při pádu projektu je nutné celý experiment spustit znovu.

Nejčastěji se vyskytují následující dva problémy:

1. Při kopii během zavolané PNtalk metody občas dojde k tomu, že metoda je volána znovu. To způsobí například u `useObject:activity:` zaseknutí, protože zdroj už o aktivitě neví a nemůže se tedy uvolnit.

2. V některých případech se simulace zastaví při čekání pomocí metody `hold:`, přičemž se stále jeví jako aktivní. To způsobí nedoběhnutí projektů.

Kapitola 6

Závěr

Cílem této diplomové práce bylo ověřit koncept, který popisoval způsob použití OOPN a nástroje PNtalk pro modelování, simulaci a optimalizaci procesů s omezenými zdroji. Tento koncept měl být dále rozšířen tak, aby bylo možné jej použít pro dynamickou optimalizaci běžících projektových portfolií a následně jej implementovat, otestovat a vyhodnotit výsledky. V rámci práce byl koncept rozšířen o některé zásadní prvky, které implementaci umožnily. Mezi tyto doplněné mechanismy patří například unikátní identifikace spuštěných procesů nebo změny v implementaci dříve navržených zdrojů tak, aby mohly být reálně implementovány v nástroji PNtalk. Také bylo nutné navrhnout způsob, jakým bude do modelů v PNtalku pomocí metaprotokolu zasahováno a jak bude zjištěno, zda simulace doběhla v pořádku, nebo jestli došlo k zaseknutí.

Byla představena formální definice rozvrhovacího problému s omezenými zdroji a ukázán princip, jakým lze instance tohoto problému převádět do popsaného systému projektových portfolií modelovaných pomocí OOPN. Dále byl popsán algoritmus, který dokáže instance těchto problému generované programem ProGen převádět automaticky do tříd nástroje PNtalk. Zároveň byl popsán model procesu, který se inspiruje reálnými výrobními procesy. V tomto modelu jsou rozpracovány různé další způsoby modelování typů zdrojů, které se v praxi objevují.

Je zde rozebráno chování řídícího agenta, který reálný systém monitoruje a následně se na základě změn v tomto systému snaží dynamicky přiřazovat rozvrhy aktivit procesů k jednotlivým zdrojům. K optimalizaci těchto rozvrhů je využito genetických algoritmů, přičemž je také navržen a způsob, jakým jsou rozvrhy kódovány včetně opravného mechanismu, který snižuje frekvenci zaseknutí simulací odstraněním křížových závislostí u aktivit alokujících více než jeden zdroj.

Popsaný systém byl implementován v prostředí Squeak Smalltalk, přičemž o OOPN části systému se stará nástroj PNtalk. Modely PNtalk objektů musely být navrženy tak, aby bylo možné s nimi při zastavené simulaci pracovat pomocí metaprotokolu PNtalku způsobem, který dovolí dále v simulaci pokračovat. Protože je PNtalk experimentální systém, který není řádně otestovaný a dosud v něm nebyly implementovány složitější modely, muselo být během této implementace ve spolupráci s vývojáři PNtalku odladěno velké množství chyb a nástroj musel být také rozšířen o novou funkcionalitu. Přesto výsledná implementace není bezchybná a často kvůli nějakým zbylým chybám není schopna dokončit běh.

Během implementace vyvstal problém související s nízkou rychlostí simulace implementovaných modelů. To způsobilo příliš dlouhou dobu optimalizace, což obratem vedlo k závěru, že použití simulace sítí pomocí PNtalku pro výpočet fitness funkce při dynamické optimalizaci je v současném stavu PNtalku nevhodné. Nízká rychlost vedla mimo jiné k ex-

perimentům, jejichž výsledky ukazují na exponenciální růst doby simulace při narůstajícím počtu paralelně spuštěných procesů. Přes chybovost systému byl úspěšně proveden experiment, který ukazuje vývoj systému při předem připraveném scénáři dynamických událostí. Tento experiment však pouze znovu potvrdil, že koncept není vhodný pro dynamickou optimalizaci v reálném čase.

