Diplomová práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Software pro výpočet sekvenčně závislých přestavbových časů ve strojní výrobě

Matěj Mihal

Vedoucí práce: Ing. Jan Smejkal
Studijní program: Otevřená informatika
Obor: Softwarové inženýrství
Květen 2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mihal** Jméno: **Matěj** Osobní číslo: **397898**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Software pro výpočet sekvenčně závislých přestavbových časů ve strojní výrobě

Název diplomové práce anglicky:

Software for calculation of sequence-dependent setup times in the machinery industry

Pokyny pro vypracování:

Optimalizace sekvenčně závislých přestavbových časů ve výrobě snižuje prostoje strojů [2]. Přestavbové časy definuje doménový odborník na základě analýzy procesů výroby na daném stroji. Matice, do které časy zaznamenává, rychle narůstá do velikosti, kdy ji už nelze udržovat ručně.
Problém lze vyřešit definováním přestavbových časů pomocí pravidel založených na parametrech výrobního procesu. Vytvořte systém, který umožní doménovému odborníkovi pravidla definovat. Implementujte také výpočet, který sestaví matici přestavbových časů na základě definovaných pravidel.
1. Seznamte se s problematikou optimalizace přestavbových časů a tvorby přestavbové matice.
2. Analyzujte možnosti definování pravidel doménovým odborníkem. V analýze zhodnoťte řešení, které by bylo založeno na doménově specifických jazycích [1].
3. Navrhněte a realizujte systém umožňující definici jednotlivých pravidel přestaveb.
4. Implementuje konstrukci přestavbové matice ze zadaných pravidel.
5. Ověřte přínos vzniklého řešení formou uživatelského testování s doménovými odborníky. Testujte minimálně se dvěma doménovými odborníky. Uvažujte alespoň následující typy strojů: laser, ohraňovací stroj a frézka.

Seznam doporučené literatury:

1] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. https://www.martinfowler.com/articles/languageWorkbench.html.
[2] V. DAL. Minimizing machine changeover time in product line in an apparel

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing Jan Smejkal, Merica s.r.o.

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.02.2019** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **20.09.2020**

Ing Jan Smejkal podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Poděkování

Rád bych na tomto místě poděkoval Ing. Janu Smejkalovi za vedení této práce, za cenné a podnětné návrhy k řešeným problémům a za poskytnutí vhledu do domény.

Dále bych rád poděkoval Ing. Veronice Vetýškové za jazykovou korekturu textu této práce a za podporu během celého jejího vzniku, jakožto i během studia, které vzniku této práce předcházelo.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 22. května 2019

Matěj Mihal

Abstrakt

Sekvenčně závislé přestavbové časy hrají důležitou roli při plánování výroby. Správnou volbou pořadí artiklů je možné přestavbové časy minimalizovat a tím zvýšit objem výroby a ušetřit zbytečnou práci. Získání těchto časů je však nesnadný úkol, kterému je prozatím věnováno jen velmi málo pozornosti.

Tato práce navrhuje řešení problému získávání přestavbových časů pomocí systému pravidel pro dílčí úkony přestaveb. Pravidla jsou založena na vlastnostech výrobního procesu. Systém cílí na podniky strojní výroby, ale může se uplatnit i v jiných oblastech. Na základě analýzy je v nástroji Jetbrains MPS vytvořen nový doménově specifický jazyk Changeovers určený pro definici pravidel doménovými odborníky. Odborníci zapojení do testování jazyka dokázaly vytvořit smysluplné a bezchybné programy a byli s užíváním systému spokojeni. Další součástí práce je i generátor, který vytvořená pravidla uplatní na data výrobního procesu a sestaví matici přestavbových časů potřebnou pro plánování výroby.

Klíčová slova: plánování ve výrobě, přestavbové časy, DSL, doménově specifický jazyk, Jetbrains MPS

Vedoucí práce: Ing. Jan Smejkal
Merica s.r.o.
Velflíkova 1417/12
Praha 6

Abstract

Sequence-dependent setup times play an important role during production scheduling. This setup times could be reduced by the wise choice of a production order. That leads to increased production and reduce the amount of unnecessary work. However, getting the setup times is a tough so far overlooked task.

This thesis provides a solution for gaining setup times. We came with the system of rules for changeover operations. The rules are based on production process attributes. The system aims to mechanical engineering production companies but could be also used in other fields. After analysis, we created a new domain-specific language called Changeovers in JetBrains MPS tool. Changeovers are meant for domain experts to define the rules. The experts were also involved in usability testing. Testing revealed that experts are capable of creating meaningful and error-free programs in Changeovers. Experts were also satisfied with the usability of the system. Setup times matrix generator was also created as part of this thesis. The generator applies the rules on production process data and composes setup times matrix. The matrix is utilized by the scheduling system.

Keywords: production scheduling, setup times, DSL, domain specific language, Jetbrains MPS

Title translation: Software for calculation of sequence-dependent setup times in the machinery industry

Obsah

1 Úvod 1
1.1 Cíl práce . . . 2
1.2 Čím se tato práce nezabývá . . . 2

2 Současný stav problematiky 3
2.1 Strojní výroba . . . 3
2.2 Přestavba stroje . . . 3
2.3 Přestavbový čas . . . 4
2.4 Optimalizace výrobního plánu . . . 4
2.5 Matice sekvenčně závislých přestavbových časů . . . 5
2.6 Získávání přestavbových časů . . . 5
2.6.1 Ruční sestavení matice . . . 6
2.6.2 Sledování vlastnosti artiklu . . . 6
2.6.3 Získání počtu úkonů . . . 6
2.6.4 Získání matice z ERP systému 6
2.6.5 Nevhodnost existujících přístupů . . . 7

3 Analýza problému 9
3.1 Analýza problémové domény . . . 9
3.1.1 Uživatelé vzniklého systému . . 9
3.1.2 Přestavby stroje . . . 9
3.1.3 Vlastnosti artiklů . . . 10
3.1.4 Stroj . . . 10
3.2 Doménově specifické jazyky . . . 11
3.2.1 Výhody využití DSL . . . 11
3.2.2 Nevýhody využití DSL . . . 12
3.2.3 Typy DSL . . . 13
3.2.4 Nástroje pro tvorbu externích DSL . . . 15
3.2.5 Grafické . . . 15
3.2.6 Textové . . . 17
3.2.7 Projekční editory . . . 19
3.3 Zvolený nástroj . . . 21
3.4 Návrh řešení . . . 22

4 Vývoj jazyka v Jebrains MPS 25
4.1 Aspekty . . . 25
4.1.1 Structure . . . 26
4.1.2 Editor . . . 26
4.1.3 Constrains . . . 27
4.1.4 Behaviour . . . 27
4.1.5 Typesystem . . . 27
4.1.6 Generator . . . 27
4.1.7 Další aspekty . . . 28
4.2 Rozšiřování jazyků . . . 28
4.2.1 BaseLanguage . . . 28

5 Jazyk Changeovers 31
5.1 Základní vlastnosti jazyka . . . 31
5.1.1 Výchozí jazyk . . . 31
5.1.2 Lokalizace Changeovers . . . 31
5.1.3 Changeovers z pohledu uživatele . . . 31
5.1.4 Struktura programů v Changeovers . . . 33
5.1.5 Skupiny konstruktů v Changeovers . . . 33
5.2 Kontejnery . . . 34
5.2.1 Changeovers . . . 34
5.2.2 Machine . . . 35
5.3 Definice časů . . . 37
5.3.1 TimeUnit . . . 37
5.3.2 Time . . . 38
5.4 Úkony přestaveb . . . 39
5.4.1 ChangeoverOperation . . . 39
5.5 Vlastnosti procesu výroby . . . 41
5.5.1 OperationProperty . . . 41
5.5.2 NextOperationProperty . . . 42
5.5.3 PrevOperationProperty . . . 43
5.5.4 BetweenOpsProp . . . 43
5.5.5 BetweenOpsPropRef . . . 44
5.6 Podmínky přestaveb . . . 45
5.6.1 ChangeoverConditionSection 45
5.6.2 ChangeoverCondition . . . 45
5.7 Zkratky . . . 47
5.7.1 OperationPropertyRef . . . 48
5.7.2 Goes . . . 48
5.7.3 Changes . . . 49

6 Generování přestavbové matice 51
6.1 Rozhraní transformovaného jazyka Changeovers . . . 51
6.1.1 Třída Changeovers . . . 51
6.1.2 Implementace rozhraní MachineRules . . . 51
6.2 Konstrukce přestavbové matice . 52
6.2.1 Data operací . . . 52
6.2.2 Vlastnosti programu . . . 52
6.2.3 Výstup programu . . . 53
6.2.4 Implemetace programu . . . 53

7 Uživatelské testovaní 55
7.1 Průběh testování . . . 55
7.1.1 Standalone IDE . . . 55
7.1.2 Seznámení s Editorem . . . 55

7.1.3 Princip vytváření programů . 56
7.1.4 Způsob vytváření programů . 56
7.1.5 Úkol pro uživatele 57
7.1.6 Sledování průběhu 57
7.1.7 Hodnocení jazyka uživateli . . 57
7.2 Odborník A 57
7.2.1 Průběh 57
7.2.2 Hodnocení jazyka 57
7.2.3 Zjištěné nedostatky 58
7.2.4 Zhodnocení 58
7.3 Odborník B 58
7.3.1 Průběh 58
7.3.2 Hodnocení jazyka 59
7.3.3 Zjištěné nedostatky 59
7.3.4 Zhodnocení 59
7.4 Vyhodnocení testování 59

8 Závěr 61
8.1 Dosažené výsledky 61
8.2 Přínos vzniklého systému 62
8.3 Možné navazující práce 62
8.4 Hodnocení použitého nástroje . . 62

Literatura 63

A Seznam použitých zkratek 67

B Pomocné třídy a rozhraní implementované přímo v Base Language 69
B.1 Třída ChangeoverOperation . . . 69
B.2 Rozhraní MachineRules 70
B.3 Rozhraní ITask 70

C Programy vniklé při testování 71
C.1 Odborník A 71
C.2 Odborník B 73

D Obsah přiloženého CD 75

Obrázky

3.1 Ukázka programování v nástroji Metaedit+ Modeler převzato z [10]. 16
3.2 Ukázka programování v prostředí jazyka Scratch. Vytvořeno na [11]. 17
3.3 Ukázka programování v prostředí webového prohlížeče v jazyce vytvořeném v nástroji Xtext. Převzato z [7] 19
3.4 Ukázka programování v prostředí MPS. 21
3.5 Pohled na fungování navrhovaného řešení. 22

5.1 Jednoduchý rozpracovaný program v jazyce *Changeovers*. 32
5.2 Zjednodušený class diagram jazyka *Changeovers*. 33
5.3 Root template. Generátor konstruktu Changeovers. 35
5.4 Generator template konstruktu *Machine*. 38
5.5 Definice nové časové jednotky. . . 39
5.6 Násobný přestavbový úkon se zobrazeným *context asistentem*. . . . 40
5.7 Aspekt Structure konstruktu *NextOperationProperty*. 43
5.8 Aspekt Constrains konstruktu *BetweenOpsPropRef*. 44
5.9 Podmínka přestavby spouštějící násobný úkon. 46
5.10 Aspekt Typesystem konstruktu *ChangeoverCondition* - infer pravidla. 47
5.11 Aspekt Typesystem konstruktu *ChangeoverCondition* - checking pravidla. 47
5.12 Podmínky spuštění redukce konstruktu *Goes* v mapování pravidel generatoru. 49

B.1 Implementace třídy `Changeovers`. 69
B.2 Implementace rozhraní `MachineRules`. 70
B.3 Implementace rozhraní `ITask`. . 70

C.1 Popis pravidel přestaveb frézky od odborníka A. 71
C.2 Popis pravidel přestaveb laseru od odborníka A. 72
C.3 Popis pravidel přestaveb ohraňovacího stroje od odborníka A. 72
C.4 Popis pravidel přestaveb frézky od odborníka B. 73
C.5 Popis pravidel přestaveb laseru od odborníka B. 73
C.6 Popis pravidel přestaveb ohraňovacího stroje od odborníka B. 74

Zdrojové kódy

3.1 Příklad užití návrhového vzoru *fluent interface*. Převzato z[29] 14

3.2 Příklad programu v *textovém* DSL. Převzato z[18] . . . 18

6.1 Metoda `main` třídy `MatrixGenerator`. 53

6.2 Metoda `loadTasks` třídy `MatrixGenerator`. 53

6.3 Metoda `generateMatrix` třídy `MatrixGenerator`. . 54

Kapitola 1

Úvod

Plánování je pro výrobní podnik velmi důležitá činnost. Dobře vytvořený plán dokáže zvýšit efektivitu podniku mnohdy až o desítky procent. S pomocí dobrého plánování dokáže podnik zvýšit objem výroby bez navýšení strojních nebo lidských zdrojů. To je nedocenitelné zejména v dnešní době, kdy je na trhu lidské síly nedostatek a cena lidské práce neustále stoupá. V tomto kontextu lze také říci, že dobré plánování pomáhá snižovat náklady.

Vytvoření dobrého plánu je velmi komplikovaný proces. Vytvořit optimální plán je v současné době jak nad lidské síly, tak i nad možnosti počítačových systému[1]. Výsledek plánování ovlivňuje velké množství faktorů. Nejen například výkonost strojů, ale i dovolená zaměstnanců. Existují však různé počítačové systémy, které dokáží navrhnout plán blížící se optimálnímu.

Takové systémy potřebují pro sestavení plánů co možná nejpřesnější informace o celém výrobním procesu. Požadují údaje o vyráběných artiklech a jejich množství, termínu, do kterého mají být daná množství vyrobena, výrobní postupy artiklů, ale také výkonnost výrobních stanovišť, skladové zásoby materiálů, odhadované doby na výrobu, přestavbové časy strojů a mnoho dalších. Přesnost informací je jeden z faktorů ovlivňujících, zda bude podnik schopen sestavený výrobní plán dodržovat.

Získat všechny potřebné údaje je však další nelehký úkol. Některé z nich totiž vůbec nemusí existovat v datech, jež má výrobní podnik k dispozici, proto je potřeba zavádět nové postupy, jimiž lze chybějící údaje získat. Jedním z takových údajů jsou i přestavbové časy strojů ve své sekvenčně závislé variantě.

Po dokončení výroby jednoho artiklu je potřeba stroj přestavit. Až poté je možné vyrábět další artikl. Dobu potřebnou k realizaci této přestavby označujeme jako přestavbový čas. Pokud délku tohoto času ovlivňuje jak předchozí, tak následující vyráběný artikl, jedná se o sekvenčně závislý přestavbový čas.

Tyto časy lze získat s pomocí pravidel využívajících vlastnosti výrobního procesu. Pokud má plánovací systém pravidla k dispozici, promítne je do výsledného plánu, který se tak více přiblíží skutečnosti. Navíc vhodnou volbou pořadí artiklů v plánu tyto časy snižuje. To redukuje prostoje strojů a šetří zbytečnou práci. Není například nutné na stroji měnit nástroj třikrát, ale postačí dvě výměny.

[1] Velké množství těchto problémů je NP obtížných.

1.1 Cíl práce

Pro vytvoření plánu je potřeba znát přestavbové sekvenčně závislé časy mezi každou dvojicí atriklů, jež se na stroji vyrábí. K tomu slouží matice sekvenčně závislých přestavbových časů.

Cílem práce je na základě analýzy problému navrhnout, vyvinout a otestovat systém, jež zjednoduší a případně pro větší množství operací vůbec umožní[2] vytvářet tyto matice.

Vzniklý systém umožní doménovému odborníkovi, zejména z oblasti strojní výroby, nadefinovat pravidla, která určují, zda je potřeba stroj mezi dvěma po sobě vyráběnými artikly přestavět a jak dlouho taková přestavba bude trvat.

Pravidla musí být natolik jednoduchá, přehledná, či podobná technickému zápisu, aby je doménový odborník po předchozím zaškolení dokázal samostatně nejen číst a porozumět jim, ale i je vytvářet.

Na základě pravidel a údajů o jednotlivých operacích, které je potřeba na stroji naplánovat, systém sestaví matici sekvenčně závislých přestavbových časů. Tato matice může být dále využita systémem pro plánování výroby.

1.2 Čím se tato práce nezabývá

Tato práce se nezabývá mapováním dat z enterprise resource planning (ERP) systému do systémů pro plánování či získávání dat o operacích z ERP systémů. V této práci předpokládáme, že veškerá data nutná pro vyhodnocení pravidel a sestavení matice sekvenčně závislých časů má náš systém k dipozici a že jsou data správně namapována.

[2] Matice přestavbových časů roste kvadraticky s počtem plánovaných operací tzn. pro pouhých 100 plánovaných operací je potřeba vyplnit 10 000 polí matice časem potřebným na přestavení stroje. Běžně se plánují i tisíce nebo desítky tisíc operací, kde už není v lidských silách takto velkou matici vyplnit.

Kapitola 2

Současný stav problematiky

Tato kapitola nejprve zavádí pojmy, se které jsou v této práci používany. Konkrétně pojmy *strojní výroba* (2.1), *přestavba stroje* (2.2), *přestavbový čas* (2.3) a *optimalizace výrobního plánu* (2.4).

Z tohoto základu pak vychází při vysvětlování co je *matice přestavbových časů* (2.5), jaká je její struktura a jaké v ní platí podmínky.

Na závěr se v sekci 2.6 věnuje několika postupům, jak získat sekvenčně závislé přestavbové časy, případně celou matici přestavbových časů. Překvapivě se tímto tématem literatura téměř nezabývá.

2.1 Strojní výroba

Strojní výroba je výroba ve které se je vstupní surovinou hutní materiál z nějž se sérií operací vyrábí z větší části kovové artikly. Mezi operace patří například svařování, řezání, ohýbání, frézování, soustružení, lisovaní, broušení, lakování a mnoho dalších. Jako příklady artiklů uveďme třeba ramena jeřábů, odpadkové koše, rozvaděčové skříně, rámy vozíků, olejové vany a nepřeberné množství dalších.

Jak již bylo řečeno výše, postup výroby artiklu se skládá ze série operací. Tyto operace se velmi často provádějí postupně na různých pracovištích s využitím různých strojů.

Například při výrobě jednoduchého kovového krytu se nejdříve na laserové vypalovačce z tabule plechu vypálí[1] díly krytu. Tyto díly ve svařovně spojí svářeč svary. V lakovně se svařenec nalakuje na požadovanou barvu a po zaschnutí barvy je finální artikl hotový.

Strojní výroba je oblast z níž pocházejí doménový odborníci se kterými bylo spolupracováno v této práci a na než cílí výsledky této práce.

2.2 Přestavba stroje

Pokud chceme změnit výrobu z jednoho artiklu na jiný artikl, musíme na stroji provést sadu úkonů, aby byl schopen jiný artikl vyrábět. Tuto změnu budeme označovat jako *přestavbu stroje*. Jedná se tedy o množinu (velmi často

[1] vyříznou

sekvenci) úkonů, které je nutné na stroji provést, abychom přešli z výroby jednoho artiklu k druhému. Jako například výměna nástroje, přišroubování upínacího přípravu a nastavení nulových bodů nebo výměna sklíčidel. Čas potřebný pro provedení této sady nazýváme jako *přestavbový čas*.

Na jednom stroji lze provádět množství přestavbových úkonů. Nemusí být nutné provádět všechny tyto úkony při každém přestavování stroje. Podle toho, na čem množina prováděných úkonů jedné konkrétní přestavby závisí, rozdělujeme typy přestaveb.

2.3 Přestavbový čas

Rozlišujeme dva základní typy přestaveb, a to *sekvenčně nezávislé* a *sekvenčně závislé*. Pokud jsou úkony přestavby nezávislé na předchozím vyráběném artiklu a závisí pouze na nově vyráběném artiklu jedná se o *sekvenčně nezávislé přestavby*. Časy těchto přestaveb nelze optimalizovat vhodným pořadím po sobě vyráběných artiklů. Oproti tomu sada úkonů *sekvenčně závislý přestaveb* závisí, jak na předchozím, tak na nově vyráběném artiklu. Je tedy možné vhodnou volbou pořadí vyráběných artiklů optimalizovat tyto přestavbové časy. Vhodné pořadí artiklů umožní například provádět v rámci přestaveb méně úkonů, nebo provádět méně časově náročné úkony.