Tato práce má i přes nepříliš příznivé závěry experimentů pozitivní výsledky. Bylo ukázáno, že pomocí OOPN lze modelovat i složitější zdroje než pouhé zdroje s omezenou kapacitou a tedy že nástroj PNtalk lze použít pro modelování nad rámec akademických příkladů. OOPN jsou navíc matematickým formalismem, který je přehledný díky vizuální reprezentaci sítí a modelování je tedy matematicky čisté a zároveň intuitivní. Dále bylo také ověřeno, že metaprotokol PNtalku lze jednoduše využít k dynamickým zásahům do běžících PN-objektů. Přínos spočívá také v ladění a vývoji PNtalku, což umožní v budoucnosti efektivnější využití tohoto nástroje a poznatky o nízké rychlosti simulace mohou vést k optimalizaci simulátoru. Tato práce je také prvním pokusem o vytvoření složitějších modelů a jejich použití ve spolupráci se Smalltalk objekty, který nebyl vytvořen autory PNtalku v rámci jeho vývoje. Může tedy mimo jiné posloužit i jako startovní místo pro nové projekty, které budou s PNtalkem pracovat.

Další vývoj a možnosti vylepšení

Nyní budou uvedeny koncepty, které již nejsou v rámci této práce rozpracovány, ale ukazují, jakým směrem by se ubírat další vývoj.

Statická analýza – relace precedence

Největším problémem při řešení rozvrhovacích problémů při zde navrženém kódování je velké množství nepřípustných řešení, která generuje genetický algoritmus. Tato řešení vznikají hlavně kvůli faktu, že GA si není vědom relace precedence aktivit a tím pádem ji při mutacích a křížení porušuje. K výraznému vylepšení by tedy mohla vést statická analýza sítí procesů a následné zakódování relace precedence pro účely genetických algoritmů, nebo do opravného algoritmu kandidátů popsaného v sekci 2.3. Pokud budou genetické operátory tuto relaci respektovat, všechna kandidátní řešení budou přípustná.

Alternativní optimalizační nástroje

Genetické algoritmy nejsou jediným optimalizačním nástrojem použitelným pro řešení rozvrhovacích problémů. Existují mnohé metody, ať už heuristické (např. různé varianty evolučních algoritmů) či optimální (např. lineární programování). Souhrn těchto metod je například v [3]. Existují i hybridní řešení, která kombinují oba přístupy. Jako příklad hybridního přístupu může sloužit kombinace lineárního programování a neuronových sítí v [7].

Rychlejší výpočet fitness

Nejužším hrdlem výkonnosti celého navrženého systému je pomalá simulace OOPN sítí. Nejschůdnějším řešením tohoto problému by byla implementace výpočtu trvání procesů rychlejším způsobem, avšak takovým, který by stále umožňoval dynamické zásahy za běhu systému, aby nebyla omezená síla navrženého modelu. Jako dobrý nástroj pro toto se jeví například nástroj SmallDEVS, který je implementaci formalismu DEVS [11]. Jako takový pracuje s hierarchicky zapouzdřenými komponentami, které reagují s určitým chováním na

vstupní zprávy. Není tedy třeba testovat proveditelnost přechodů složitým způsobem, jak je tomu v OOPN. Takový koncept by ale bylo nutné rozpracovat, což není předmětem této práce.

Jiným řešením by bylo využití paralelního výpočtu fitness, což by sice nevyřešilo neefektivitu simulace PNtalku, ale díky hrubé síle výpočetního výkonu by to umožnilo optimalizovat mnohem rychleji. Tento přístup by vyžadoval úpravy v externím optimalizátoru, který by musel o optimalizaci žádat více běžících Squeak Smalltalk prostředí, což by také znamenalo nutnost předání zastavené optimalizační simulace do jiné instance Squeaku. Běh více simulací za účelem zjištění fitness v jedné instanci Squeaku není efektivní, protože Squeak nyní nepodporuje využití vícejádrových architektur procesorů – nyní i paralelní procesy Squeaku běží reálně v jednom vlákně hostujícího operačního systému.