Sekvenčně závislý přestavbový čas označujeme jako $s_{i,j}$, kde i značí předchozí a j následující vyráběný artikl. Předpokládáme, že pokud $i = j$ pak platí $s_{i,j} = 0$. Pokud má na přestavbový čas vliv i stroj přidáme do notace písmeno k značící daný stroj $s_{i,j,k}$. I v tomto případě platí, že pokud $i = j$ pak $s_{i,j,k} = 0$ pro všechna $k \in \{\text{množina strojů}\}$[13, 28].

2.4 Optimalizace výrobního plánu

Optimální výrobní plán[2] je takový plán, který má nejnižší hodnotu kritéria, kterým plán hodnotíme. Tyto kritéria mohou být různá, například nejkratší výrobní plán, nejmenší zpoždění zakázek, nejmenší celková doba přestaveb strojů a mnoho dalších[4]. Velmi často se používají kritéria složená z více dílčích kritérií. Takové kritérium získáme jako sumu všech dílčích kritérií. Je obvyklé každé dílčí kritérium před sčítáním násobit jeho váhou, čímž se určí jeho důležitost. Pro vyhodnocení některých dílčích kritérií plánu výroby je potřeba znalost sekvenčně závislých přestavbových časů. Jedná se například o výše zmíněnou celkovou délku přestavbových časů a potažmo i celkovou délku plánu.

Pro nalezení optimálního plánu výroby je potřeba sestavit všechny možné plány, vyhodnotit jejich kritéria a vybrat plán, jehož kritérium je nejmenší. To však často není možné provést v rozumně dlouhém čase. Jedná se totiž o problém známý jako rozvrhování a značná část rozvrhovacích problémů patří do třídy složitost *NP*[19]. V praxi se tak často využívá různých heuristických

[2] Pořadí vyráběných artiklů na jednotlivých strojích ve výrobě.

algoritmů. Jejich výsledkem však nemusí být optimální plán, měl by se však optimálnímu plánu co možná nejvíce blížit[14].

2.5 Matice sekvenčně závislých přestavbových časů

Matice sekvenčně závislých přestavbových časů je sestavená pro každý stroj samostatně a zaznamenává *přestavbový čas* pro každou možnou dvojici typů artiklů vyráběných na daném stroji v pořadí za sebou.

Očíslujeme-li všechny typy artiklů, které je potřeba na daném stroji rozvrhnout[3] čísly od 1 do n bude mít matice přestaveb rozměr $n \times n$. Elementy matice tvoří přestavbové časy $s_{i,j,k}$. Protože matici sestavujeme pro každý stroj samostatně slouží index k přestavbového času k výběru matice a posléze již indexujeme v matici pouze indexy i a j. Matici označujeme jako K_k a její znázornění je uvedeno po tímto odstavcem.

$$\mathrm{K}_k = \begin{pmatrix} 0 & s_{1,2,k} & s_{1,3,k} & \dots & s_{1,n,k} \\ s_{2,1,k} & 0 & s_{2,3,k} & \dots & s_{2,n,k} \\ \dots & \dots & \dots & \dots & \dots \\ s_{n,1,k} & s_{n,2,k} & s_{n,3,k} & \dots & 0 \end{pmatrix}$$

Na diagonále matice K_k jsou nuly, což plyne z vlastnosti přestavbových časů pokud $i = j$, pak $s_{i,j,k} = 0$ pro všechna k z množiny strojů[13]. Zjednodušeně to znamená, že pokud se na stroji po sobě vyrábějí dva artikly stejného typu, tak není potřeba stroj přestavovat.

Druhou podmínku, kterou musí matice splňovat je takzvaná trojúhelníková nerovnost. Tedy musí platit nerovnost $s_{i,h,k}+s_{h,j,k} \geq s_{i,j,k}$ pro všechna $i, j, h \in \{1..n\}$, $k \in \{\text{množina strojů}\}$[13]. Což se dá interpretovat tak, že vždy musí být stejně rychlé nebo rychlejší přestavět stroj z jedno artiklu na druhý, jako součet časů pokud se přestavuje stroj z jednoho artiklu na třetí a poté z třetího na druhý.

Vzhledem k tomu, že počet elementů matice roste kvadraticky s počtem artiklů velmi rychle přestává být v lidských silách matici vytvářet, natož udržovat.

2.6 Získávání přestavbových časů

Literatura se o získávání sekvenčně závislých představových moc často nezmiňuje, ale v praxi se používá několik přístupů:

- Ruční sestavení malých matic přestavbových časů
- Sledování jedné či více vlastností artiklů, které zapříčiňují nejvíce náročný úkon

[3] Je samozřejmě možné postupovat stejně i se všemi typy artiklů co lze na daném stroji vyrábět, ale dostali bychom zbytečně větší matici, jejíž část se při rozvrhování nevyužije.

- Získání počtu úkonů a uvažování konstantního času na jeden úkon[14]
- Získání matice z ERP systému[30]

2.6.1 Ruční sestavení matice

Ruční sestavení matice sekvenčně závislých přestavbových časů je zdánlivě nejjednodušší přístup. Spočívá v tom, že doménový odborník vyplní všechny elementy všech matic[4] na základě analýzy nebo své zkušenosti. To je však možné pouze u malého počtu artiklů. Řádově se může jednat maximálně o menší desítky artiklů. Údržba takové matice pak obvykle vyžaduje součinnost dvou lidí, doménového odborníka, který určí nové časy a někoho dalšího, kdo upraví matici v systému, v němž je uložena.

2.6.2 Sledování vlastnosti artiklu

Další možností jak získat přestavbové časy je zjednodušující model. Sledujeme pouze jednu vlastnost po sobě vyráběných artiklů. Jedná se například o barvu, rozměr nebo materiál. Pokud mají tuto vlastnost stejnou není podle modelu potřeba stroj přestavovat. V opačném případě uvažujeme konstantní přestavbový čas. Sledovaná vlastnost by měla být taková, která zapříčiní nutnost provedení úkonu trvajícího nejdelší čas. Úkony trvající kratší dobu jsou zanedbány. Je samozřejmě možné sledovat vlastností více, ale je poté zapotřebí udělat důslednou analýzu a implementovat složitější logiku. U jednoduchého modelu je tedy problém zanedbávání značného množství úkonů. U složitého modelu je problém model vůbec vybudovat a posléze udržovat.

2.6.3 Získání počtu úkonů

V oděvním průmyslu se vyskytuje problém velmi podobný problému získávání sekvenčně závislých přestavbových časů pro jeden stroj. Jedná se však o získávání přestavbových časů pro celou linku na níž se artikl vyrábí. Linka je sestavena z konečného množství stanovišť, na kterých se mohou nacházet různé typy strojů. Pro každý artikl máme dáno jaký stroj musí být na jakém pracovišti[5]. Pokud se při přechodu z jednoho artiklu na jiný podíváme na každé pracoviště a pokud je potřeba stroj měnit, přičteme k celkovému přestavbovému času konstantní čas. Tím pádem mají artikly jež vyžadují na více stejných pracovištích stejné stroje mezi sebou kratší přestavbové časy, než ty s nižší shodou[14].

2.6.4 Získání matice z ERP systému

Poslední známou možností jak získat přímo matici přestavbových časů poskytuje ERP systém SAP, kde je možno přímo definovat celé matice v podstatě

[4] V tomto případě musí vyplnit matici pro všechny artikly, které lze na stroji vyrábět.
[5] Všechna pracoviště nemusí být obsazena.

stejně jako je popsáno v prvním odstavci. Nebo je možné sestavit více menších matic pro jednu vlastnost[6] artiklů. Z těchto menší matic je po sérii logických kroků stavena kombinovaná výsledná matice sekvenčně závislých přestavbových časů[30].

Tento přístup je vázán pouze na ERP systém SAP, který nemusí mít každý výrobní podnik. Jeho nasazení, případně přechod z jiného systému právě na systém SAP je velmi finančně a časově náročný proces, což využitelnost tohoto přístupu dost snižuje.

2.6.5 Nevhodnost existujících přístupů

Žádný z výše uvedených přístupů není použitelný ve všech případech. *Ruční sestavení matice* selhává pro větší matice, které se však běžně vyskytují. *Sledování vlastnosti artiklu* buď zanedbává méně důležité přestavbové úkony nebo je potřeba vytvořit komplexní program, což je obtížné. *Získání počtu úkonů* není vždy jednoduše možné. Často je potřeba složitější logika pro získání počtu úkonů a navíc většinou přestavbové úkony trvají různě dlouhou dobu. *Získání matice z ERP systému* je možné pouze z ERP systému jednoho výrobce. Navíc pravidla pro získání přestaveb je potřeba vytvářet v tomto systému, což může být dost komplikované. Proto chceme vytvořit nový systém, který bude více univerzální a umožní co možná nejjednodušší vytváření matice.

[6] Je možné, aby jedna matice zachycovala i více vlastností, ale s přibývajícím počtem vlastností rychle roste velikost matice.

Kapitola 3

Analýza problému

Tato kapitola se v úvodu věnuje *analýze problémové domény* (3.1). Z níž vyplynou požadavky na nově vznikající systém. Dále se zaměřuje na možnosti tvorby *doménově specifických jazyků* (3.2), jelikož řešení postavené na doménově specifických jazycích bylo požadováno v zadání této práce. Poté učiníme *volbu nástroje* (3.3), ve kterém vytvoříme doménově specifický jazyk. V závěru kapitoly se věnujeme *návrhu systému* (3.4).

3.1 Analýza problémové domény

Analýza vychází zejména z jednání s doménovými odborníky, kdy jsme sbírali jejich požadavky na nově vznikající systém a také znalosti z domény. Dále analýza vychází z naší vlastní zkušenosti z práce zejména v oblasti optimalizace plánování výroby ve strojní doméně.

3.1.1 Uživatelé vzniklého systému

Cíloví uživatelé vznikajícího systému jsou doménoví odborníci z oblasti výroby, zejména z oboru strojírenství. Jejich pozice bývá označována jako technolog.

Jedná se o osoby znalé všech postupů při výrobě artiklů včetně přestavbových operací na strojích zapojených do výroby. Ve většině případů však neumí programovat. Jsou však zvyklí pracovat s počítačem a umí pracovat zejména s ERP systémy a nástroji sady Microsoft Office. Je tedy potřeba, aby byla práce se systémem jednoduchá a v ideálním případně alespoň vzdáleně připomínala práci s některým ze známých nástrojů.

3.1.2 Přestavby stroje

Na jednom stroji lze provádět konečné množství úkonů vedoucích k jeho přestavění z produkce jednoho artiklu na druhý. Každý úkon má předem známý čas, za který je proveden a má také svůj název.

Úkony vetšinou nelze provádět souběžně a to jednak kvůli tomu, že stroj obvykle obsluhuje jeden člověk. Ten se také stará o provádění úkonů a bývá problematické a zdlouhavé shánění dalších lidí. Další věc, co brání souběžnému provádění více úkonů je to, že si pracovníci provádějící různé úkony budou

navzájem překážet nebo se dokonce blokovat, což provádění úkonů značně prodlouží.

Jeden úkon může být například výměna upínacího přípravku, který trvá 10 minut.

Některé úkony je potřeba provést opakovaně, kdy počet opakování je dán proměnlivým koeficientem, který je získán z vlastností typů materiálů. Jeden takový úkon může být například výměna nástrojů v bubnovém zásobníku, kde koeficientem je počet nástrojů, které je potřeba vyměnit. Výměna jednoho nástroje trvá 2 minuty. Čas na provedení takového úkonu získáme ze vzorce *koeficent* × *čas na jedno provedení úkonu*. V našem případě by tedy výpočet času na úkon vypadal takto *počet nástrojů které je potřeba vyměnit* × *2 minuty*.

Vlastnosti předchozího a následujícího vyráběného artiklu určují, které úkony se v rámci přestavby mají provést. Čas potřebný k provedení přestavby získáme sečtením časů potřebných na prováděné úkony.

3.1.3 Vlastnosti artiklů

Na každém stroji je na artiklu prováděna určitá operace výrobního postupu. Tato operace má vlastnosti, které mohou být různých datových typů. Z toho vyplývá, že vlastnosti jednoho artiklu mohou být na různých strojích různé. Vlastnosti artiklu specifické pro jeden stroj budeme tedy nazývat *vlastnosti operace*. Pro stanovení, které úkony se budou na stroji provádět je potřeba znát stejnou množinu vlastností pro předchozí a následující operaci prováděnou na stroji.

V některých případech je vhodné využít vlastnosti získané složitějším algoritmem z vlastností jak předchozí, tak následující operace. Příklad takové vlastnosti je výše zmíněný počet nástrojů, které je potřeba vyměnit. Ten je získán ze soupisu nástrojů potřebných pro předchozí operaci a stejného soupisu pro následující operaci. Takové vlastnosti označujeme jako *vlastnosti mezi operacemi*.

Vlastnost je určena názvem a datovým typem.

3.1.4 Stroj

Pro výpočet sekvenčně závislých přestavbových časů pro jeden stroj či skupinu identických strojů[1] je potřeba, definovat následující vlastnosti:

- název stroje/skupiny strojů
- identifikátory strojů
- vlastnosti operace na něm prováděné
- vlastnosti mezi operacemi

[1] Předpis pro výpočet přestavbových časů je stejný pro celou skupinu, matice sekvenčně závislých přestavbových časů je však sestavena pro každý stroj samostatně.

- úkony přestavby včetně délky trvání
- podmínky vyvolávající úkony přestaveb

3.2 Doménově specifické jazyky

Doménově specifický jazyk (DSL)[2] je programovací jazyk, který je silně spjat s danou problémovou doménou pro kterou byl vytvořen. To je jeho hlavní a zásadní rozdíl oproti obecnému programovacímu jazyku[3][33, 16].

Mezi nejznámější zástupce DSL se řadí jazyky HTML a CSS pro vytváření a stylování webových stránek, jazyk SQL pro dotazování nad relačními databázemi, jazyk Mathematica pro technické a matematické výpočty a v neposlední řadě jazyk LaTeXpro sazbu textu, v němž je mimochodem sázen text této práce[33, 32, 26, 20].

DSL obvykle mívají menší vyjadřovací schopnosti oproti obecným programovacím jazykům, většinou nejsou turingovsky kompletní tzn. nelze v nich implementovat všechno, co lze spočítat pomocí turingova stroje[33, 31]. Množství možných konstruktů bývá výrazně menší než u obecných programovacích jazyků. DSL také často postrádají svůj smysl mimo svou problémovou doménu. Mohou existovat DSL, jež jsou turingovsky kompletní nebo mají více konstruktů než obecné programovací jazyky. Jejich přínos je však značně diskutabilní a přímo se nabízí otázka, proč místo nich nepoužít nějaký existující obecný programovací jazyk, u kterého není potřeba věnovat úsilí jeho vývoji a údržbě.

3.2.1 Výhody využití DSL

Zvýšení produktivity je jednou ze zmiňovaných výhod užití DSL oproti obecným programovacím jazykům. To, co programátor vyjádří v obecném jazyce na několik jednotek až desítek řádků dokáže často v DSL vyjádřit jedním řádkem. Navíc vzhledem k omezené velikosti jazyka většinou najde vhodný koncept rychleji, nemusí přemýšlet, jak pomocí obecných konstruktů popsat problém v doméně, ale použije koncept, který byl pro daný problém přímo navržen[33, 26].

Zlepšení komunikace programátorů s doménovými odborníky nebo přímé zapojení doménových odborníků do vývoje je další výhoda plynoucí z toho, že jazyk je navržen, tak aby opisoval problémovou doménu. Díky tomu se v něm doménovým odborníkům, kteří nejsou programátoři a neznají obecné programovací jazyky, lépe orientuje. S tím souvisí i další výhoda, doba učení nového jazyka, která bývá v případě DSL výrazně kratší[33, 32, 18].

DSL taky často slouží jako návod, jak o problému uvažovat. Jazyk blízký doméně nám umožní nezatěžovat se implementačními problémy a soustředit se na jádro problému. Svou blízkostí k doméně také jazyk pomáhá formovat

[2] Z anglického Domain Specific Language.

[3] V angličtině označováno jako General-purpose programming language. Mezi zástupce patří např. Java, Python nebo C++

a ustalovat termíny používané v komunikaci o problémech v dané doméně[16, 33, 32].

Zlepšení kvality a usnadnění kontrol a analýz vznikajícího kódu je u DSL možné díky nižší složitosti jazyka. To, že není výsledný kód zatížen implementačními detaily ulehčuje jeho kontrolu dalšími lidmi[33, 32].

Poskytnutí kvalitnějšího vývojového prostředí (IDE)[4], které rozumí danému jazyku je někdy zmiňováno jako další z výhod, která se však dá uplatnit jen u *externích* DSL[33, 32].[5]. Díky tomu, že dnes existují velmi kvalitní IDE i pro obecné programovací jazyky, je tato výhoda přinejmenším diskutabilní. Obstojnou kvalitu nástrojů totiž dostane i uživatel *interního* DSL nebo obecného programovacího jazyka. Pravdou však zůstává, že nástroje pro *externí* DSL mohou jít mnohem dál a poskytnout uživateli největší komfort. Je to však vyváženo značným úsilím, které je potřeba věnovat tvorbě takových nástrojů.

3.2.2 Nevýhody využití DSL

Hlavní jasnou nevýhodou je nutnost vytvořit DSL. Do čehož je potřeba vložit úsilí úměrné velikosti vytvářeného jazyka. Také je téměř jisté, že vytvořit DSL a následně v něm vyřešit daný problém bude trvat delší dobu, než vyřešit daný problém v obecném programovacím jazyce. Doba na vytvoření nového DSL je navíc výrazně delší pokud to děláme poprvé a nejsme seznámeni s žádným nástrojem pro tvorbu DSL[33, 18, 20].

Z tohoto důvodu může být velmi těžké obhájit náklady spojené s tvorbou DSL, pokud se jedná o jazyk, který najde využití jen v jednom projektu pro jeden konkrétní úkol. Vyšší počáteční náklady se však mohou vrátit, pokud se nám podaří stejný jazyk využit ve více projektech, případně z něj lze vyjít a využít ho v jiných projektech jen s drobnými úpravami. Rovněž pokud se business logika často mění a tyto změny je potřeba do projektu zapracovávat v průběhu údržby, tak se vyšší počáteční náklady vrátí v pozdější fázi projektu. Je však nutné počítat s rizikem, že se požadavky nebo trh změní na tolik, že jim náš jazyk nebude schopen vyhovět a naše investice tím může být zmařena[33].

Další nevýhodou je, že vytváření nového jazyka je jiný úkol, než vytváření softwaru a vyžaduje určité know-how. Před příchodem moderních nástrojů typu *language workbench* [6] mělo vytváření DSL pověst značně složitého a zdlouhavého procesu, při kterém bylo nutno vytvářet většinou tolik neoblíbené parsery a překladače. Moderní nástroje sice celý proces tvorby značně zjednodušily, avšak pořád je nutno se s nimi naučit, jak již bylo zmíněno výše. Navíc ani moderní nástroje nám nezaručí, že náš nový jazyk bude mít správnou míru abstrakce a bude snadno použitelný[33, 32, 18].

Vendor lock-in je další z nástrah se kterou je potřeba počítat, pokud se rozhodneme problém řešit s využitím DSL. Většina nástrojů pro tvorbu DSL je

[4] z anglického Integrated Development Environment

[5] DSL definované mimo hlavní jazyk více viz. 3.2.3

[6] Pojem zavedl Martin Fowler v roce 2005 a v komunitě kolem DSL se uchytil. Jeho český ekvivalent neexistuje proto ho používám v originále.

sice open-source, ale je však v podstatě nemožné přenášet náš jazyk z jednoho do druhého. Jediné, co je možné v případě přechodu na jiný nástroj zachovat je model nebo myšlenka stojící za naším jazykem. Vše ostatní je potřeba implementovat znovu. Nejedná se pouze o implementaci jazyka samotného, ale je nutné reimplementovat i všechno, co v daném jazyce vzniklo[33, 32].

Odmítnutí je další problém který ohrožuje přístup postavený na DSL. Nový jazyk může být sebelepší, ale pokud ho ani doménoví odborníci, ani programátoři nechtějí používat, nebude nám k ničemu[33].

Pokud již jsme v tvorbě nových DSL zběhlí a tvorba nového jazyka je pro nás rychlou záležitostí, můžeme být sváděni k tomu tvořit vždy nový DSL, místo využití již existujícího jazyka. Toho je potřeba se vyvarovat. Je totiž nutno pamatovat na to, že každý jazyk je potřeba vyvíjet, udržovat a podporovat, což se bude při velkém množství různých nekompatibilních jazyků dělat velmi obtížně a zabere to velké množství času. Jedná se však o problém hodně podobný případu knihoven a frameworků pro obecné programovací jazyky, kterému se dá předejít dobrou sebekázní[33].

3.2.3 Typy DSL

Pokud se rozhodneme pro náš problém vytvořit DSL, je možné zvolit dva zcela odlišné způsoby, jak to provést.

Můžeme vytvořit dojem nového jazyka pomocí prostředků, které nám poskytuje jazyk ve kterém vyvíjíme. Takovýto jazyk se označuje jako *interní*, někdy také jako embedded. Syntaxe *interních* DSL je však značně limitovaná programovacím jazykem, ve kterém nový jazyk vytváříme[18, 22].

Oproti tomu DSL, které označujeme jako *externí* nám umožňují vytvořit téměř libovolnou syntaxi[16]. Pro vytváření *externích* DSL se často vyžívají nástroje, které celý proces usnadňují. Nástrojů existuje celá řada od těch jednodušších jako je třeba generátor parseru (*ANTLR*)[7] po ty nejrozsáhlejší, které zpravidla označujeme jako *language workbench* jejichž zástupci jsou např. Jetbrains MPS nebo Xtext o niž se zmiňuje sekce 3.2.4[32, 33].

Interní

Jak již bylo zmíněno výše, *interní* DSL jsou postaveny uvnitř obecného programovacího jazyka. Tento jazyk bývá označován jako hostující jazyk. Syntaxe vzniklých jazyků je velmi omezena tím, co dovoluje syntaxe hostujícího jazyka. Některé jazyky jsou pro vytváření *interních* DSL vhodnější než jiné. Velmi tradiční je vytváření *interních* DSL v jazyce Lisp nebo v jazyce Ruby. Obecně jsou vhodnější spíše jazyky dynamicky typované jazyky ideálně i s dobrou podporou metaprogramovaní[8][33, 18, 22, 21].

Díky tomu, že *interní* DSL vycházejí, z obecného programovacího jazyka není potřeba pro ně vytvářet parsery a překladače. Máme také k dispozici

[7] `https://www.antlr.org/`

[8] Metaprogramování je technika, kdy program může pracovat s programem jako s daty a tím může např. modifikovat sám sebe[3]

všechny konstrukty hostujícího jazyka pro případ, že náš DSL nemá dostatečnou vyjadřovací sílu pro složitější problém. Na druhou stranu není prakticky možné pro *interní* DSL dodat větší podporu v podobě nástrojů pro vývoj, než je IDE pro hostující jazyk. To sice rozumí hostujícímu jazyku a dokáže napomoci více než prostý textový editor, ale nerozumí novým konstruktům a logice DSL[18, 22].

Hranice mezi tím, co je ještě abstraktní konstrukt v hostujícím jazyce a co už je *interní* DSL není nijak pevně daná. Zejména návrhový vzor *fluent interface* se pohybuje přímo na této hranici[16, 33].

```
Graph()
  .edge()
    .from("a")
    .to("b")
    .weight(12.3)
  .edge()
    .from("b")
    .to("c")
    .weight(10.5)
```

Zdrojový kód 3.1: Příklad užití návrhového vzoru *fluent interface*. Převzato z[29]

Někteří autoři tvrdí, že *interní* DSL nejsou opravdovými DSL, protože nedodávají podporu v nástrojích pro vývoj, což je zásadní pro zapojení doménových odborník. Považují za jediné opravdové DSL jen DSL *externí*[32].

Externí

Na rozdíl od *interních* DSL bývají *externí* DSL kompletně oddělené od obecného programovacího jazyka, v němž probíhá vývoj aplikace. Jsou tedy vyvíjeny pomocí jiných převážně specializovaných jazyků nebo nástrojů. Jejich syntaxe a konstrukty mohou být prakticky libovolné. Jsou omezeny pouze tím, že musíme být schopni syntaxi a konstrukty zpracovat parserem a překladačem[18, 22].

Interním DSL parser a překladač poskytuje hostující jazyk. U *externích* DSL je nutno parser a překladač vyvinout, což je největší úskalí využívání *externích* DSL. Tato překážka se dá odstranit využitím nástrojů pro vývoj DSL. To, o kolik bude vývoj *externího* DSL náročnější záleží do značné míry na sofistikovanosti využitých nástrojů. Nejpokročilejší nástroje pro vývoj DSL spadají do kategorie tzv. *language workbench*[18, 35].

Díky tomu, že v případě *externího* DSL máme celý proces tvorby jazyka pod kontrolou, můžeme uživateli jazyka poskytnout nejlepší nástroje pro vývoj v daném jazyce. S jejich tvorbou nám pomohou již zmiňované nástroje typu *language workbench*. Ty nám umožní vytvořit pro náš jazyk IDE, které snese srovnání s moderními IDE pro obecné programovací jazyky. Pokud

tvorbě našeho IDE věnujeme dostatečné úsilí, může dokonce IDE pro obecné programovací jazyky překonat[16, 33, 32, 22].

Poskytnutí kvalitních nástrojů pro vývoj a co možná nejtěsnější kopírování problémové domény je velmi důležité pro zapojení doménových odborníků do vývoje.

Možné důvody odmítnutí DSL[33, 26, 32, 34]:

- Nástroje pro vývoj se špatně používají.
- DSL je moc komplikovaný.
- DSL nereflektuje dobře doménu a nepoužívá pojmy v ní ustálené.
- DSL nepoužívá reprezentaci známou doménovým odborníkům.

Pokud tedy chceme, aby náš DSL využívali i doménoví odborníci, jež nejsou znalí programování, jsou *externí* DSL vhodnější volbou[32, 34, 18].

3.2.4 Nástroje pro tvorbu externích DSL

K vytváření *externích* DSL je možné zvolit zcela odlišné přístupy, které jsou určeny tím, jakou podobu bude náš výsledný jazyk mít. Rozlišujeme tři hlavní podoby nebo také styly práce s DSL[32].

- Grafické
- Textové
- Projekční editory

Protože jedním z našich cílů je co možná největší zapojení doménových odborníků do vývoje, budeme se soustředit zejména na nástroje, které poskytnou dobré nástroje pro vývoj v nově vytvořeném jazyce. Jedná se totiž o vlastnost bez které má nový jazyk jen malou šanci uspět[32, 18, 34].

V následujících sekcích se zaměříme na každou podobu samostatně a uvedeme i typické zástupce nástrojů, které do této kategorie patří.

3.2.5 Grafické

Většinou se začíná kategorií *textových* DSL, nicméně my začneme *grafickými* DSL, protože se pro náš problém nehodí a uvádíme je zejména pro úplnost.

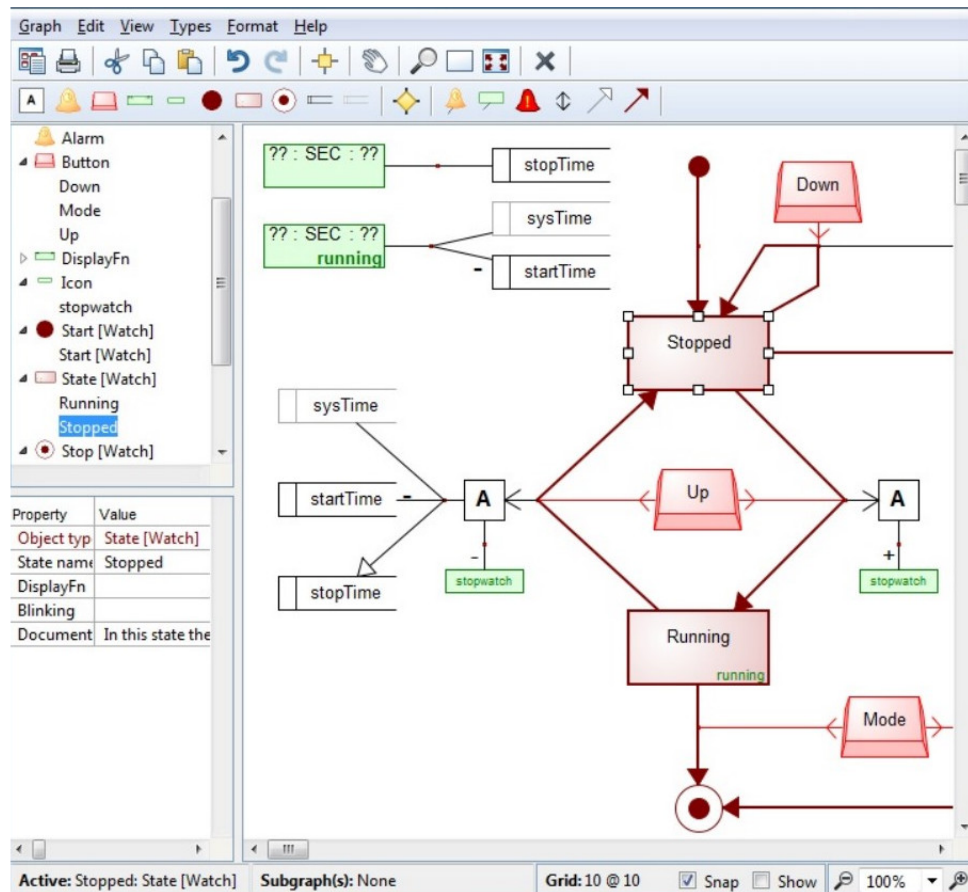
Grafické DSL k sobě musí mít editor, protože programování v *grafických* DSL probíhá formou vkládání a spojování grafických bloků, čímž se vytváří schéma a výsledná logika. Velmi časté jsou notace podobné jazyku UML nebo notace připomínající skládání puzzlů viz obrázek 3.2.

Hlavní výhodou *grafických* DSL je, že pokud jsou správně nastavena pravidla pro propojování bloků, nelze při programování vytvořit syntaktickou chybu.

Náš problém není snadné graficky reprezentovat. Také doménoví odbornicí jsou spíše zvyklí pravidla pro přestavování strojů zapisovat formou tabulky, případně textu. Vytvářet pro ně uměle grafickou reprezentaci by bylo spíše

kontraproduktivní, i z toho důvodu, že pokud nejsou zvyklí o problému uvažovat jako o nějaké formě diagramu, bylo by pro ně těžké takové diagramy vytvářet.

Níže si uvedeme dva zástupce nástrojů, které volí odlišnou strategii k vytváření jazyků. *Language workbench* MetaEdit+, kde notace výsledných jazyků vzdáleně připomíná UML, a knihovnu Blockly, která umožňuje vytvářet editory založené na spojování do sebe zapadácích bloků, což může připomínat spojování dílků puzzlů.



Obrázek 3.1: Ukázka programování v nástroji Metaedit+ Modeler převzato z [10].

MetaEdit+

MetaEdit+ se skládá ze dvou komerčních nástrojů MetaEdit+ Workbench a MetaEdit+ Modeler, které dohromady tvoří *language workbench*. Jazyk je vytvořen v nástroji MetaEdit+ workbench, kde definujeme koncepty, jejich vlastnosti, pravidla, kontroly a generátory. Programování ve vytvořeném jazyce potom probíhá v nástroji MetaEdit+ Modeler. Představu o tom, jak program vytvořený v MetaEdit+ Modeleru vypadá je možné získat z obrázku 3.1 [10, 32].

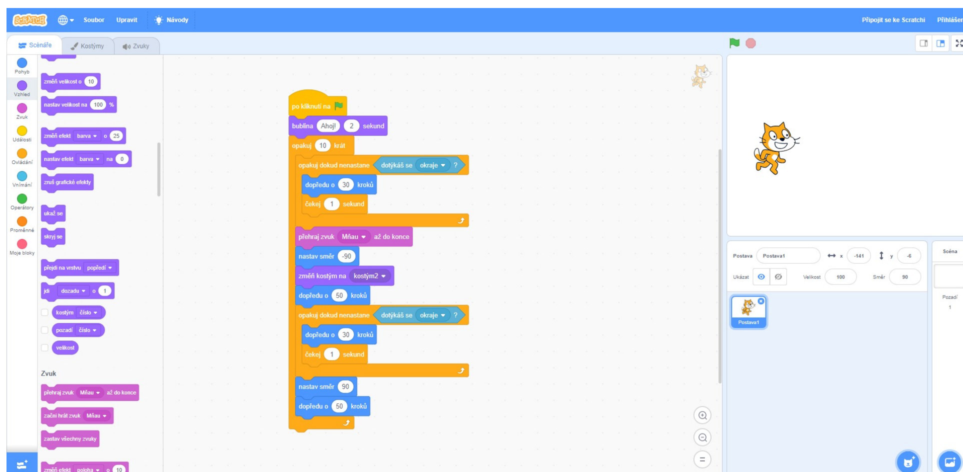
Blockly

Blockly je open-source knihovna vyvíjená společností Google LLC, která umožňuje vytvářet programy spojováním do sebe zapadajících bloků vzdáleně připomínajících puzzle. Viz obrázek 3.2. Knihovna poskytuje editor s bloky a framework pro vytváření překladačů do textových jazyků[27, 8].

Knihovna implementuje překladače do jazyků JavaScript, Python, PHP, Lua a Dart. Editor je implementován v JavaScriptu, takže jazyk postavený nad touto knihovnou lze bez problému provozovat ve webovém prohlížeči[27, 8].

Na knihovně blockly staví zejména programovací jazyky určené pro výuku programování dětí. Nejznámějším projektem je jazyk Scratch, určený právě pro výuku programování dětí. Další velmi známé projekty jsou Ozo Blockly určený pro programování mini-robotů ozobot a micro:bit určený programování stejnojmenného mikro-počítače[8].

Nevýhodou této knihovny je, že se v podstatě jedná o grafickou reprezentaci pseudokódu. Aby byl výsledný jazyk DSL, je potřeba vytvořit kompletní sadu vlastních bloků, což může být časově dosti náročné.



Obrázek 3.2: Ukázka programování v prostředí jazyka Scratch. Vytvořeno na [11].

3.2.6 Textové

Textové DSL a nástroje na jejich vytváření jsou nejznámější a nejčastěji používané. *Textové* DSL jsou nejvíce podobné obecným programovacím jazykům. Programy v nich vytvořené mají formu prostého textu[32]. Je však nutné, aby byla syntaxe, reprezentace, abstrakce a klíčová slova co nejvíce spjata s problémovou doménu. Jak takový program může vypadat ukazuje zdrojový kód 3.2. Vidíme zde ukázku jazyka pro mapování částí řádku v textových souborech do proměnných.

Aby bylo možné *textový* DSL používat, je potřeba vytvořit parser a lexer, generátor kódu z abstraktního syntaktického stromu (AST)[9], a volitelně

[9] AST je stromová reprezentace struktury syntaxe zdrojového kódu.

editor[12]. Editor je sice volitelnou součástí *textových* DSL. Programy v nich je možno vytvářet v obyčejném textovém editoru. Pokud se však rozhodneme jít tímto směrem nemůžeme očekávat zapojení doménových odborníků do vývoje[33, 32]. Proto se i pro vytváření *textových* DSL doporučuje využít nástroj typu *language workbench*.

```
mapping SVCL dsl.ServiceCall
  4-18: CustomerName
  19-23: CustomerID
  24-27 : CallTypeCode
  28-35 : DateOfCallString

mapping USGE dsl.Usage
  4-8 : CustomerID
  9-22: CustomerName
  30-30: Cycle
  31-36: ReadDate
```

Zdrojový kód 3.2: Příklad programu v *textovém* DSL. Převzato z[18]

Vzhledem k tomu, že se do vývoje snažíme zapojit doménové odborníky, jako příklady nástrojů pro vytváření jazyků zmíníme pouze nástroje typu *language workbench*. Je samozřejmě možné vytvořit *textové* DSL s podstatně jednoduššími nástroji. Například lze využít generátor parserů *ANTLR* a následující kroky vyvinout svépomocí[32]. Níže zmiňuji nástroj *Xtext* stavějící na ekosystému *Eclipse*, jež je asi nejznámější nástroj pro vytváření DSL vůbec a nástroj *Spoofax*.

Xtext

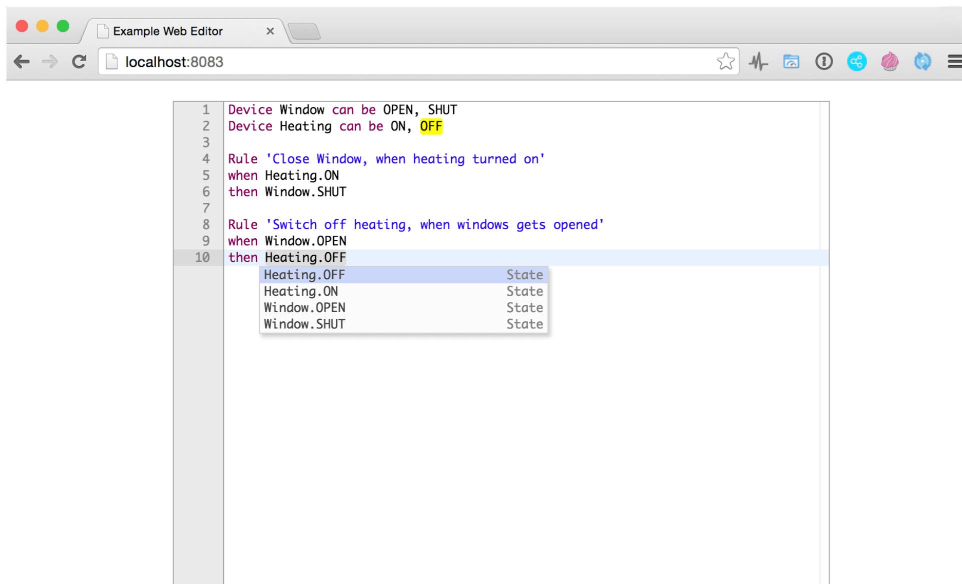
Xtext je nástroj pro vytváření *textových* DSL postavený na platformě *Eclipse*. V nástroji se definují pravidla gramatiky jazyka, z nich nástroj vygeneruje parser a editor. Posléze je potřeba ještě definovat generátor do obecného programovacího jazyka. Pro tento účel lze využít *Xtend*, který je určený ke generovaní jazyka Java verze 8 a celý úkol značně zjednoduší[32, 15, 7].

Jako editor vzniklého jazyka lze využít buď IDE Eclipse, pro které je podpora jazyka dodána pomocí plug-inu, nebo webový prohlížeč, viz obrázek 3.3 Nebo jakýkoliv editor podporující *language server protocol*[10].

Spoofax

Spoofax je podobně jako Xtext nástroj pro tvorbu *textových* DSL postavený na platformě *Eclipse*. Jazyky se v něm definují odlišným způsobem než v Xtext.

[10] Jedná se protokol skrze který client (editor) dostává od serveru (který rozumí jazyku) informace jak pomoci s vytvářením programu v daném jazyce. Jedná se informace typu doplňování kódu, najdi všechna užití apt. Mezi nejrozšířenější editory podporující *language server protocol* patří Visual Studio Code, IntelliJ IDEA a Vim. [2]



Obrázek 3.3: Ukázka programování v prostředí webového prohlížeče v jazyce vytvořeném v nástroji Xtext. Převzato z [7]

Pro definici jazyka se využívá sestava meta-DSL pro definici syntaxe, vazeb jmen, služeb editoru a definici transformací[15].

Pro nový jazyk můžeme jako editory využít Eclipse IDE nebo IntelliJ IDEA, pro které je podpora dodáná ve formě plug-inů. Podpora IntelliJ IDEA je zatím v experimentální fázi[5].

3.2.7 Projekční editory

Projekční editory jsou z neznámého důvodu často opomíjená skupina nástrojů pro vytváření DSL. Na rozdíl od textových editorů, kde je výsledný program uložen jako textový soubor, projekční editory reprezentují program jako model. Nejčastěji ve formě AST, který uživateli prezentují formou projekce[16, 32].