Lepší odhad doby trvání optimalizace

Při zvolení efektivnějšího způsobu odhadu trvání optimalizace by systém mohl čekat menší dobu, než mu mohou být dosazeny nové rozvrhy. Pro efektivitu optimalizace je nejlepší, když běží na co nejmladším systému, protože takový systém obsahuje více aktivit a tím pádem je větší šance nalézt nějakou kvalitní permutaci. Větší počet aktivit ale znamená pomalejší optimalizaci, což vede k většímu potřebnému času pro optimalizaci, což zase kvůli delšímu běhu systému před optimalizací způsobí uběhnutí více aktivit před začátkem optimalizovaného běhu a tím přímo zmenšuje velikost chromozomu, se kterým se pracuje. Nalezení rovnováhy je nový optimalizační problém.

Dynamický genetický algoritmus

Pro dynamické úlohy byla rozpracována řada variant genetických algoritmů. Tyto algoritmy ale počítají s proměnlivou fitness funkcí na stejném stavovém prostoru. Charakteristikou zde popsaného systému je však proměnlivý stavový prostor, který se zvětšuje při spuštění nového procesu a naopak snižuje se s časem – každá další optimalizace probíhá na systému, který už je o nějaký čas v budoucnosti a nemá tedy smysl při optimalizaci uvažovat aktivity, které už proběhly.

Literatura

[1] *PSPLIB – Library for Project Scheduling Problems* [online]. 1996 [Cit. 12.5.2012].
URL http://129.187.106.231/psplib/

[2] Alander, J.: On optimal population size of genetic algorithms. IEEE Comput. Soc. Press, ISBN 0-8186-2760-3, s. 65–70, doi:10.1109/CMPEUR.1992.218485.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=218485&tag=1

[3] Brucker, P.; Knust, S.; Brucker, P.; aj.: Resource-Constrained Project Scheduling. In *Complex Scheduling*, GOR-Publications, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-23929-8, s. 117–238.
URL http://dx.doi.org/10.1007/978-3-642-23929-8_3

[4] Congram, R. K.; Potts, C. N.; Van de Velde, S. L.: An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem. *INFORMS Journal on Computing*, ročník 14, č. 1, 2002: str. 52, ISSN 10919856.

[5] Dasgupta, D.; Michalewicz, Z.: *Evolutionary algorithms in engineering applications*. Springer, 1997, ISBN 9783540620211.

[6] Demeulemeester, E. L.; Herroelen, W. S.: New benchmark results for the resource-constrained project scheduling problem. *Manage. Sci.*, ročník 43, č. 11, Listopad 1997: s. 1485–1492, ISSN 0025-1909.
URL http://dx.doi.org/10.1287/mnsc.43.11.1485

[7] Foo Yoon-Pin Simon; Takefuji, T.: Integer linear programming neural networks for job-shop scheduling. IEEE, 1988, ISBN 0-7803-0999-5, s. 341–348 vol.2.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=23946&tag=1

[8] Garey, M. R.; Johnson, D. S.; Sethi, R.: The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, ročník 1, č. 2, Květen 1976: s. 117–129.

[9] Janoušek, V.: PNtalk: Object Orientation In Petri Nets. 1995.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.8080

[10] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Dizertační práce, 1998.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=6001

[11] Janoušek, V.; Kironský, E.: Exploratory Modeling with SmallDEVS. In *Proc. of ESM 2006*, EUROSIS, 2006, ISBN 90-77381-30-9, s. 122–126.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8202

[12] Janoušek, V.; Kočí, R.: Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*, EUROSIM-FRANCOSIM-ARGESIM, 2004, ISBN 3-901608-28-1, s. 31–36.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=7558

[13] Janoušek, V.; Květoňová, Š.: On the Multilevel Petri Nets-Based Models in Project Engineering. In *International Workshop on Petri Nets and Software Engineering 2009*, University of Pierre and Marie Curie, 2009, s. 173–188.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9023

[14] Józefowska, J.: *Just-in-time scheduling: models and algorithms for computer and manufacturing systems*. Springer, 2007, ISBN 9780387717173.