Uživatel v editoru tedy vidí například text, který může upravovat, avšak tyto úpravy vedou pouze k transformaci modelu z jednoho validního stavu do druhého. Text tedy nemusí být možné upravovat zcela libovolně, jako v textovém editoru. Práce s textovou reprezentací spíše připomíná vyplňování formuláře. AST je samozřejmě možné reprezentovat i jiným způsobem, například graficky. Díky tomu, že reprezentace zobrazená programátorovi je pouhá projekce, je možné definovat těchto projekcí více a dát uživateli na výběr, kterou projekci zvolí. Projekční editory v mnohém připomínají nástroje pro vytváření grafických DSL. Hlavní rozdíl je však v tom, že pro projekční editory je typické prezentovat AST programátorovi textovou projekcí a je možné těchto projekcí definovat více[17, 24].

Díky tomu, že programátor přímo upravuje AST odpadá jazykům, které byly vytvořeny v projekčních editorech, nutnost vytvářet parsery a lexery. Zároveň tím, že úpravy jsou jen změny z jedno konzistentního stavu AST

do druhého, není prakticky možné aby uživatel vytvořil syntaktickou chybu. Jedná se tedy o dvě značné výhody projekčních editorů[17, 32].

Za nevýhodu lze považovat nutnost zaškolit uživatele, jak s projekčním editorem pracovat. Další nevýhodou je vendor-lockin. Ten je u projekčních editorů asi největší ze všech uvedených druhů nástrojů. Je to způsobeno tím, že neexistuje žádný standardní formát pro ukládání AST. Poslední nevýhodu je horší verzovatelnost programů vytvořených v projekčním editoru oproti programu vytvořeném v textovém DSL[17].

Jako zástupce jsem vybral nástroje Jetbrains MPS a Whoole platform. Jako velmi nadějný je také často zmiňován nástroj Intentional Platform [18, 32], nicméně v roce 2017 společnost Microsoft a.s. koupila firmu Intentional Software, která stála za vývojem tohoto nástroje a od té doby není nástroj nikde k dispozici[1].

Jetbrains MPS

Jebrains MPS je *language workbench*, který umožňuje vytvářet DSL typu projekční editor. Je vhodný k vytváření celých rodin vzájemně propojených jazyků[32]. Umožňuje také při vývoji nového jazyka využít a rozšířit jeden čí více stávajících jazyků, což je vlastnost, alespoň mezi zkoumanými nástroji, unikátní.

Nástroj je postaven na stejné platformě jako všechna IDE od společnosti Jetbrains s.r.o., což zajišťuje dle mého názoru obstojný vzhled vytvořených editorů. Nástroj je dostupný pod open-source licencí Apache 2.0. Jazyky se v nástroji definují pomocí aspektů popisujících vlastnosti vyvíjeného jazyka jako jsou struktura, editor a jeho akce, podmínky, typový systém, generátor[11] a další. Většina aspektů k sobě má přidružen DSL, v němž se aspekt definuje[25]. Podrobněji se jednotlivým aspektům a celému nástroj věnuje kapitola 4.

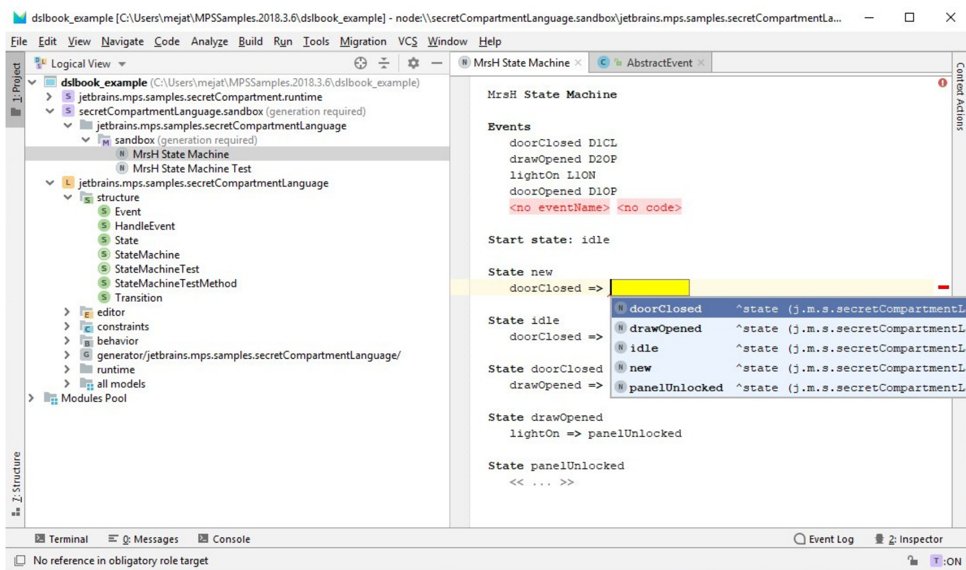
Projekční editory vytvořené v tomto nástroji mohou využívat textové, tabulkové, symbolické a grafické prvky a jejich různou kombinaci[33, 15, 25]. Nejběžnější je však využití textových prvků, což dokládá i obrázek 3.4, jež ukazuje programování v jednom z ukázkových jazyků, které jsou distribuovány společně s MPS.

Nejedná se sice o tolik rozšířený nástroj jako v případě nástroje *Xtext*, ale pořád se jedná o jeden z nejpoužívanějších nástrojů pro tvorbu DSL. Zároveň jde asi o nejpokročilejší nástroj pro vytváření projekčních editorů s velmi dobrou dokumentací, která obsahuje i základní výukový kurz. Ta je zajisté potřeba, protože vstupní bariera tohoto nástroje je o něco vyšší[32, 33].

Whoole platform

Whoole platform je nástroj pro vytváření projekčních editorů, který umožňuje snadno adaptovat existující datové formáty, které popisuje pomocí grafické notace gramatiky[32, 15].

[11] Jedná se o pravidla transformací z jednoho modelu do druhého. Velmi často se využívá pro převod DSL do *Base Language*, který následně zajistí převod do jazyka Java.



Obrázek 3.4: Ukázka programování v prostředí MPS.

Jedná se o nástroj, který podobně jako spousta ostatních staví na platformě Eclipse[6]. Dle mého názoru jsou však editory produkované tímto nástrojem vzhledově zastaralé a nevzbuzují dojem příjemné, jednoduché a pohodlné práce.

Dokumentace tohoto nástroje je poněkud řídká[32] a přesto, že vývoj nástroje zjevně probíhá[12], dokumentaci ani komunikaci s okolím nikdo neudržuje[13].

3.3 Zvolený nástroj

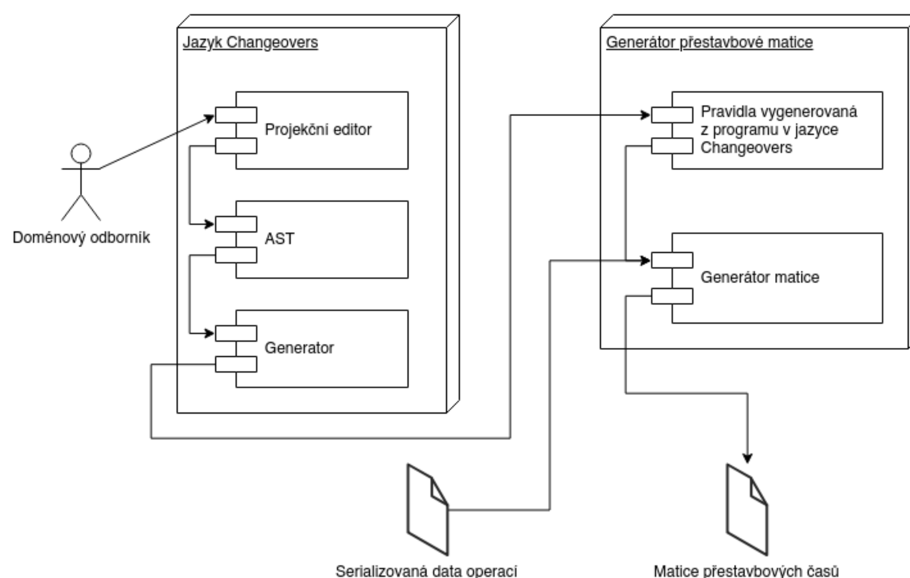
Jak jsem již zmiňoval v sekci 3.2.5 grafické DSL se pro náš problém moc nehodí. Grafická reprezentace problému by nebyla přirozená a doménoví odborníci také problém popisují spíše textově nebo ve formě tabulek. Pokud nejsou doménoví odborníci zvyklí problém popisovat grafickou formou, bylo by pro ně velmi obtížné grafickou reprezentaci problému vytvářet a hrozilo by odmítnutí jazyka.

Pokud bych však řešil jiný problém, pro který by se grafická reprezentace hodila nebo bych pracoval s doménovými odborníky, kteří například rozumí logickým obvodům, rozhodně bych uvažoval o grafickém jazyce s využitím knihovny Blockly.

Z výše uvedeného vyplývá, že v úvahu připadaly jen textové DSL a DSL založené na projekčním editoru. Rozhodl jsem se zvolit řešení postavené na projekčním editoru a to hned z několika důvodů. Doménovým odborníkům může práce v projekčním editoru vzdáleně připomínat práci v ERP systému,

[12] Poslední commit na Githubu proběhl před 5 dny.

[13] Domovská stránka projektu[6] byla naposledy aktualizována v roce 2017 a účet na sociální síti Twitter vykázal poslední aktivitu v květnu roku 2016.



Obrázek 3.5: Pohled na fungování navrhovaného řešení.

kde se vyplňují různé formuláře, na což jsou zvyklí. Dále se vyvarujeme toho, že doménoví odborníci budou dělat syntaktické chyby. Dále není potřeba vytvářet parser a lexer, což je často považováno za velmi složité a časově náročné. V neposlední řadě dostaneme snáze přehlédnou reprezentaci programů vzniklých v našem DSL, což je výhodné i v případě, kdy má doménový odborník pouze validovat business logiku ve vzniklém programu.

Z nástrojů pro vytváření DSL s projekčními editory jsem se rozhodl zvolit **Jetbrains MPS**. V jeho prospěch mluvilo zejména to, že se jedná o jedničku mezi projekčními editory a dvojku mezi nástroji typu *language workbench*. Dále také to, že je k dispozici dobrá dokumentace, včetně výukového kurzu. Jako poslední argument uvedu, že mně zaujaly možnosti editorů vytvořených v tomto nástroji.

3.4 Návrh řešení

Pro řešení problému budou vytvořeny dva spolupracující systémy. Provázání systému je zachyceno na obrázku 3.5.

Prvním z propojených systémů je doménově specifický jazyk typu projekční editor. K jeho vytvoření bude využit nástroj MPS zvolený v předchozí sekci. Tento jazyk bude pojmenován *Changeovers*. Programováním v jeho editoru budou doménový odborníci vytvářet pravidla přestaveb strojů. Pravidla budou v jazyce uloženy ve formě AST. Pravidla se s pomocí generatoru jazyka *Changeovers* transformují do jazyka Java. Transformovaná pravidla využívá druhý systém.

Druhý systém je realizován v jazyce Java a slouží k sestavení matice

sekvenčně závislých přestavbových časů. Na obrázku 3.5 je označen jako Generátor přestavbové matice. Systém načte data plánovaných operací, sestaví z nich všechny možné dvojice a na každou z nich aplikuje pravidla vygenerovaná z prvního systému. Tím získá přestavbové časy, které zaznamená do matice. Tuto matici uloží do výstupního souboru, aby ji mohl dále využít systém pro plánování výroby.

Kapitola 4

Vývoj jazyka v Jebrains MPS

Jetbrains MPS je nástroj, jež umožňuje vytvářet jazyky typu *projekční editor*. V tomto případě se tedy jazyk vytváří ve stejném nástroji, ve kterém potom probíhá samotné programování.

Jelikož programování se skládá ze série transformací AST z jednoho validního stavu do druhého, je potřeba důsledně popsat, jak validní AST nového jazyka vypadá a zajistit jeho projekci do editoru. Případně i připravit netriviální transformace, jež uživateli jazyka usnadní programování. V neposlední řadě, je také potřeba zajistit možnost vykonávání programů v novém jazyce, což je možné dvěma způsoby. Buď můžeme implementovat interpreter nového jazyka nebo zajistit transformaci nového jazyka do obecného programovacího jazyka.

Nástroj MPS k tomu prostupuje pomocí *aspektů* (4.1), kdy každý aspekt popisuje jazyk z jiného pohledu. Například z pohledu struktury, projekce k uživateli, typového systému a mnoho dalších.

Velkou výhodou MPS je, že umožňuje *rozšiřovat jazyky* (4.2), které jsou již pomocí MPS vytvořeny a tím velmi usnadňuje tvorbu nových jazyků a snižuje množství duplicitního kódu.

4.1 Aspekty

Jak již bylo řečeno, MPS využívá aspekty k popisu vlastností tvořeného jazyka. Většina aspektů se popisuje pomocí DSL vytvořeného speciálně k popsání daného aspektu. Jedná se například o aspekty *structure*, *editor*, *constrains*, *behaviour*, *generator* a některé další, které jsem však v rámci této práce nevyužili.

Všechny aspekty jsou sdružené dohromady přes jeden konstrukt jazyka, takže je velmi jednoduché přecházet mezi různými pohledy na jeden konstrukt jazyka.

Níže budou velmi stručně popsány aspekty, které byly využity při tvorbě jazyka z kapitoly 5. Pro hlubší porozumění je možné využít oficiální dokumentaci MPS[1].

[1] `https://www.jetbrains.com/help/mps/mps-user-s-guide.html`

4.1.1 Structure

Structure je nejdůležitějším aspektem nového jazyka. Jedná se také o jediný povinný aspekt, kterým je nutno každý konstrukt popsat[2].

Pomocí aspektu *structure* totiž dáváme vědět, že nějaký konstrukt v novém jazyce vůbec existuje. *Structure* tedy říká, z jakých uzlů je možné AST našeho jazyka stavět a jak je možné tyto uzly propojovat.

Aspekt *structure* v mnohém připomíná definici třídy v objektově orientovaném programování. Každý uzel může mít vlastnosti, což jsou místa pro uložení hodnot primitivních typů. Dále potomky, což jsou instance ostatních konstruktů. Jejich počet je určen pomocí kardinality. A jako poslední reference, což jsou odkazy na jiné uzly AST.

Konstrukty mohou rozšiřovat jiné konstrukty, či implementovat rozhraní. Je dokonce možné vytvářet abstraktní konstrukty, od nichž není možno vytvářet instance, ale slouží jen jako základ pro rozšiřující konstrukty. Inspirace objektově orientovaným programování je zde zjevná.

4.1.2 Editor

Druhým nejdůležitějším aspektem je *editor*. Ten popisuje to, jak uživatel s konstruktem pracuje v editoru. A to nejen pomocí projekce z AST do editoru, ale umožňuje definovat i velké množství nejrůznějších akcí, které dokáží programování velmi usnadnit. Jedná se o nejvíce rozsáhlý aspekt.

Základním stavební kamenem editorů jsou pole. Ty mohou být různých druhů ze základních lze zmínit:

- **kontejnery** - obsahují v sobě další pole
- **konstantní pole** - obsahuje předem definovaný text
- **pole pro zadávání či výběr hodnot** - různé druhy
- **pole pouze pro čtení** - obsahuje text daný výpočtem z vlastnosti instance konstruktu

Vzhled všech polí editoru je možné ovlivňovat pomocí stylů. Dále může editor obsahovat řídící prvky, které například ovlivní to, kdy se má dané pole zobrazit.

Editory lze také skládat a využívat tak editory jiných konstruktů jež mají s konstruktem jehož editor tvoříme vztah v AST.

Definici uživatelský akcí na tomto místě rozvádět nebudu. Existuje jich velké množství a v jazyce z kapitoly 5 jich je využito jen velmi málo. Získat povědomí o akcích, jakožto i o dalších vlastnostech editoru lze v oficiální dokumentaci.

Na tom, jak kvalitně se podaří tento aspekt zpracovat závisí, jak pohodlné bude vytváření programů v novém jazyce a potažmo i, jestli nebude náš jazyk uživateli odmítnut.

[2]Editor je také nutný proto, aby bylo možné s konstruktem pracovat. MPS však umí využít základní editor pokud není žádný jiný definovaný.

4.1.3 Constrains

Aspekt *constrains* popisuje další omezující podmínky, jak musí AST vypadat. Zejména to, jak mohou být uzly propojeny a jaké hodnoty mohou nabývat jejich vlastnosti.

V jazyce z kapitoly 5 se využívá pouze k omezení rozsahu, ze kterého mohou být některé konstrukty referencovány.

4.1.4 Behaviour

Behaviour je jednoduchý aspekt, který umožňuje konstruktu definovat metody, které je poté možno využívat z ostatních aspektů.

4.1.5 Typesystem

Aspekt *typesystem* má tři hlavní využití. Umožňuje určovat pravidla, podle kterých instance získá svůj typ. Omezovat jakého typu může instance na daném místě být a provádět kontroly kódu připomínající statickou analýzu.

Manipulace a vynucování typů se provádí pomocí *inference rule*. Statická analýza kódu pomocí *checking rules*.

4.1.6 Generator

Generator slouží k transformaci AST našeho jazyka do AST cílového jazyka. Pomocí generátoru se tedy převádí program v jednom jazyce do programu v jiném jazyce. Nejčastěji se tak činí proto, že cílový jazyk má definovaný aspekt *TextGen*. Jeho využitím získáme program v obecném programovacím jazyce, jež lze provádět nebo dále využít.

Samotný *Generator* se skládá ze dvou částí. Z pravidel generátoru a konfigurace mapování.

Konfigurace mapování je jednoduchý konstrukt, který v sobě sdružuje všechna pravidla generátoru, pre a post processing scripty a mapovací štítky.

Mapovací štítky jsou pomocné struktury, do kterých se nejčastěji ukládá na jakou instanci nového jazyka byla transformována instance konstruktu v původní jazyce. To může být potřeba znát v dalších fázích transformace.

Pravidla generátoru se skládají ze dvou částí, z předpokladu a důsledku. Předpoklad určuje, na který konstrukt starého jazyka se má důsledek aplikovat. Umožňuje definovat i jednoduché podmínky, které rozhodují, jaký důsledek použít, pokud jich má jeden konstrukt definovaných více. Důsledky jsou ve většině případů šablony v cílovém jazyce, do kterých se s pomocí maker doplní vlastnosti konstruktu.

Pravidel generátoru existuje více druhů, ale jazyk z kapitoly 5 vyžívá jen *root mapping rule* a *reduction rule*.

Stejně tak existuje i více druhů šablon, ale využity jsou jen *root template*, která v cílovém jazyce vytváří kořenové uzly AST. A *TemplateDeclaration*, která vytváří jiné než kořenové uzly AST.

4.1.7 Další aspekty

MPS obsahuje ještě další aspekty jako *Data flow*, *TextGen*, *Migrations*, *Intentions*, *Refactoring* a některé další. Žádný z nich však v jazyce z kapitoly 5 nebylo třeba využít.

4.2 Rozšiřování jazyků

MPS umožňuje na rozdíl od ostatních prostředí pro vytváření DSL pracovat s již existujícími jazyky definovanými v MPS. Je možné existující jazyky spojovat a rozšiřovat. Díky tomu, že MPS pracuje pomocí projekčního editoru přímo s AST nevadí ani to, že mají dva využité jazyky stejná klíčová slova. Vytvořením nového prvku v editoru totiž vytváříme nový uzel AST a MPS musí už při vytváření vědět, o jaký uzel se jedná. Editor tedy musí nechat uživatele zvolit, z kterého jazyka chce konfliktní prvek použít. Podle metainformací uzlu pak MPS daný prvek jednoznačně identifikuje.

Možnosti rozšiřovat existující jazyky jsem využil i při tvorbě jazyka pro definici přestavbových času z kapitoly 5. Ten rozšiřuje jazyk *Base Language*, jež je přímou součástí MPS a zároveň se jedná o nejčastěji rozšiřovaný jazyk[23].

4.2.1 BaseLanguage

Zjednoduše řešeno se jedná o implementaci jazyka Java verze 6 v MPS, do kterého byly postupně přidány i vlastnosti verze 7 a 8. To znamená, že *Base Language* umožňuje v MPS vytvářet programy se stejným vzezřením jako mají programy napsané v jazyce Java, ve formě AST a ty poté převádět pomocí TextGen na zdrojové soubory jazyka Java. Jak již bylo zmíněno v 4.1.6, generátor umožňuje převádět jeden AST na jiný, čehož využívá i jazyk *Changeovers*, který definuje pouze generátor do jazyka *Base Language*. Ten následně zajistí převod do zdrojových souborů jazyka Java. Převod na zdrojové soubory jazyka Java je také velmi časté využití *Base Language*

Z dalších konceptů jazyka *Base Language* využívá jazyk *Changovers Expression* a *Type*.

Expression

Expression je konstrukt, který může být složený z referencí na proměnné, hodnot a operací mezi nimi. Při provádění programu je výsledkem hodnota některého datového typu. Např. *Expression* $1 + 2$ se při provádění vyhodnotí na hodnotu 3 datového typu integer. Výsledný datový typ je znám již v době překladu a výsledná hodnota může být známá až za běhu programu.

Type

Type je nadřazen všem datovým typům implementovaným v *Base Language*. Rozšířením konceptu Type je tedy možno definovat nové datové typy, které

budou s *Base Language* kompatibilní, toho jsme však v našem případě nevyužili. Místo toho jsem umožnil některým novým konceptům přejímat datové typy implementované v *Base Language*. To umožňuje nad novým konceptem provádět stejné operace, jako nad datovým typem z *Base Language* bez toho, aby bylo nutné tyto operace znovu implementovat.

Kapitola 5

Jazyk Changeovers

Jak již bylo naznačeno v sekci 3.3 pro řešení hlavní části problému, tedy pro definování pravidel jednotlivých přestaveb strojů, jsem se rozhodl vytvořit systém postavený na DSL typu projekční editor. Tento systém jsem vytvořil v nástroji Jetbrain MPS. Vytvořený DSL jsem nazval *Changeovers*.

5.1 Základní vlastnosti jazyka

5.1.1 Výchozí jazyk

Changeovers staví na jazyce *Base Language*, který je součástí MPS. Jedná se o implementaci jazyka Java verze 6 s rozšířením o vlastnosti verze 7 a 8 [23]. Z jazyka *Base Language* jsou využity *Expression*, *Typy* a *TextGen*. Podrobněji se využitým vlastnostem věnuje sekce 4.2.1.

5.1.2 Lokalizace Changeovers

Pro lepší přilnutí k problémové doméně jsem se rozhodl, že všechny **mnou vytvořené**, pro uživatele viditelné části jazyka budou v češtině[1]. Uživatelé jsou totiž zvyklí o problému uvažovat v češtině a program vytvořený z velké části v češtině pro ně bude přehlednější a budou se v něm lépe orientovat. Části které jsou viditelné zejména vývojářům jazyka jsou však v angličtině.

5.1.3 Changeovers z pohledu uživatele

Uživatel pracuje s jazykem *Changeovers* poměrně jednoduše. Pro jeden výrobní podnik vytvoří jeden pojmenovaný program. Ukázka rozpracovaného velmi jednoduchého programu je uvedena na obrázku 5.1. Pro každý stroj nebo skupinu stejných strojů vytvoří jeden stroj v tomto programu (na obrázku blok Laser). Pro každý stroj v programu uvede jméno stroje a kód stroje[2] [3].

[1] Angličtina se však vyskytuje v rozhraní editoru MPS a v konstruktech přejatých z jazyka *Base Language*. Jedná se zejména o jména datových typů a metody datového typu string jako `.equals()` či `.startsWith()`. U metod se však předpokládá, že je budou využívat jen zkušenější uživatelé.

[2] Označení stroje v ERP systému výrobního podniku.

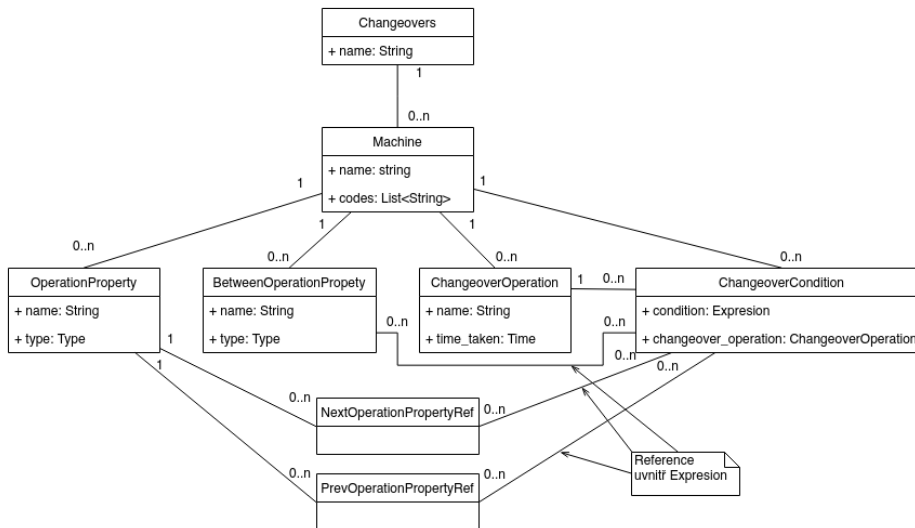
[3] Lze uvést několik kódů, jeden kód pro každý stroj z popisované skupiny.

Dále uvede vlastnosti operace prováděné na stroji (na obrázku například `síla plechu`) a mezioperační vlastnosti prováděné na stroji, které mají vliv na úkony přestavby stroje. Dále uvede úkony přestavby a délku jejich trvání (na obrázku `výměna hlavy` trvající 15 minut), a jako poslední vyplní podmínky, za kterých je nutno úkony provést (na obrázku `mění se materiál`). Jedna podmínka vždy spustí jeden úkon. V podmínkách je možné využívat definované vlastnosti.

V programu z obrázku jsou popsány dva stroje Laser a Frézka. Zatímco popis přestaveb Laseru je dokončen, popis Frézky teprve začíná. Z ukázky je patrné, že programovaní v jazyce *Changeovers* vzdáleně připomíná vyplňování formuláře.

```
Prestavbove casy pro zakaznika Výrobní podnik A
 Stroj: Laser

  Kód: laser6540, laser54321

  Vlastnosti operace:
    sila plechu : int, material : string

  Vlastnosti mezi operacemi:


  Přestavby:
    výměna hlavy => 15 min
    čístění => 30 min

  Podmínky přestaveb:
    Druh: materiálové
      1)  sila plechu jde skrz hranici 8 => výměna hlavy
        (15 min)

      2)  mění se material => čístění
        (30 min)

 Stroj: Frézka

  Kód: Zadejte kód

  Vlastnosti operace:
    Zadejte jméno : Vyberte typ

  Vlastnosti mezi operacemi:


  Přestavby:
    Zadejte jméno => Zadejte čas Vyberte jednotku

  Podmínky přestaveb:
    Druh: Zadejte jméno
      1)  Zadejte podmínku => Vyberte přestavbu
```

Obrázek 5.1: Jednoduchý rozpracovaný program v jazyce *Changeovers*.



Obrázek 5.2: Zjednodušený class diagram jazyka *Changeovers*.

5.1.4 Struktura programů v Changeovers

Představu o struktuře programů v jazyce *Changeovers* poskytuje obrázek 5.2. Na obrázku jsou kvůli větší přehlednosti vynechány méně důležité prvky jazyka. Z obrázku je vidět, že jeden program obsahuje definici více strojů (Machine). Každý stroj má definované vlastnosti operace (OperationProperty), vlastnosti mezi operacemi (BetweenOperationProperty), přestavbové úkony (ChangeoverOperation) a podmínky za nichž je potřeba úkony provést (ChangeoverCondition).

Podmínky přestaveb jsou místo, kde se všechno spojí dohromady a začne dávat smysl. Jedna podmínka spouští jeden úkon. Jeden úkon však může být spuštěn více různými podmínkami a klidně i vícekrát. Ve výsledku se však každý úkon projeví jen jednou[4].

V podmínkách je možné využívat reference na hodnoty vlastností operace a to jak předchozí, tak následující operace. Dále je možné využívat reference hodnoty vlastností mezi operacemi, předdefinované zkratky a všechny *Expression* z *Base Language*.

5.1.5 Skupiny konstruktů v Changeovers

Níže si představíme konstrukty jazyka *Changeovers*, které jsou rozděleny do skupin, podle toho, jaký mají v jazyce význam. *Kontejnery* (5.2) v sobě zabalují ostatní prvky jazyka a poskytují hranice kontextu. *Definice časů* (5.3) umožňuje definovat vlastní časové jednotky a obsahuje sadu předdefinovaných jednotek. *Úkony přestaveb* (5.4) zajišťují definici úkonů přestaveb. *Podmínky přestaveb* (5.6), jež umožňují definovat podmínky za kterých je potřeba

[4]Toto neplatí pro vícenásobné úkony přestaveb. Ty jsou však speciálním případem. Více bude vysvětleno dále.

provést úkon přestavby. *Vlastnosti procesu výroby* (5.5), ve kterých se definuje jaké vlastnosti má výrobní proces na daném stroji. Poslední kategorií jsou *zkratky* (5.7), které umožňují nejčastější podmínky přestaveb zapsat snáze a pro uživatele více srozumitelně. Popisy půjdou po jednotlivých aspektech konstruktu a budou se zaměřovat hlavně na odlišnosti a zajímavé prvky.

5.2 Kontejnery

Kontejnery jsou prvky, které mají v jazyce zejména strukturní význam. Zapouzdří do sebe další konstrukty jazyka a nesou dodatečné informace jako například jméno. Některé z nich také slouží jako takzvaný *scope provider*, omezují tedy rozsah použitelnosti prvků v nich obsažených. Například vlastnost operace definovanou uvnitř jednoho stroje, nelze použít v podmínce přestavby jiného stroje, protože stroj je *scope provider*.

V jazyce *Changeovers* se vyskytují kontejnery *Changeovers* (5.2.1), *Machine* (5.2.2) a *ChangeoverConditionSection* (5.6.1), který je však zařazen do skupiny *Podmínky přestaveb* (5.6).

5.2.1 Changeovers

Changeovers konstrukt pojmenovaný stejně jako celý jazyk je hlavním ze dvou root konceptů. Založením nové instance Changeovers vytvoříme nový program v jazyce *Changeovers*. Jedná se tedy o kořenový uzel AST každého programu v jazyce *Changeovers*. Všechny ostatní instance konstruktů použité v programu se poté navazují jako potomci tohoto uzlu.

Structure

Struktura konstruktu *Changeovers* je velmi jednoduchá. Jak již bylo řečeno výše jedná se root koncept. Instance konstrukt v sobě může obsahovat 0 až n konstruktů *Machine*. Tedy každý program může nést 0 až n popisů přestaveb strojů. Dále už má každá instance jen přiřazené jméno. Toto jméno je tedy i jméno programu.

Editor

Editor konstruktu je také velmi jednoduchý. Skládá se jen z řádky nadpisu, jež obsahuje statický text „Přestavbové časy pro zákazníka“ a pole pro zadání jména[5]. A dále obsahuje cyklus, který vkládá editory všech konstruktů Machine[6].

V něm se nachází největší zajímavost celého editoru a to je šedá oddělovací čára vkládaná mezi editory konstruktu Machine. K čemuž je využit separátor, jež umožňuje cyklus vkládat mezi jednotlivé prvky. Vkládá se dostatečné množství znaků „-“ a stylem je nastavena barva pozadí i textu na šedou, což se

[5]Na obrázku 5.1 hned první řádek.
[6]Na obrázku 5.1 bloky začínající řádkem „Stroj:“ a končící šedou horizontální čarou.

```
root template
input Changeovers

public class Changeovers {

  private HashMap<String, MachineRulles> machineRules;

  public Changeovers() {
    machineRules = new HashMap<String, MachineRulles>();
    $LOOP$[$LABEL$ machineRulesIntaces[MachineRulles $[machineR] = new ->$[MachineHelperImp]()]; ]

    $LOOP$[$LOOP$[machineRules.put("$[machine_code]", ->$[machineR]); ]]
  }
  public MachineRulles getMachineRules(string machineCode) {
    return machineRules.get(machineCode);
  }

  $LOOP$[$COPY_SRC$[]]

}
```

Obrázek 5.3: Root template. Generátor konstruktu Changeovers.

vizuálně projeví jako šedá oddělovací čára. Tu je možno vidět na obrázku 5.1, stejně jako zbytek editoru, který vidí uživatel při tvoření programu.

Generator

Nejzajímavějším aspektem je rozhodně generátor. Jelikož je konstrukt *Changeovers* jediný root koncept, který se transformuje do *Base Language*,[7] jedná se o jediný *root template* v celém jazyce. To znamená, že toto je jediný konstrukt, ze kterého po generování vznikne soubor. Všechny ostatní konstrukty budou vygenerovány uvnitř tohoto souboru.

Konstrukt se do *Base Language* přeloží jako třída se jménem `Changeovers`. Třída má jednu metodu, která pro kód stroje vrátí třídu s pravidly jeho přestaveb. To je možné díky tomu, že se v konstruktoru třídy nejprve vytvoří instance tříd s pravidly a poté se naplní do mapy jejíž klíčem jsou kódy stroje a hodnoty instance příslušných tříd s pravidly. Tyto třídy jsou získány přeložením všech konstruktů *Machine* pomocí jejich generátorů. Hlavní struktura generátoru je vidět na obrázku 5.3[8].

5.2.2 Machine

Kontejner *Machine* je hlavním organizačním útvarem jazyka. Zapouzdřuje vše, co je potřebné pro sestavení matice přestavbových časů jednoho stroje [9]. Jeden program většinou obsahuje několik instancí tohoto konstruktu.

Jedna instance obsahuje kompletní popis přestaveb jednoho stroje, což zahrnuje:

- jméno stroje

[7]Druhý root koncept se netransformuje. Je tak jednoduchý, že se během překladu rovnou přeloží na počet sekund.

[8]Nastavení maker generátoru je skryto v okně inspektoru a nelze je na obrázku zachytit.

[9]Jednou instancí je možno popsat i skupinu strojů se stejnými pravidly přetaveb. To, jestli se jedná o jeden stroj nebo o skupinu strojů poznáme podle počtu uvedených kódu. Technicky je vlastně popis jednoho stroje popis skupiny strojů obsahující pouze jeden stroj.

- kód stroje
- vlastnosti operací prováděných na stroji
- předpočítané vlastnosti mezi dvěma po sobě jdoucími operacemi
- přestavbové úkony s délkou jejich provádění
- podmínky spouštějící přestavbové úkony

Z pohledu AST se jedná o jediné přímé potomky root uzlu *Changeovers*, což vidíme na obrázku 5.2. Jedna instance je vyobrazena na obrázku 5.1. Instance začíná řádkem `Stroj: Laser` a končí prázdným řádkem nad šedou čárou.

Structure

Ze strukturního pohledu musí mít každá instance jméno a obsahovat 1 až n kódu stroje, které jsou typu string a 1[10] kontejner *Operation*, který v sobě sdružuje vlastnosti operace. Všech ostatních navázaných konstruktů musí být od 0 do n. Konkrétně se jedná o konstrukty *BetweenOpsProp* (předpočítané vlastnosti mezi operacemi), *ChangeoverOperation* (přestavbový úkon) a *ChangeoverConditionSection* (podmínka spouštějící přestavbový úkon).

Machine je *scope provider*, což je velmi důležité. Díky tomu, je možné v podmínkách referencovat pouze vlastnosti daného stroje nikoliv jiného stroje.

Typesystem

V typovém systému má *Machine* definovaná dvě kontrolní pravidla. První zajišťuje unikátnost jmen strojů v jednom programu. Druhé kontroluje, zda má instance vyplněné alespoň kód.

Generator

Generator konstruktu *Machine* je nejdůležitější pro transformaci jazyky *Changeovers* do *Base Language*. Krom svého vlastního konstruktu transformuje nebo připravuje potřebné pro další konstrukty.

Celý generátor je zachycen na obrázku 5.4. Níže budu v popisu postupovat shora dolů.

Samotný *Machine* se do *Base Language* transformuje jako třída, která implementuje rozhraní `MachineRules`[11].

Uvnitř třídy se připravují proměnné. První cyklus připraví HashMapu pro každou instanci konstruktu *BetweenOpsProp*. Druhý cyklus pro každou instanci *OperationProperty* pomocí jejího generátoru založí instanci třídy `ChangeoversOperation`.

[10] Kontejner může být prázdný, ale musí být přítomen.
[11] K nahlédnutí v příloze B.

V konstruktoru třídy je zajímavý pouze poslední řádek, který všechny instance `ChangoversOperation` přidá do ArrayListu.

Metoda `getChangeoverTime` slouží k získání přestavbového času pro dvě zadané operace. V cyklu uvnitř této metody se transformují instance *ChangeoverCondition*, detaily jsou popsány v jejich sekci. Forcyklus na konci metody slouží k vyhodnocení přestavbového času sečtením časů potřebných na všechny spuštěné úkony.

Uvnitř statické třídy `Task` se v cyklu transformují všechny instance *OperationProperty* na atributy této třídy s odpovídajícím typem a jménem[12].

Nenápadný řádek `$COPY_SRCL$[]` spustí generátor všech instancí *BetweenOpsProp*.

Metoda `getTaskArrayClass` slouží pro účely deserializace operací z formátu *json* do instanci třídy Task. Více se tomuto věnuje kapitola 6.

5.3 Definice časů

U přestavbových úkonů je potřeba uvést dobu trvání. Pro tyto potřeby slouží dva jednoduché konstrukty *TimeUnit* (5.3.1) reprezentující časovou jednotku a *Time* (5.3.2) reprezentující samotný čas trvání.

5.3.1 TimeUnit

Jednoduchý konstrukt *TimeUnit* reprezentuje časové jednotky. Každá jednotka je definována jménem, zkratkou a počtem sekund na jednu tuto jednotku.

V jazyce jsou předpřipravené 3 základní jednotky (sekunda, minuta a hodina). Uživatel si však může u svých programů definovat nové jednotky, což umožňuje definice jednotky jako samostatného konstruktu.

Z pohledu AST jde o kořenový uzel na nějž se odkazují uzly typu *Time*. *Time Unit* se do *Base Language* nepřevádí, slouží jen pro přepočet doby trvání v jednotce na dobu trvání v sekundách.

Structure

Kromě toho, že se jedná o root koncept, je struktura velmi jednoduchá. Jednotka se skládá pouze ze jména a zkratky, obě typu string, a počtu vteřin na jednu jednotku, který je typu integer.

Editor

Editor je také velmi jednoduchý obsahuje v podstatě jen tři pole pro vyplnění hodnot a pomocné texty. Jak takový editor pro uživatele vypadá ukazuje obrázek 5.5.

[12]Jméno musí projít transformací aby se znej stalo validní jméno proměnné v *Base Language*.

```
template map_Machine
input    Machine

parameters
<< ... >>

content node:
<TF [public class $[Machine] implements MachineRulles {                                        ] TF>
      private String name;
      private List<String> codes;
      private List<ChangeoverOperation> changeoverOperations;
      $LOOP$BetweenMatrix[ private HashMap<Integer, HashMap<Integer, $INSERT$[Integer]>> $[betweenMatrix]; ]
      $LOOP$[$LABEL$ ChangeoverOperationInstance[ private ChangeoverOperation $[operation] =
                                                     $COPY_SRC$[new ChangeoverOperation()]; ]]

      public Machine() {
        codes = new ArrayList<String>();
        changeoverOperations = new ArrayList<ChangeoverOperation>();
        name = "$[name]";
        $LOOP$[codes.add("$[code]"); ]
        $LOOP$[changeoverOperations.add(->$[operation]); ]
      }

      @Override
      public int getChangeoverTime(ITask prevTask, ITask nextTask) {
        Task prev = (Task) prevTask;
        Task next = (Task) nextTask;

        $LOOP$[{
                 if ($COPY_SRC$[true]) {
                   ->$[operation].coefficient += $IF$[$COPY_SRC$[1]] /
                   $ELSE$[<T  1  T>];
                 }
               }]

        int result = 0;
        for (ChangeoverOperation op : changeoverOperations) {
          result += op.getChangeoverTime();
          op.coefficient = 0;
        }
        return result;
      }

      public static class Task implements ITask {
        public int id;
        $LOOP$[$LABEL$ TaskProperty[ public $INSERT$[int] $[property]; ]]
        @Override
        public int getId() {
          return id;
        }
      }

      $COPY_SRCL$[]

      @Override
      public Class<? extends ITask[]> getTaskArrayClass() {
        return Task[].class;
      }
    }
```

Obrázek 5.4: Generator template konstruktu *Machine*.