[15] Kolisch, R.; Sprecher, A.: PSPLIB – A project scheduling problem library. *European Journal of Operational Research*, ročník 96, 1996: str. 205–216.

[16] Madureira, A.; Ramos, C.; Silva, S.: Using Genetic Algorithms for Dynamic Scheduling. In *Production and Operations Management 2003 Conference (POMS'2003)*, 2003.

[17] Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, ročník 77, č. 4, Duben 1989: s. 541–580, ISSN 0018-9219.

[18] Persson, A.; Grimm, H.; Ng, A.; aj.: Simulation-based multi-objective optimization of a real-world scheduling problem. In *Proceedings of the 38th conference on Winter simulation*, WSC '06, Winter Simulation Conference, 2006, ISBN 1-4244-0501-7, str. 1757–1764.
URL http://dl.acm.org/citation.cfm?id=1218112.1218431

[19] Ripon, K.; Tsang, C.-H.; Kwong, S.: An Evolutionary Approach for Solving the Multi-Objective Job-Shop Scheduling Problem. In *Evolutionary Scheduling*, *Studies in Computational Intelligence*, ročník 49, Springer Berlin / Heidelberg, 2007, ISBN 978-3-540-48582-7, s. 165–195.

[20] Sharapov, R.; Lapshin, A.: Convergence of genetic algorithms. *Pattern Recognition and Image Analysis*, ročník 16, 2006: s. 392–397, ISSN 1054-6618, 10.1134/S1054661806030084.
URL http://dx.doi.org/10.1134/S1054661806030084

[21] Sule, D. R.: *Production Planning and Industrial Scheduling: Examples, Case Studies and Applications, Second Edition*. CRC Press, druhé vydání, Říjen 2007, ISBN 1420044206.

[22] Wall, M.: *GAlib: A C++ Library of Genetic Algorithm Components* [online]. 1996 [Cit. 14.5.2012].
URL http://lancet.mit.edu/ga/

Příloha A

Obsah přiloženého média

Přiložené médium obsahuje následující adresářovou strukturu:

- `/`
  - `projekt`
    - `ga` — Zdrojový kód externího optimalizátoru
    - `squeak` — Squeak image
    - `log` — Logy z experimentů
      - `exp1.log` — Optimalizace SMRCPSP(5,4) problému
      - `exp2.log` — Jednoduchý projekt, více událostí
      - `exp3.log` — Více projektů v jednom portfoliu
    - `smrcpsp` — Instance SMRCPSP vygenerované programem ProGen
  - `doc` — Technická zpráva
    - `src` — Zdrojový kód této technické zprávy

Příloha B

Konfigurace a spuštění

Postup při spuštění projektu:

1. Nakopírovat adresář projekt z přiloženého média do nějaké zapisovatelné lokace.
2. Nainstalovat knihovnu GALib verze 2.4.7 nebo vyšší, je potřeba k přeložení optimalizátoru.
3. Nainstalovat si aktuální virtuální stroj pro Squeak.
4. Přeložit optimalizátor pomocí utility `make` ve složce projekt/ga.
5. Spustit Squeak image ve složce projekt/squeak.

Pro přeložení textu technické zprávy je nutné mít nainstalovaný latex s balíčkem `dirtree`. Podrobnější pokyny k použití projektu jsou zobrazeny ve spuštěném image.

Příloha C

Detailní popis výrobních postupů

Zde je uvedený detailní popis výrobních postupů, které byly navrženy na základě reálného systému. Protože cílem projektu není přímo optimalizace konkrétního problému, ale studium možností navrženého inteligentního systému a následkem zjištěné nedostatečné výkonnosti simulátoru je v PNtalku implementován pouze menší ze dvou výrobků – osvětlení s ventilátory. To je také důvod proč tabulka C.1 neobsahuje přesné počty jednotlivých zdrojů v projektovém portfoliu - tyto zdroje měly vzniknout na základě pokročilého experimentování s optimalizací systému, která bohužel nebyla realizovatelná. Veškeré v práci popsané zdroje ale implementovány jsou a samotné vytvoření metod projektů je zdlouhavá, ale přímočará záležitost.