Accesories

Jak již bylo zmíněno výše, jazyk obsahuje 3 předdefinované jednotky. A to díky tomu, že v aspektu *Accesories* jsou definovány 3 instance konstruktu *TimeUnit*. Jedna pro každou jednotku (sekunda, minuta, hodina).

5.3.2 Time

Time, jež reprezentuje dobu trvání je druhým podobně jednoduchým konstruktem z této skupiny. Jedná se jen o počet jednotek a referenci na jednotku. Například `15 min` nebo `45 s`, případně `2 h`.

Z pohledu AST se time nachází jako potomek konstruktu *ChangeoverOperation*, kde značí, jak dlouho přestavbový úkon trvá.

```
Časová jednoka: čvrthodina [ h/4 ] =  900 s
```

Obrázek 5.5: Definice nové časové jednotky.

Structure

Struktura je triviální, jedná se pouze o počet jednotek, typu integer a referenci na *TimeUnit*.

Editor

Editor je taktéž triviální, pouze dvě pole. Do jedno se zadává počet jednotek, ve druhém se volí jednotka. Příklad je vidět na obrázku 5.1 na řádcích s definicemi přestavbových úkonů vždy za znaky `=>`.

Typesystem

Pomocí typového systému je kontrolováno, že hodnota počtu jednotek je pozitivní. Nelze tedy zadávat záporné časy.

Behaviour

V aspektu Behaviour má *Time* definovanou metodu `getSeconds()`, jež vrací počet sekund trvání a jež se využívá při převodu do *Base Language*. *Time* se do *Base Language* převádí jen jako číslo a to právě jako počet sekund trvání.

5.4 Úkony přestaveb

5.4.1 ChangeoverOperation

ChangeoverOperation popisuje úkon přestavby stroje. Jedná se o jednoduchý, ale velmi důležitý konstrukt jazyka, jehož úkolem je popsat jeden úkon přestavby daného stroje, a to jen za pomoci jména, délky jeho provádění a označení, zda se jedná o násobnou operaci[13].

Úkony mohou být dvou druhů prosté a násobné. Každý z nich se při vyhodnocování přestavbového času chová odlišně. Zatímco prostý úkon je započítán pouze jednou, i pokud jej spustilo více podmínek, tak násobný úkon je započítán tolikrát, kolikrát jej všechny podmínky spustily[14].

Násobné úkony mají za úkol zachytit situaci, kdy je potřeba jeden úkon provést několikrát přičemž počet opakování úkonu je závislý na vlastnostech výrobního procesu. Například stroj má zásobník na 20 nástrojů. Výměna jednoho z nich trvá 2 minuty a počet měněných nástrojů závisí na tom, jaké nástroje vyžadovala předchozí operace a jaké nástroje vyžaduje další operace. Stejné nástroje není potřeba měnit.

[13] Prostá operace je výchozí a není označena.

[14] Jedna podmínka může násobný úkon spustit několikrát, má k tomu speciálně určený koeficient, více viz 5.6.2

```
výměna nástroje => n × 2 min Odebrat násobnost What's this?
```

Obrázek 5.6: Násobný přestavbový úkon se zobrazeným *context asistentem*.

Z pohledu AST jsou instance toho konstruktu přímými potomky instancí *Machine*.

Structure

Struktura je velmi jednoduchá *ChangeoverOperation* má pouze jméno typu string a příznak násobnosti typu boolean a posléze jednoho potomka konstruktu *Time*, který označuje délku trvání.

Editor

V editoru se vyskytují dvě, respektive tři pole. Jedno pole pro zadání jména a druhé pro zadání času. Toto pole však přebírá editor z konstruktu *Time*, takže se uživateli zobrazí pole dvě. Jedno pro počet a druhé pro časovou jednotku.

Jak si bylo možné povšimnout pole pro zadání příznaku násobnosti není v editoru přítomné a to z důvodu, že se násobnost nastavuje pomocí akcí. Násobný úkon se od prostého odlišuje zobrazenými symboly `n` × před délkou trvání. Příklad násobné operace je na obrázku 5.6, oproti tomu všechny úkony z obrázku 5.1 jsou prosté.

Akci změnit úkon z prostého na násobný lze provést dvěma způsoby, buď pomocí *context asistenta*, který je vidět na obrázku 5.6[15] jen pro prostý úkon zobrazuje tlačítko „násobná“ místo tlačítka „Odebrat násobnost“, nebo pomocí transformace, kdy je možné za šipku napsat „n *“ nebo „n x“ případně jen část a vyvolat *code completiton*.

Z násobného úkonu na prostý, ale opět využít context asistenta nebo smazat pomocí kláves backspace nebo delete pole se symboly `n` ×[16].

Typesystem

V typovém systému je implementována kontrola unikátnosti jména přestavby v rámci scope, tedy v rámci jednoho stroje.

Generator

ChangeoverOperation se do *Base language* transformuje jako volání konstruktoru pomocné třídy *ChangeoverOperation*. Konstruktoru třídy se předá jméno, počet sekund, jež úkon trvá a příznak násobnosti.

Třída *ChangeoverOperation* je implementovaná přímo v *Base Language* uvnitř MPS. Kromě konstruktoru má jednu metodu `getChangeoverTime`. Ta

[15] Bílé obdélníčky.

[16] Tato akce vyžaduje dvě stisknutí dané klávesy. První pouze obarví pole červenou barvou a čeká na potvrzení akce druhý stiskem

vrací čas v sekundách potřebný pro provedení úkonu podle hodnoty atributu koeficient, jež značí kolikrát byl tento úkon spuštěn podmínkami a podle logiky pro násobný a prostý úkon popsané výše.

5.5 Vlastnosti procesu výroby

Konstrukty ze skupiny *vlastnosti procesu výroby* se vlastnosti operací prováděných na stroji dostávají do programu v jazyce *Changeovers*.

Konstrukty v této skupině jsou dvou typů. Definiční, jež umožňují přenést do programu vlastnost výrobního procesu. Například stroj má vlastnost operace na něm prováděné, „použitý nástroj" a tato vlastnost je typu string. Referenční jež umožňují s takovou vlastností pracovat například v podmínce, můžeme chtít testovat, zda použitý nástroj v předchozí operaci odpovídá textu „vrták".

Dále je možné dělit vlastnosti podle toho, zda se vztahují jen k jedné operaci, nebo ke dvěma po sobě jdoucím operacím. K jedné operaci se vztahují kontejner *Operation*, který je ale tak nezajímavý, že se mu dále nevěnuji[17], definiční konstrukt *OperationProperty* (5.5.1), referenční konstrukty *NextOperationProperty* (5.5.2) a *PrevOperationProperty* (5.5.3), jež slouží k práci s hodnotou vlastnosti předcházející, respektive následující operace prováděné na stroji.

Hlavním posláním konstruktů z této skupiny je dostat do programu informaci o datových typech vlastností a následně umožnit typovému systému kontrolovat přípustnost prováděných operací nad těmito vlastnostmi. K tomu je využíváno datových typů implementovaných v *Base Language* a pravidel typového systému umožňující přejmout typ do vztažených konstruktů.

5.5.1 OperationProperty

Konstruktem *OperationProperty* se popisují vlastnosti, jež mají všechny operace prováděné na stroji a jež nemají závislost na pořadí ostatních operací.

Vlastnosti se popisují pomocí jména a datového typu hodnoty vlastnosti. Příkladem takového popisu je „síla plechu" datového typu double.

Z pohledu AST jsou instance *OperationProperty* přímými potomky instancí *Operation*[18], které jsou přímými potomky instancí *Machine*.

Structure

Ze strukturního pohledu má *OperationProperty* pouze jméno typu string a typ, jež je typu *Type* z *Base Language*. To umožňuje, aby vlastnost byla libovolného datového typu obsaženého v *Base Language*.

[17] Jak již bylo zmíněno výše, jedná se pozůstatek z rané fáze vývoje jazyka.
[18] Tento konstrukt je na obrázku 5.2 záměrně vynechán.

Typesystem

Typový systém obsahuje jedno pravidlo, které definuje, že *OperationProperty* přejímá typ od své proměnné typ. To znamená, že pokud například instance konstruktu má proměnou typ nastavenou na integer, můžeme k ní v jiných částech programu přistupovat jako k libovolné jiné proměnné integer.

Druhé pravidlo typového systému kontroluje unikátnost jména vlastnosti v rámci stroje.

5.5.2 NextOperationProperty

Při výpočtu sekvenčně závislých přestavbových časů je počítaný čas mezi dvěma operacemi. Jedna operace se na stroji prováděla před začátkem provádění přestavby (předchozí operace) a druhá se bude provádět po dokončení přestavby (následující operace). Obě mají stejné vlastnosti stejných datových typů, jejich hodnoty se však liší. Konstrukt *NextOperationProperty* nám umožňuje pracovat s hodnotami definovaných vlastností na následující operaci.

Konstrukt se využívá v podmínkách přestaveb například pro porovnání, zda je `síla materiálu následující operace` větší, než 5.

Z pohledu AST může být instance tohoto konstruktu zanořena kdekoliv v *Expression* podmínky přestavby nebo v *Expression* koeficientu podmínky přestavby pro násobnou operaci.

Structure

Z pohledu struktury obsahuje pouze referenci na instanci *OperationProperty*. Jedná o takzvanou *smart referenci*.

Zajímavější je však, že je potomkem *Expression*. Vzhledem k tomu, že některé *Expression* přijímají jako svoje argumenty další *Expression* lze z těchto konstruktu skládat složitější výrazy. Příkladem takové *Expression* je sčítání celých čísel, které jako argumenty přijímá dvě *Expression* typu integer. Struktura je také zachycena na obrázku 5.7.

Constraints

Constraints jsou nastaveny tak, že je možno referencovat pouze *OperationProperty*, které jsou definovány ve stejném stroji, odkud se je snažíme referencovat. Pokud tedy má stroj A *OperationProperty* A1 a A2 a stroj B *OperationProperty* B1 a B2, je možně v podmínce umístěně u stroje A využít podmínku `A1 následující operace > 2`, ale nelze u stejného stroje využít podmínku `B2 následující operace > 2`, protože B2 je mimo rozsah stroje A.

Typesystem

NextOperationProperty přebírá typ od referencované *OperationProperty*, potažmo od její proměnné typ.

```
@smart reference
  reference: property
  presentation: <empty prefix><ref. presentation> nasledující operace
concept NextOperationProperty extends    Expression
                              implements <none>

  instance can be root: false
  alias: <no alias>
  short description: Vlastnost nasledujici operace

  properties:
  << ... >>

  children:
  << ... >>

  references:
  property : OperationProperty[1]
```

Obrázek 5.7: Aspekt Structure konstruktu *NextOperationProperty*.

Generator

Do *Base Language* se transformuje jako přístup k atributu třídy *Task*. V tom případě se přistupuje k atributům instance uložené v proměnné `next`, jež reprezentují následující operace prováděné na stroji.

V *Base Language* bude tedy na místě, kde byla v jazyce *Changeovers* instance *OperationProperty*, `next.jmenoVlastnosti`.

5.5.3 PrevOperationProperty

PrevOperationProperty funguje velmi podobně jako *NextOperationProperty*, s tím rozdílem, že se místo hodnot následující operace odkazuje na hodnoty předchozí operace.

5.5.4 BetweenOpsProp

Konstrukt *BetweenOpsProp* je velmi podobný konstruktu *OperationProperty*, jen popisuje vlastnost, kterou mezi sebou má dvojice po sobě jdoucích operací. Například počet nástrojů, které je potřeba vyměnit mezi dvěma operacemi. Ten se získá, jako mohutnost množiny vzniklé rozdílem množin nástrojů potřebných pro následující operaci a nástrojů potřebných pro předchozí operaci.

Pro výpočet jednoho sekvenčně závislého přestavbového času jsou k dipozici dvě hodnoty vlastnosti operace (jedna pro předchozí operaci a druhá pro následující operaci), ale jen jedna vlastnost mezi operacemi.

Nejčastěji se bude jednat o vlastnost získanou výpočtem, který byl příliš složitý na zapsání v rámci podmínky přestavby nebo koeficientu vícenásobné přestavbové operace.

Z pohledu AST jsou instance na rozdíl od *OperationProperty* přímými potomky instancí *Machine*.

```
concepts constraints BetweenOpsPropRef {
  can be child <none>

  can be parent <none>

  can be ancestor <none>

  <<property constraints>>

  link {property}
    referent set handler <none>
    scope inherited for BetweenOpsProp
    <no presentation (deprecated)>


  default scope <no default scope>
}
```

Obrázek 5.8: Aspekt Constrains konstruktu *BetweenOpsPropRef*.

Generator

Vzhledem k tomu, že hodnoty vlastností mezi operacemi jsou počítané externě, je potřeba přepočítané tabulky s hodnotami nastavit před začátkem sestavování matice přestavbových časů. Tabulka má dva rozměry v prvním rozměru se indexuje ID předchozího operace a v druhém ID následující operace.

Do *Base Language* se tedy tento konstrukt překládá jako dvě funkce. První je setter tabulky[19] Druhá je funkce s argumenty předcházející operace a následující operace, která vrací hodnotu vlastnosti.

5.5.5 BetweenOpsPropRef

BetweenOpsPropRef je opět referenční konstrukt, který referencuje hodnotu vlastností mezi operacemi. Jeho vlastnosti jsou velmi podobné konstruktu *NextOperationProperty*.

Structure

Obsahuje referenci na instanci *BetweenOpsProp* a jde o takzvanou *smart referenci*. Stejně jako v případě *NextOperationProperty* jde o potomka *Expression*.

Constrains

Constrains zajišťují, že stejně jako v případě vlastností operace lze referencovat pouze vlastnosti mezi operacemi definované na stejném stroji, ze kterého se je snažíme referencovat. Jak je toho dosaženo ukazuje obrázek 5.8.

[19] Tabulka je reprezentována typem `HashMap<Integer, HashMap<Integer, Datový typ vlastnosti>>`.

Generator

Do *Base Language* se konstrukt transformuje jako volání druhé funkce na níž se transformuje *BetweenOpsProp*.

5.6 Podmínky přestaveb

Podmínky přestaveb jsou místo, kde se všechny předchozí definice propojí dohromady, obohatí se o logiku a vytvoří tak popis přestavbových úkonů popisovaného stroje. Pro větší přehlednost a lepší uspořádání podmínek je možné podmínky sdružovat do sekcí *ChangeoverConditionSection* (5.6.1). Pro definici samotných podmínek pak slouží konstrukt *ChangeoverCondition* (5.6).

5.6.1 ChangeoverConditionSection

Kontejner *ChangeoverConditionSection* umožňuje *ChangeoverCondition* skládat do pojmenovaných sekcí. Jedná se tedy o kontejner, který v sobě drží podmínky přestaveb, které mají například stejný spouštěč[20]. Využití či nevyužití více sekcí je na uživateli jazyka, ale jedna sekce je vždy povinná.

Jedná se pouze o vizuální dělení zajištující lepší organizaci podmínek, dále se s těmito sekcemi nijak nepracuje. Konstrukt byl do jazyka přidán na základě testování prvotní verze jazyka s uživateli, kteří o něj projevili zájem.

Z pohledu AST jsou instance *ChangeoverConditionSection* přímými potomky instancí *Machine*.

5.6.2 ChangeoverCondition

Konstrukt *ChangeoverCondition* slouží k definici podmínky, při jejímž splnění je potřeba při přestavbě provést jeden přestavbový úkon. Podle toho, zda je spouštěný úkon násobný či nikoliv rozlišuje dva druhy těchto podmínek.

Pokud podmínka spouští prostý úkon, mělo by být pro většinu úkonů možné definovat více jednodušších podmínek[21], které daný úkon spustí. Úkon bude i při spuštění více podmínkami započítán pouze jednou. Dva příklady takových podmínek je možné vidět na obrázku 5.1 pod řádkem „Druh: materiálové“.

Pokud podmínka spouští násobný úkon, nese ještě koeficient, který určuje kolikrát se má několikanásobný úkon provést. Tento druh podmínek najde nejlepší uplatnění spolu s vlastnostmi mezi operacemi, kdy právě hodnota této vlastnosti slouží jako koeficient. Příklad takové podmínky zobrazuje obrázek 5.9.

Jako podmínka může sloužit libovolný výraz složený z *Expression* definovaných jak v jazyce *Changeovers*, tak v jazyce *Base Language*, který lze logicky

[20] Např. pokud několik různých podmínek vztahujících se k vlastnostem materiálu na němž se operace provádějí zapříčiní různé přestavby stroje, nazveme ty přestavby jako „materiálové“ a dáme je do jedné takto pojmenované sekce.

[21] U podmínek, které spouští násobné úkony nedoporučuji více jak jednu podmínku na úkon, zejména pro méně pokročilé uživatele.

```
3)  počet nástrojů k výměně > 0 => počet nástrojů k výměně × výměna nástroje
  (počet nástrojů k výměně x 2 min)
```

Obrázek 5.9: Podmínka přestavby spouštějící násobný úkon.

ohodnotit. S pomocí *zkratek* by mělo být i pro začínajícího uživatele snadné vytvořit jednoduché podmínky. Oproti tomu je pro pokročilejší uživatele k dispozici velké množství konstruktů z *Base Language*.

Z pohledu AST jsou instance *ChangeoverCondition* přímými potomky instancí *ChangeoverConditionSection*.

Structure

Konstrukt se skládá z podmínky, koeficientu a reference na *ChangeoverOperation*. Podmínka i koeficient jsou oba typu *Expression*, což u podmínky umožňuje vytvářet logické výrazy popsané výše a u koeficientu provádět s hodnotami operace, nejčastěji ty matematické.

Editor

Kromě toho, že editor obsahuje pole pro zadávání všech třech potřebných vlastností, z nichž je pole pro koeficient zobrazeno pouze pro příslušný druh podmínek. Obsahuje editor ještě dvě pole pouze pro čtení.

První z nich zajišťuje očíslování podmínek v rámci sekce pro snazší orientaci a komunikaci o dané podmínce.

Druhé z nich pak zobrazuje dobu trvání úkonu, kterou spuštění podmínky vyvolá. U násobných úkonů je zobrazen i koeficient. Pro uživatele má sloužit jako připomenutí jaký čas provádění danému úkonu nastavil a umožnit mu rychle vidět co doopravdy spuštění podmínky způsobí.

Constrains

Jako spuštěný úkon je možné zvolit jen úkon definovaný na stejném stroji.

Behaviour

Expression není možné v poli pro čtení zobrazit přímo, proto má konstrukt definovanou metodu chování `getKoeficientText`, která převede koeficient na string a umožní tak jeho zobrazení v poli pro čtení.

Typesystem

V typovém systému je nastaveno, že *Expression* určený pro podmínku musí být po vyhodnocení typu boolean a *Expression* určený pro koeficient musí být po vyhodnocení typu integer. Vše zachycuje obrázek 5.10.

Jedno pravidlo typového systému ještě kontroluje, zda je koeficient nastaven, pokud je spouštěný úkon násobný. Toto pravidlo je k vidění na obrázku 5.11.

```
inference rule typeof_ChangeoverCondition {
  applicable for concept = ChangeoverCondition as changeoverCondition
  applicable always
  overrides false

  do {
    infer typeof(changeoverCondition.condition) :<=: <boolean>;
    infer typeof(changeoverCondition.koeficient) :<=: <int>;
  }
}
```

Obrázek 5.10: Aspekt Typesystem konstruktu *ChangeoverCondition* - infer pravidla.

```
checking rule check_ChangeoverCondition {
  applicable for concept = ChangeoverCondition as changeoverCondition
  overrides false

  do {
    if (changeoverCondition.changeoverOperation.mutiplicable && changeoverCondition.koeficient.isNull) {
      error "Koeficient musí být nastanen pro násobné operace" -> changeoverCondition;
    }
  }
}
```

Obrázek 5.11: Aspekt Typesystem konstruktu *ChangeoverCondition* - checking pravidla.

Generator

Transformaci do *Base Language* pro tento koncept sice zajišťuje generátor konstruktu *Machine*, ale podstatu této transformace vysvětlíme zde.

Konstrukt se transformuje na podmínku (`if`), kde se jako logický výraz použije *Expression* přiřazené do podmínky[22]. Při splnění tohoto výrazu se přičte ke koeficientu spouštěného úkonu jednička nebo hodnota daná *Expression* v koeficientu podmínky, pokud je spouštěný úkonu násobný.

5.7 Zkratky

Zkratky jsou konstrukty, které uživatelům usnadňují zápis častých podmínek. *Zkratky* jsou navrženy tak, aby podmínky v nich zapsané připomínaly přirozený jazyk. Nejen, že zápis *zkratek* je rychlejší a jednodušší, ale čtení podmínek zapsaných *zkratkami* je jednodušší.

V jazyce jsou prozatím implementovány dvě *zkratky*. *Goes* (5.7.2), která slouží k testování překročení určené hranice, kdy vlastnost předchozí operace byla na jedné straně hranice a vlastnost následující operace je na druhé straně hranice. *Changes* (5.7.3), která umožňuje testovat, jestli se hodnota vlastnosti předchozí operace liší od hodnoty vlastnosti následující operace. Obě zkratky využívají referenci na vlastnost operace *OperationPropertyRef* (5.7.1).

[22] Na *Expression* se spustí její generátor, který jí přetransformuje do *Base Language*.

5.7.1 OperationPropertyRef

Konstrukt, který umožňuje referencovat vlastnost operace a je velmi podobný ostatním referencím v jazyce *Changeovers*. Vznikl hlavně proto, aby nebylo potřeba v každé zkratce znova implementovat omezení scope, ze kterého lze vlastnosti referencovat.

5.7.2 Goes

Konstrukt *Goes* umožňuje testovat, jestli hodnota vlastnosti operace překračuje stanovenou hranici. Tedy jestli je hodnota vlastnosti předchozí operace na jedné straně stanovené hranice a hodnota vlastnosti následující operace na druhé straně. Například pokud zvolíme hranici 8 a hodnota pro předchozí operaci bude 7 a pro následující operaci 9, došlo k překročení hranice.

Konstrukt také umožňuje zvolit, kterým směrem nás překročení hranice zajímá. Na výběr jsou tři možnosti *{přes, pod, skrz}*. Pokud zvolíme možnost *přes* znamená to, že hodnota předchozí operace musí být menší, než zvolená hranice a hodnota následující operace musí být větší, než hranice, aby byla podmínka splněna. *Pod* funguje přesně opačně a *skrz* hlídá překročení hranice oběma směry.

Příklad, jak se tento konstrukt v jazyce zapíše je vidět na obrázku 5.1 v první podmínce stroje Laser.

Z pohledu AST se instance *Goes* může nacházet jako součást *Expression* podmínky v *ChangeoverCondition*.

Structure

Konstrukt se skládá z hodnoty hranice, která je datového typu integer, směru překročení hranice, který je určen výčtovým typem *GoesDirections* a z jedné instance *OperationPropertyRef*, jež určuje u které vlastnosti operace se má překročení kontrolovat.

Konstrukt je potomkem *Expression*. Lze s ním tedy zacházet stejně jako s ostatními *Expression*.

Typesystem

Typový systém nastavuje, že datový typ po vyhodnocení toho konstruktu bude boolean. Také vynucuje, že typ referencované vlastnosti musí být integer.

Generator

Při transformaci do *Base Language* se u toho konceptu využívá podmínek u spuštění redukčního pravidla, které podle hodnoty směru překročení hranice spustí jen správnou template pro transformaci. Tyto podmínky zachycuje obrázek 5.12

Do *Base language* je poté transformováno jako vhodné porovnání atributů instancí třídy `Task` uložené v proměnných `next` a `prev`.

```
concept    Goes                                  --> map_Goes_under
inheritors false
condition  (genContext, node)->boolean {
             node.direction.is(< pod >);
           }
```

```
concept    Goes                                  --> map_Goes_over
inheritors false
condition  (genContext, node)->boolean {
             node.direction.is(< pres >);
           }
```

```
concept    Goes                                  --> map_Goes_through
inheritors false
condition  (genContext, node)->boolean {
             node.direction.is(< skrz >);
           }
```

Obrázek 5.12: Podmínky spuštění redukce konstruktu *Goes* v mapování pravidel generatoru.

5.7.3 Changes

Konstrukt *Changes* umožňuje zjišťovat, jestli se hodnota vlastnosti předchozí a následující operace mění nebo zůstává stejná. Jedná se o zjednodušení zápisu `předchozí operace vlastnost != následující operace vlastnost`.

Structure

Konstrukt se skládá z jedné instance *OperationPropertyRef*, jež určuje u které vlastnosti operace se má změna kontrolovat.

Konstrukt je potomkem *Expression*, lze s ním tedy zacházet stejně jako s ostatními *Expression*.

Typesystem

Typový systém nastavuje, že datový typ po vyhodnocení toho konstruktu bude boolean.

Generator

Při transformaci do *Base Language* se u toho konceptu využívá dvou template, jedné pro primitivní datové typy a jedné pro ostatní typy. Mezi těmito template se rozhoduje pomocí podmínky u spuštění *redukčního pravidla*.

Do *Base language* je poté transformováno jako porovnání atributů instancí třídy `Task` uložené v proměnných next a prev. Primitivní datové typy se porovnávají operátorem `!=` a ostatní typy pomocí metody `.equals()`.

Kapitola 6

Generování přestavbové matice

Tato kapitola se věnuje tomu, jak sestrojit matici sekvenčně závislých časů s pomocí pravidel vytvořených v jazyce *Changeovers*.

V první části se věnuje užitečným rozhraním, které poskytuje program v jazyce *Changeovers* po převodu do jazyka Java (6.1). V druhé části potom ukazuje, že sestrojení matice (6.2) je díky dobře připraveným rozhraním a využití vhodné knihovny jednoduché. Což je velmi dobře, protože implementovaný DSL usnadňuje práci jak uživatelům jazyka, tak vývojářům, kteří se musí s jazykem spojit. Druhá část se rovněž věnuje formátům dat skrze které výsledný program komunikuje s okolím.

6.1 Rozhraní transformovaného jazyka Changeovers

Jak již bylo řečeno, v předchozích kapitolách jazyk *Changeovers* se pomocí *Generátoru* transformuje do *BaseLanguage* a ten posléze zajistí pomocí *TextGen* převod do zdrojových souborů jazyka Java.

Kód vygenerovaný z jazyka *Changeovers* poskytuje téměř vše potřebné pro generování matice přestavbových časů.

6.1.1 Třída Changeovers

Třída `Changeovers` zajišťuje získání pravidel potřebných pro sestavení matice pro konkrétní stroj. To zajišťuje její metoda `getMachineRules(String machineCode)`. Ta pro kód stroje vrací instanci implementace rozhraní `MachineRules`[1].

6.1.2 Implementace rozhraní MachineRules

Pro každý konstrukt *Machine* je vygenerována jedna třída implementující rozhraní `MachineRules`, která v sobě nese vše potřebné pro sestavení matice přestavbových časů pro konkrétní stroj.

[1] K nahlédnutí v příloze B.2.

K vypočtení přestavbového času mezi dvěma operacemi slouží metoda `getChangeoverTime(ITask prevTask, ITask nextTask)`, ta pro dvě operace vrátí čas potřebný na přestavění stroje ve vteřinách. Operace do metody vstupují v podobě instance třídy implementující rozhraní `ITask`[2].

Každá implementace rozhraní `MachineRules` má vlastní implementaci rozhraní `ITask` a je bezpodmínečně nutné, aby do metody `getChangeoverTime` vstupovaly pouze instance této implementace. Každá implementace rozhraní `ITask` má totiž atributy, které odpovídají vlastnostem operací na daném stroji.

Data operací jsou programu předávána serializovaná. Pro jejich deserializace se používá knihovna *Jackson*[9], ta pro deserializaci pole objektů potřebuje znát typ pole těchto objektů. Proto byla do implementace přidána metoda `getTaskArrayType()`, která tento typ vrací.

Původně byla implementována metoda, která pouze vracela typ objektu a ne typ pole těchto objektů. Nicméně se ukázalo, že s využitím reflexe jazyka Java není možné z typu získat typ pole tohoto typu.

6.2 Konstrukce přestavbové matice

Při využití všech vlastností, co nám nabízí jazyk *Changeovers* převedený do jazyka Java, je konstrukce matice přestavbových časů triviální. Stačí pouze pro každou dvojčlenou variaci z množiny všech operací, které je potřeba na stroji naplánovat, využít metodu `getChangeoverTime`. Čímž získáme hodnoty všech elementů matice, kromě hlavní diagonály. Diagonálu vyplníme nulami a matice je sestavena.

6.2.1 Data operací

Data operací slouží jako vstup na jehož základě se sestavení matice. Jak již bylo naznačeno výše, program dostává data v serializované formě. Konkrétně se jedná o data serializovaná ve formátu *json*. Data jsou serializovaná jako pole objektů operací. Jako klíče objetu slouží názvy vlastností operací a hodnoty jsou poté hodnoty vlastností pro danou operaci.

6.2.2 Vlastnosti programu

Program pro generování matice je realizován jako konzolová aplikace v jazyce Java, která přijímá tři vstupní argumenty. První argument je kód stroje, pro nějž se má matice vygenerovat. Druhý argument je cesta k souboru s daty operací a třetí argument je cesta k souboru, do kterého má být uložena vygenerovaná matice.

[2]K nahlédnutí v příloze B.3

6.2.3 Výstup programu

Program generuje matici ve formě dvourozměrného pole celých čísel (`integer`) a ty potom serializuje do formátu *json*. Výstupem programu je tedy soubor typu *json*, který obsahuje dvojrozměrné pole celých čísel. Každé číslo na indexu i, j odpovídá jednomu prvku matice přestavbových časů $s_{i,j,k}$, kde k odpovídá stroji pro nějž byla matice sestavena. Přestavbový čas je udán v sekundách.

6.2.4 Implemetace programu

Jak již bylo zmíněno, program je implementován jako konzolová aplikace v jazyce Java. Jeho implementaci tvoří jedna třída s názvem `MatrixGenerator`.

Průběh sestavování matice má tři fáze. Tyto tři fáze provádí metoda `main` zachycená v zjednodušené podobě ve zdrojovém kódu 6.1. V první fází je získání předpisu přestavbových časů pro zadaný stroj (řádky 6 a 7). Ve druhé fázi jsou s využitím metody `loadTasks` načteny a deserializovány data operací (řádek 8). Ve třetí fázi je pomocí metody `generateMatrix` generována a průběžně ukládána matice přestavbových časů (řádek 9).

```
1  public static void main(String[] args) throws IOException {
2      if (args.length < 3) {
3          printHelp();
4          System.exit(1);
5      }
6      Changeovers changeovers = new Changeovers();
7      MachineRules rules = changeovers.getMachineRules(args
           [0]);
8      ITask[] tasks = loadTasks(args[1], rules);
9      generateMatrix(rules, tasks, args[2]);
10 }
```

Zdrojový kód 6.1: Metoda `main` třídy `MatrixGenerator`.

```
1  private static ITask[] loadTasks(String inputFilePath,
       MachineRules rules){
2      ObjectMapper mapper = new ObjectMapper();
3      mapper.configure(DeserializationFeature.
           FAIL_ON_UNKNOWN_PROPERTIES, false);
4      return mapper.readValue(new File(inputFilePath), rules.
           getTaskArrayClass());
5  }
```

Zdrojový kód 6.2: Metoda `loadTasks` třídy `MatrixGenerator`.

Metodu `loadTasks` znázorňuje zdrojový kód 6.2. Ze zdrojového kódu je patrné, že díky využití knihovny *Jackson* a metodě `getTaskArrayClass` třídy implementující rozhraní `MachineRules` je načtení a deserilizace triviální. Zajímavý je řádek 3, který zajistí, že jsou data bez chyby serializováná i pokud deserializovaný objekt obsahuje nějakou vlastnost navíc, oproti očekávaným.

Metoda `loadTasks` jejíž implementace je vidět ve zdrojovém kódu 6.3, zajišťuje samotné generování a ukládání prvků matice. Matice je generována po řádcích. Každý řádek je ihned serializován a zapsán do výstupního souboru, což velmi snižuje paměťovou náročnost programu. Lze si všimnout, že pro elementy, jež nejsou na hlavní diagonále matice je pro získání přestavbového času využita metoda `getChangeoversTime` třídy implementující rozhraní `MachineRules`. Prvkům na hlavní diagonále matice je rovnou přiřazena nula.

```
1  private static void generateMatrix(MachineRules rules,
       ITask[] tasks, String outputFileName){
2      FileOutputStream out = new FileOutputStream(
           outputFileName);
3      JsonFactory jfactory = new JsonFactory();
4      JsonGenerator jGenerator = jfactory.createGenerator(new
            BufferedOutputStream(out), JsonEncoding.UTF8);
5      jGenerator.writeStartArray();
6      int[] matrixLine = new int[tasks.length];
7      for (int i = 0; i < tasks.length; i++) {
8          ITask prevTask = tasks[i];
9          for (int j = 0; j < tasks.length; j++) {
10             ITask nextTask = tasks[j];
11             if (i == j) {
12                 matrixLine[j] = 0;
13             } else {
14                 matrixLine[j] = rules.getChangeoverTime(
                       prevTask, nextTask);
15             }
16         }
17         jGenerator.writeArray(matrixLine, 0, matrixLine.
               length);
18     }
19     jGenerator.writeEndArray();
20     jGenerator.close();
21 }
```

Zdrojový kód 6.3: Metoda `generateMatrix` třídy `MatrixGenerator`.

Kapitola 7

Uživatelské testovaní

Tato kapitola popisuje, jak byl jazyk *Changeovers* testován. Vzhledem k tomu, že hlavním cílem bylo zapojení doménových odborníků do vývoje se testování zaměřilo na uživatelskou část jazyka.

V úvodu kapitoly je popsán průběh testování (7.1), dále se zabývá tím, jak testování probíhalo s prvním odborníkem (7.2) a druhým odborníkem (7.3). Vhledem k tomu, že testování probíhalo podle očekávání a uživatelům se dařilo vytvářet smysluplné programy bez chyb, se zde zaměřujeme zejména na objevené nedostatky a jejich řešení. V závěru kapitola hodnotí výsledky a navrhuje další možnosti, jak zploštit učící křivku jazyka (7.4).

Při testování jazyka každý odborník vytvořil jeden program v jazyce *Changeovers*, tyto programy jsou uvedeny v příloze C.

7.1 Průběh testování

Uživatelské testování bylo zaměřeno na programování v editoru jazyka *Changeovers*, což je jediná část jazyka se kterou přijdou uživatelé do styku. Jazyk byl testován dvěma doménovými odborníky. Oba testy proběhly podle stejného scénáře.

7.1.1 Standalone IDE

Pro potřeby testování byla vytvořena distribuce jazyka *Changeovers* typu standalone IDE. Jedná se o speciální přenositelnou distribuci MPS s přibaleným jazykem *Changeovers*. V tomto IDE byl připraven jednoduchý projekt obsahují jeden prázdný program[1].

7.1.2 Seznámení s Editorem

Na začátku testování si uživatel otevřel IDE a předpřipravený projekt. Potom byl seznámen s tím, jak editor funguje.

[1] Jednu instanci konstruktu *Changeovers*.

Seznamování se zaměřovalo zejména na:

- Využívání dialogu code completion (ctrl+mezerník).
- Navigaci mezi poli, jež je potřeba vyplnit (tabulátor, shift+tabulátor).
- Funkce klávesy enter.
  - Potvrzuje většinu výběrů.
  - Na konci sekce, která může obsahovat více konstruktů vytvoří novou instanci konstruktu.
- Označení více prvků.
  - Nelze využít tažení myší.
  - Nutno provádět kombinací kláves ctrl a kurzorových šipek.

7.1.3 Princip vytváření programů

Po seznámení s editorem byl uživateli vysvětlen princip vytváření programů v jazyce *Changeovers*.

Bylo vysvětleno jak se vytvářejí vlastnosti, k čemu v programu slouží a také bylo vysvětleno, jaké datové typy je doporučeno u vlastností využívat[2].

Dále bylo vysvětleno, jak vytvářet a k čemu slouží přestavbové úkony. Posléze přišly na řadu sekce podmínek, u kterých bylo naznačeno, že jejich využití je dobrovolné.

Nakonec byl vysvětlen princip vytváření podmínek přestaveb. U tohoto konstruktu byl důraz kladen na využívání zkratek a maximální jednoduchost podmínek. Bylo zdůrazněno, že spuštění jednoho úkonu více podmínkami najednou je zamýšlené chování, které není na škodu.

Uživatel byl rovněž upozorněn, že cokoliv je v editoru vyznačeno červeně, ať už se jedná o barvu písma nebo podtržení textu značí chybu.

Záměrně nebyl popisován konstrukt vlastnosti mezi operacemi, protože pro popis strojů ze zadání není potřeba a je určen pouze velmi zkušeným uživatelům jazyka nebo vývojářům pro doplnění popisů.

7.1.4 Způsob vytváření programů

Každému uživateli bylo naznačeno, že při vytváření programů v jazyce *Changeovers* může postupovat alespoň dvěma způsoby. Buď může nejdříve definovat všechny vlastnosti operací prováděných na popisovaném stroji, pak všechny přestavbové úkony a posléze všechny podmínky přestaveb. Nebo může postupovat po jednotlivých přestavbových úkonech. Pro každý úkon nejprve definuje přestavbový úkon, pak vlastnosti operací, které s ním souvisí a na konec podmínky, které úkon spouští a pokračuje zase od začátku dokud existují další nepopsané úkony.

[2]Je doporučeno vyžívat primitivní datové typy. Typ `integer` pro celá čísla, typ `double` pro desetinná čísla a typ `boolean` pro logické hodnoty. Pro řetězce poté neprimitivní datový typ `string`.

7.1.5 Úkol pro uživatele

Po seznámení s editorem a jazykem byl uživatel požádán, aby vytvořil program popisující přestavby na třech typech strojů. Jednalo se stroje ze zadání práce, tedy laser, ohraňovací stroj a frézku. Pořadí popisovaných strojů bylo ponecháno na uživateli.

7.1.6 Sledování průběhu

U testování byl jako pozorovatel přítomen autor této práce. Sledoval jak uživatelé s editorem pracují a odpovídal na případné dotazy uživatelů.

7.1.7 Hodnocení jazyka uživateli

Po vytvoření programu byly uživatelé dotázání, jak se jim s editorem pracovalo. Co by na jazyce nebo editoru změnili. Snažil jsem se také zjistit, jestli by si dokázali představit, že by s pomocí tohoto nástroje popsali přestavby na všech typech strojů v jejich podniku. Uživatel byl rovněž požádán, aby se vrátil k prvnímu popsanénu stroji, přečetl si jeho popis a vyjádřil se ke srozumitelnosti zápisu.

7.2 Odborník A

Odborník A křestním jménem Jan ve věku 45 let pracuje jako technolog v podniku zabývajícím se kovovýrobou. S Janem bylo spolupracováno i při testování rané fáze jazyka. Od něj pochází náměty na implementaci sekcí podmínek a doplnění vizuálního oddělení jednotlivých strojů pomocí horizontální šedé čáry.

7.2.1 Průběh

Odborník si při vytváření programu vedl poměrně dobře. Největší problém mu dělalo používání enteru k vytvoření nové instance konstruktů a občas zapomínal potvrdit zadáváné údaje pomocí enteru. Někdy naopak vytvořil víc nových instancí, než původně zamýšlel.

Jako téměř neproveditelné se ukázalo vytváření nové sekce podmínek, odborník si však poradil. Zkopíroval již existující sekci a vložil ji pod ni.

K vytváření programu zvolil poněkud hybridní způsob. Postupoval druhým způsobem, jen místo jednoho vytvářel více přestavbových úkonů najednou.

Včetně úvodního školení a následného hodnocení jazyka zvládl odborník vytvořit požadovaný program za 90 minut.