ID zdroje	Název zdroje	Typ zdroje	Parametry
PRAC	Pracovník	HumanResource	Prac. doba: 7-15
MATERIAL	Příprava materiálu	ScheduledResource	
SOUSTRUH	Soustruh	ScheduledResource	
LASER	Laserové řezání	OrderedResource	Dovoz: druhý den v 16:00
CHROM	Chromování	OrderedResource	Dovoz: druhý den v 15:00
OBRABENI	Vrtání, broušení atd.	ScheduledResource	
OXID	Alkalická oxidace	BatchResource	Kapacita: 10kg, Trvání: 4 hodiny
ZINEK	Zinkování	BatchResource	Kapacita: 4kg, Trvání: 30 minut
ZAMPRAC	Zámečnické práce	ScheduledResource	
MONTAZ	Montáž	ScheduledResource	
KONTROLA	Kontrolní stanoviště	ScheduledResource	

Tabulka C.1: Konkrétní typy zdrojů modelované v systému. Přesné počty zdroju každého typu by vyplynuly až z optimalizačních experimentů s modelem, které ovšem nebylo možné z důvodů nízké efektivity a spolehlivosti výpočtu fitness funkce provést.

Fáze	Vstupní výrobky	Zdroje	Čas (minuty)	Váha (kg)
	Výrobek: VENT - Ventilátor			
1.		LASER		
2.		ZAMPRAC, PRAC	4	
3.		OXID		0,3
4.		KONTROLA, 0,5xPRAC	3	
	Výrobek: SLO1 - Sloupek			
1.		MATERIAL, 0,5xPRAC	5	
2.		SOUSTRUH, PRAC	4	
3.		OXID		0,1
4.		KONTROLA, 0,5xPRAC	3	
	Výrobek: OSVV - Osvětlení s ventilátory			
1.	9xVENT, 4xSLO1	MATERIAL, 0,5xPRAC	2	
2.		MONTAZ, 2xPRAC	60	
3.		KONTROLA, 0,5xPRAC	10	

Tabulka C.2: Pracovní postup pro sestavu *Osvětlení s ventilátory*

Fáze	Vstupní výrobky	Zdroje	Čas (minuty)	Váha
	Výrobek: BUB1 - Bubínek sací, první část			
1.		MATERIAL, 0,5xPRAC	5	
2.		SOUSTRUH, PRAC	20	
3.		ZAMPRAC, PRAC	10	
4.		KONTROLA, 0,5xPRAC	5	
5.		CHROM		0,1
	Výrobek: BUB2 - Bubínek sací, druhá část			
1.	BUB1	MATERIAL, 0,5xPRAC	2	
2.		OBRABENI, PRAC	25	
3.		OBRABENI, 0,5xPRAC	23	
4.		OBRABENI, PRAC	22	
5.		KONTROLA, 0,5xPRAC	10	
	Výrobek: HRI1 - Hřídel			
1.		MATERIAL, 0,5xPRAC	5	
2.		SOUSTRUH, PRAC	14	
3.		OBRABENI, PRAC	4	
4.		ZINEK		3
5.		KONTROLA, 0,5xPRAC	5	
	Výrobek: ROVN - Rovnač stohu			
1.	HRI1, JEZD	MATERIAL, 0,5xPRAC	12	
2.		MONTAZ, PRAC	18	
3.		KONTROLA, 0,5xPRAC	5	
	Výrobek: UNAS - Unášecí sestava			
1.		MATERIAL, 0,5xPRAC	5	
2.		MONTAZ, PRAC	14	
3.		KONTROLA, 0,5xPRAC	3	
	Výrobek: DES1 - Deska			
1.		LASER		
2.		OBRABENI, 0,5xPRAC	20	
3.		ZAMPRAC, PRAC	5	
4.		OBRABENI, PRAC	37	
5.		ZAMPRAC, PRAC	6	
6.		OBRABENI, PRAC	70	
7.		OXID		1,1
8.		KONTROLA, 0,5xPRAC	10	
	Výrobek: JEZD - Jezdec			
1.		LASER		
2.		KONTROLA, 0,5xPRAC	2	
3.		ZAMPRAC, PRAC	1	
4.		OXID		0,15
5.		KONTROLA, 0,5xPRAC	5	

Tabulka C.3: Pracovní postup pro sestavu *Brzda archu* - první část. Tento model není v práci implementován z důvodů objasněných v kapitole 5.

Fáze	Vstupní výrobky	Zdroje	Čas (minuty)	Váha
	Výrobek: VTYC - Vodící tyč			
1.		MATERIAL, 0,5xPRAC	15	
2.		SOUSTRUZIT, PRAC	52	
3.		OXID		0,5
4.		KONTROLA, 0,5xPRAC	5	
	Výrobek: VZAV - Vyvažovací závaží			
1.		MATERIAL, 0,5xPRAC	17	
2.		SOUSTRUZIT, 0,5xPRAC	58	
3.		OBRABENI, PRAC	45	
4.		OXID		2
5.		KONTROLA, 0,5xPRAC	10	
	Výrobek: DRZ1 - Držák 1			
1.		MATERIAL, 0,5xPRAC	14	
2.		OBRABENI, PRAC	32	
3.		ZAMPRAC, PRAC	5	
4.		OBRABENI, PRAC	80	
5.		OBRABENI, 0,5xPRAC	21	
6.		ZAMPRAC, PRAC	5	
7.		OBRABENI, PRAC	31,5	
8.		ZAMPRAC, PRAC	5	
9.		OBRABENI, PRAC	27	
10.		OXID		0,1
11.		KONTROLA, 0,5xPRAC	20	
	Výrobek: HRI2 - Hřídel 2			
1.		MATERIAL, 0,5xPRAC	16,5	
2.		SOUSTRUH, PRAC	63	
3.		OBRABENI, 0,5xPRAC	65	
4.		OBRABENI, PRAC	17,5	
5.		SOUSTRUH, PRAC	46	
6.		OBRABENI, PRAC	17,5	
7.		SOUSTRUH, PRAC	27	
8.		OBRABENI, PRAC	89	
9.		KONTROLA, 0,5xPRAC	10	
10.		OBRABENI, PRAC	39,5	
11.		ZAMPRAC, PRAC	10	
12.		OXID		3
13.		KONTROLA, 0,5xPRAC	15	

Tabulka C.4: Pracovní postup pro sestavu *Brzda archu* - druhá část. Tento model není v práci implementován z důvodů objasněných v kapitole 5.

Fáze	Vstupní výrobky	Zdroje	Čas (minuty)	Váha
	Výrobek: DRZ2 - Držák 2			
1.		MATERIAL, 0,5xPRAC	15	
2.		OBRABENI, PRAC	32	
3.		ZAMPRAC, PRAC	5	
4.		OBRABENI, PRAC	90	
5.		OBRABENI, 0,5xPRAC	21	
6.		ZAMPRAC, PRAC	5	
7.		OBRABENI, PRAC	31,5	
8.		ZAMPRAC, PRAC	5	
9.		OXID		0,1
10.		KONTROLA, 0,5xPRAC	20	
	Výrobek: NATR - Nátrubek			
1.		MATERIAL, 0,5xPRAC	2	
2.		OBRABENI, PRAC	26	
3.		ZAMPRAC, PRAC	8,5	
4.		ZINEK		0,1
5.		KONTROLA, 0,5xPRAC	5	
	Výrobek: BRZD - Brzda archu			
1.	DRZ1, DRZ2,	MATERIAL, PRAC	15	
	4xNATR, 4xBUB2,			
	ROVN, 2xUNAS			
	HRI2			
2.		MONTAZ, PRAC	100	
3.	DES1	MATERIAL, PRAC	2	
4.		MONTAZ, PRAC	5	
5.	2xVZAV, VTYC	MATERIAL, PRAC	5	
6.		MONTAZ, PRAC	80	
7.		MONTAZ, PRAC	10	
8.		KONTROLA, 0,5xPRAC	30	

Tabulka C.5: Pracovní postup pro sestavu *Brzda archu* - třetí část