7.2.2 Hodnocení jazyka

Odborník označil práci s editorem za zpočátku nezvyklou, avšak pochopitelnou. Byl přesvědčen o tom, že při popisu čtvrtého stroje by již s editorem pracoval poměrně svižně.

Odborník by ocenil, kdyby editor smazal prázdnou instanci konstruktu, pokud na ní není v editoru focus. Také by ocenil kontrolu, za jsou všechny definované konstrukty spouštěné alespoň jednou podmínkou.

Na dotaz, zda by si pomocí podobného nástroje dokázal představit popsat všechny stroje z podniku kde pracuje, odpověděl kladně s výhradou, že není odborníkem na všechny stroje, ale ty kterým dobře rozumí by takto klidně popsal.

Zápis prvního stroje přišel odborníkovi přehledný a srozumitelný.

7.2.3 Zjištěné nedostatky

Nelze přidat sekci podmínek pod poslední sekci podmínek, ale pouze nad ní. Tento nedostatek je v opravené verzi jazyka napraven přidáním prázdného řádku za poslední sekci podmínek.

Kontrolovat spuštění přestavbových úkonů. Kontrola byla do opravené verze jazyka doplněna. Nevyužité přestavbové úkony jsou nyní označeny oranžovým podtržením[3].

7.2.4 Zhodnocení

Odborník A vytvářel komplexnější popisy přestaveb strojů, než odborník B. Program vzniklý během testování je k dipozici v příloze C.1. Vzniklé podmínky byly s odborníkem po skončení testování ještě jednou zkontrolovány tím způsobem, že je pozorovatel interpretoval v přirozeném jazyce a odborník potvrdil jejich správnost. Všechny podmínky této kontrole vyhověly.

Bylo znát, že doménový odborník nepracoval se systém úplně poprvé, ale měl již zkušenosti z předchozích verzí.

7.3 Odborník B

Odborník B křestním jménem Tomáš ve věku 51 let pracuje jako normovač v podniku zabývajícím se strojní výrobou.

7.3.1 Průběh

Odborníkovi dělala z počátku práce s editorem problémy, nedařilo se mu zapamatovat si jak s editorem pracovat a musel se dotazovat pozorovatele. Největší problémy mu dělalo užívání zkratek. Byl zmaten z toho, že zkratky nejsou k dispozici na jedné z prvních pozic *code completion menu* jako je tomu u předchozí a následující operace. Tyto problémy však postupně překonal a u popisu druhého a třetího stroje již pracoval plynule.

K vytváření programu zvolil odborník první ze zmiňovaných způsobů.

Doba celého testování byla přibližně 120 minut, ta zahrnuje úvodní školení, vytváření programu a zhodnocení jazyka.

[3] Oranžové podtržení značí varování.

7.3.2 Hodnocení jazyka

Odborník označil práci s editorem za zpočátku velmi matoucí, ale poté, co si na ní člověk zvykne za zvládnutelnou.

Jak již bylo zmíněno výše, odborník by ocenil, kdyby se zkratky zadávaly stejně jako vlastnosti předchozí a následující operace.

Odborník si dovede představit, že by s pomocí tohoto nástroje popsal přestavby na všech typech strojů v jeho podniku.

Zápis prvního stroje přišel odborníkovi pochopitelný.

7.3.3 Zjištěné nedostatky

Zkratky fungují odlišně, než vlastnosti předchozí a následující operace. Zatímco pro vlastnosti operace je potřeba začít zadávat jméno vlastnosti, pro zkratky je potřeba začít zadávat jméno zkratky a posléze vybrat, ke které vlastnosti se vztahuje. Jedná se o vlastnost, jejíž oprava není triviální a původně byla takto zamýšlena. V opravené verzi jazyka toto chování zůstává.

7.3.4 Zhodnocení

Odborník B vytvořil jednodušší popisy strojů, než odborník A a trvalo mu to delší dobu. Program vzniklý během testování je k nahlédnutí v příloze C.2. Podmínky byly kontrolovány podobně jako v případě odborníka A a všechny kontrolou prošly.

7.4 Vyhodnocení testování

Testování dopadlo až nad očekávání dobře. Objevené nedostatky byly spíše drobného rázu a většina z nich byla opravena v opravné verzi jazyka.

Oba odborníci dokázaly s editorem pracovat a vytvořit smysluplné a bezchybné programy. Rovněž doba potřebná k popsání strojů je přijatelná a odpovídá očekávání. Navíc se dá předpokládat, že popisy dalších strojů by odborníci vytvářeli rychleji.

Za úspěch také považuji, že oba odborníci přijali nástroj kladně a vyjádřili ochotu v nástroji popsat i ostatní stroje v jejich podnicích.

Kromě drobných chyb v jazyce byl nalezen ještě jeden problém. I přes důkladné školení byla zpočátku práce s nástrojem problematická, proto bych doporučil vytvářet několik prvních popisů strojů metodou párového programování, kde druhý z týmu je znalý nástroje. Případně dodat alespoň několik vzorových popisů strojů, ze kterých by mohli uživatelé jazyka při popisování strojů vycházet.

Kapitola 8

Závěr

Pro sestavení co možná nejlepšího výrobního plánu je potřeba znát velké množství informací o výrobním procesu. Jednou z těchto informací jsou i sekvenčně závislé přestavbové časy strojů. Získání časů je neprávem opomíjený nelehký úkol.

Stávající přístupy k řešení problému mají mnoho nedostatků. Jejich řešení je komplikované a velmi pracné, případně naopak neposkytuje veškerou potřebnou funkcionalitu. Tato práce se věnuje vytvoření systému, který podobnými nedostatky netrpí. Vzniklý systém umožňuje pomocí co možná nejsnazších pravidel získat sekvenčně závislé přestavbové časy z dat výrobního procesu, které již podniky mají.

8.1 Dosažené výsledky

Podařilo se úspěšně dosáhnout všech cílů vytyčených v úvodu práce a rovněž se podařilo naplnit všechny body zadání této práce. V práci jsou popsány základy optimalizace výrobního plánu a vlivu přestavbových časů na tento proces. Rovněž bylo analyzováno, jak přestavbové časy vznikají, jaké mají složky a jaké je jejich spojení s problémovou doménou. Proběhla rešerše současných přístupů k získávání přestavbových časů a byly zjištěny jejich nedostatky. Dále bylo analyzováno, jak všechny předchozí poznatky společně s poznatky z oblasti doménově specifických jazyků přetransformovat v systém umožňující doménovým odborníkům vytvářet pravidla pro výpočet přestavbových časů z dat o výrobním procesu.

Vzniklý systém byl realizován jako DSL typu projekční editor. Pro realizaci byl využit nástroj MPS, v němž vznikl nový jazyk *Changeovers*. Programy napsané v tomto jazyce se transformují do jazyka Java, čímž vznikne soubor pravidel pro výpočet přestavbových časů. Dále v rámci této práce vznikl generátor matice přestavbových časů, jež tato pravidla aplikuje na data výrobního procesu a zkonstruuje matici pro zadaný stroj.

Programování v jazyce *Changeovers* bylo testováno s doménovými odborníky, jež jsou uživatelé toho jazyka. Testování dopadlo nad očekávání dobře. Během testování byly nalezeny pouze drobné nedostatky, z nichž většina byla ihned snadno opravena. Testovaný jazyk byl uživateli dobře přijímán. Programy vytvořené během testování byly smysluplné a bez chyb. Doménovým

odborníkům se je podařilo vytvořit rychle. Tím byla potvrzena smysluplnost celého vytvořené systému.

8.2 Přínos vzniklého systému

Systém realizovaný v rámci práce umožňuje získávat sekvenčně závislé přestavbové časy i tam, kde to dříve nebylo možné. Jeho využití je však možné i na místech, kde se používají jiné přístupy k získávání přestavbových časů. Zde může celý proces zjednodušit a zpřehlednit. Zejména vyšší přehlednost a jednodušší úprava parametrů může být velkou výhodou při ladění plánování, což je velmi komplikovaný a zdlouhavý proces.

Díky tomu, že v systému definují pravidla přímo doménoví odborníci, kteří pravidla rovněž znají, odpadá možnost zkreslení pravidel díky nedorozumění mezi vývojářem a doménovým odborníkem, který pravidla zná. I v případě, kdy potřebuje doménový odborník pomoc vývojáře při definici složitějších pravidel, je možnost nedorozumění snížena tím, že jazyk slouží jako slovník pojmů, čímž zlepšuje komunikaci o problému.

8.3 Možné navazující práce

Jednou z možností, jak navázat na tuto práci, je věnovat se přenosu jazyka *Changeovers* do jiných domén. To by vyžadovalo seznámení se s novými problémovými doménami a implementaci dalších *zkratek*, které by zjednodušily podmínky přestaveb v těchto doménách. Jednoduché podmínky jsou totiž klíčem k úspěchu.

Druhou z možností je věnovat se dalšímu zlepšení uživatelského zážitku z editoru jazyka a to zejména implementací dalších akcí. Společně s tímto bylo vhodné vylepšit zážitek i ze standalone IDE. Zaměřit se lze na individualizaci vzhledu pro odlišení od samotného MPS a zjednodušení vytváření nových projektů.

Třetí možností je věnovat se vývoji nového příbuzného jazyka nebo nových prvků jazyka *Changeovers*, které by umožnili mapování dat z ERP systémů na konstrukty jazyka *Changeovers*. To by umožnilo ještě snazší integraci systému vzniklého v této práci.

8.4 Hodnocení použitého nástroje

I přes velkou vstupní bariéru a místy horší dokumentaci byl nástroj Jetbrains MPS dobrou volbou. Investice do seznámení se s nástrojem se určitě vyplatí navíc začátečnický tutoriál je velmi kvalitně zpracován, takže se zde žádné problémy nevyskytují. Informace, které v dokumentaci chybí dokáže suplovat komunita utvořená kolem tohoto nástroje, která na dotazy odpovídá. Pro vytváření DSL, které jsou vhodné pro využití doménovými odborníky se dle mého názoru jedná o nejlepší v současnosti dostupný nástroj. Velice si cením zkušeností získaných při práci s tímto nástrojem.

Literatura

[1] Intentional Software [online]. 2017. Dostupné z: https://www.intentsoft.com/.

[2] Langserver.org [online]. 2019. Dostupné z: https://langserver.org/.

[3] Metaprogramming [online]. 2019. Dostupné z: https://en.wikipedia.org/wiki/Metaprogramming.

[4] Managing large setup matrices in a complex manufacturing environment [online]. 2016. Dostupné z: http://schedulingzoo.lip6.fr/.

[5] The Spoofax Language Workbench [online]. 2019. Dostupné z: http://www.metaborg.org.

[6] Whole Platform [online]. 2017. Dostupné z: https://whole.sourceforge.io/.

[7] Xtext [online]. 2019. Dostupné z: https://www.eclipse.org/Xtext.

[8] Google for Education Blockly [online]. 2019. Dostupné z: https://developers.google.com/blockly/.

[9] FasterXML/jackson [online]. 2019. Dostupné z: https://github.com/FasterXML/jackson.

[10] MetaCase [online]. 2019. Dostupné z: https://www.metacase.com/.

[11] Scratch - Imagine, Program, Share [online]. 2019. Dostupné z: https://scratch.mit.edu/.

[12] BARASH, M. Implementing DSLs in practice [online]. 2019. Dostupné z: http://dsl-course.org/implementing-dsls-in-practice/.

[13] BRUCKER, P. Scheduling algorithms. Berlin New York : Springer, 2007. ISBN 9783540695165.

[14] DAL, V. Minimizing machine changeover time in product line in an apparel industry. Tekstil ve Konfeksiyon. 04 2013, 23, s. 159.

[15] ERDWEG, S. a kol. The State Of The Art In Language Workbenches. Conclusions From The Language Workbench Challenge. 8225, 10 2013. doi: 10.1007/978-3-319-02654-1_11.

[16] FOWLER, M. Domain Specific Language [online]. ThoughtWorks, 2008. Dostupné z: https://martinfowler.com/bliki/DomainSpecificLanguage.html.

[17] FOWLER, M. ProjectionalEditing [online]. 2008. Dostupné z: https://martinfowler.com/bliki/ProjectionalEditing.html.

[18] FOWLER, M. Language workbenches: The killer-app for domain specific languages. 2005.

[19] HANZÁLEK, Z. Scheduling [online]. 2018. Dostupné z: https://rtime.felk.cvut.cz/~hanzalek/KO/sched_e.pdf.

[20] HINSEN, K. Domain-Specific Languages in Scientific Computing. Computing in Science & Engineering. 2018, 20, 1, s. 88–92.

[21] LÄMMEL, R. A Story of a Domain-Specific Language, s. 51–86. Springer International Publishing, Cham, 2018. doi: 10.1007/978-3-319-90800-7_2. Dostupné z: https://doi.org/10.1007/978-3-319-90800-7_2. ISBN 978-3-319-90800-7.

[22] LORENZ, D. H. – ROSENAN, B. Cedalion: A Language for Language Oriented Programming. SIGPLAN Not. October 2011, 46, 10, s. 733–752. ISSN 0362-1340. doi: 10.1145/2076021.2048123. Dostupné z: http://doi.acm.org/10.1145/2076021.2048123.

[23] MAKARKIN, A. Base Language [online]. Jetbrains, 2019. Dostupné z: https://confluence.jetbrains.com/display/MPSD20183/Base+Language.

[24] MAKARKIN, A. – PECH, V. ProjectionalEditing [online]. 2018. Dostupné z: https://confluence.jetbrains.com/display/MPSD20183/Basic+notions.

[25] MATHIAS GUTTORMSEN, S. – PRINZ, A. – GJøSæTER, T. Consistent Projectional Text Editors. 02 2017. doi: 10.5220/0006264505150522.

[26] MERNIK, M. – HEERING, J. – M. SLOANE, A. When and How to Develop Domain-Specific Languages. ACM Comput. Surv. 12 2005, 37, s. 316–. doi: 10.1145/1118890.1118892.

[27] Pasternak, E. – Fenichel, R. – Marshall, A. N. Tips for creating a block language with blockly. In 2017 IEEE Blocks and Beyond Workshop (B B), s. 21–24, Oct 2017. doi: 10.1109/BLOCKS.2017.8120404.

[28] PINEDO, M. L. Scheduling. Springer, 2012. doi: 10.1007/978-1-4614-2361-4. Dostupné z: http://dx.doi.org/10.1007/978-1-4614-2361-4.

[29] SANAULLA, M. Creating Internal DSLs in Java & Java 8 [online]. 2013. Dostupné z: https://dzone.com/articles/creating-internal-dsls-java.

[30] SATYAMURTHY, S. Managing large setup matrices in a complex manufacturing environment [online]. 2017. Dostupné z: http://blog.olivehorse.com/sap-apo-managing-large-setup-matrices-in-a-complex-manufacturing-environment.

[31] STREMBECK, M. – ZDUN, U. An approach for the systematic development of domain-specific languages. Software: Practice and Experience. 2009, 39, 15, s. 1253–1292. doi: 10.1002/spe.936. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.936.

[32] TOMASSETTI, F. The complete guide to (external) Domain Specific Languages [online]. 2016. Dostupné z: https://tomassetti.me/domain-specific-languages/.

[33] VOELTER, M. DSL Engineering. Stuttgart : CreateSpace Independent Publishing Platform, 1 edition, 2013. ISBN 978-1481218580.

[34] VOELTER, M. What makes a good Business DSL [online]. 2018. Dostupné z: https://languageengineering.io/what-makes-a-good-business-dsl-82b8e15cff99.

[35] VOELTER, M. a kol. Using language workbenches and domain-specific languages for safety-critical software development. Software & Systems Modeling. May 2018. ISSN 1619-1374. doi: 10.1007/s10270-018-0679-0. Dostupné z: https://doi.org/10.1007/s10270-018-0679-0.

Příloha A

Seznam použitých zkratek

AST Abstraktní syntaktický strom

DSL z anglického Domain-specific language česky Doménově specifický jazyk

ERP z anglického Enterprise Resource Planning

ID identifikátor

IDE z anglického Integrated Development Environment česky Vývojové prostředí

MPS Jetbrains Meta Programing systém - language workbech založený na principu projekčního editoru

Příloha B

Pomocné třídy a rozhraní implementované přímo v Base Language

Vzhledem k tomu, že se jedná o projekce AST přikládám implementace jako obrázky.

B.1 Třída ChangeoverOperation

```
public class ChangeoverOperation {

  private string name;
  private boolean multiplicable;
  public int coefficient;
  private int seconds;

  public ChangeoverOperation(string name, int seconds, boolean multiplicable) {
    this.name = name;
    this.seconds = seconds;
    this.multiplicable = multiplicable;
    this.coefficient = 0;
  }

  public int getChangeoverTime() {
    if (coefficient > 0) {
      if (multiplicable) { return coefficient * seconds; }
      return seconds;
    }
    return 0;
  }

  public string getName() {
    return name;
  }
}
```

Obrázek B.1: Implementace třídy `Changeovers`.

B.2 Rozhraní MachineRules

```
public interface MachineRules {
  int getChangeoverTime(ITask prevTask, ITask nextTask);
  Class<? extends ITask[]> getTaskArrayClass();
}
```

Obrázek B.2: Implementace rozhraní `MachineRules`.

B.3 Rozhraní ITask

```
public interface ITask {
  int getId();
}
```

Obrázek B.3: Implementace rozhraní `ITask`.

Příloha C

Programy vniklé při testování

Projekce programů vzniklých při testování. Vhledem k velké délce uvádím popis každého stroje samostatně.

C.1 Odborník A

Popisy přestaveb strojů vzniklých během testování jazyka odborníkem A.

```
Stroj: Frézka čelní malá

  Kód: 003, 004, 005

  Vlastnosti operace:
    Druh upnutí : string, Nástroj : string, Přípravek : string, Druh materiálu : string

  Vlastnosti mezi operacemi:


  Přestavby:
    Změna upnutí => 5 min
    Výměna nástroje => 5 min
    Změna přípravku => 5 min
    Čištění stroje => 15 min

  Podmínky přestaveb:
    Druh: Materiálové závislosti
      1)  mění se Druh materiálu => Čištění stroje
        (15 min)
    Druh: Pomůcky
      1)  mění se Přípravek => Změna přípravku
        (5 min)
      2)  mění se Nástroj => Výměna nástroje
        (5 min)
      3)  mění se Druh upnutí => Změna upnutí
        (5 min)
```

Obrázek C.1: Popis pravidel přestaveb frézky od odborníka A.

```
Stroj: Laser
  Kód: 001

  Vlastnosti operace:
    Tloušťka plechu : Integer, Druh materiálu : string, Délka : int, Váha : int

  Vlastnosti mezi operacemi:


  Přestavby:
    Čištění stroje => 15 min
    Výměna podkladu => 30 min
    Výměna hlavy => 10 min
    Manipulace s podpěrami => 10 min
    Pomoc druhé osoby => 10 min
    Přesun jeřábem => 8 min

  Podmínky přestaveb:
    Druh: Materiálové závislosti
      1)  mění se Druh materiálu => Čištění stroje
        (15 min)

    Druh: Rozměrové závislosti
      1)  Tloušťka plechu jde skrz hranici 10 => Výměna podkladu
        (30 min)

      2)  Tloušťka plechu jde skrz hranici 10 => Výměna hlavy
        (10 min)

      3)  Délka jde skrz hranici 4000 => Manipulace s podpěrami
        (10 min)

      4)  následující operace Délka > 4000 => Pomoc druhé osoby
        (10 min)

      5)  následující operace Váha > 25 => Přesun jeřábem
        (8 min)
```

Obrázek C.2: Popis pravidel přestaveb laseru od odborníka A.

```
Stroj: Ohraňovací lis
  Kód: 002

  Vlastnosti operace:
    Horní nástroj : string, Dolní nástroj : string, Přípravek : string, Váha : int, Velký rozměr : int

  Vlastnosti mezi operacemi:


  Přestavby:
    Výměna - horní nástroj => 10 min
    Výměna - dolní nástroj => 15 min
    Změna přípravku => 10 min
    Pomoc druhé osoby => 10 min
    Seřízení podpěry => 5 min

  Podmínky přestaveb:
    Druh: Rozměrové závislosti
      1)  Velký rozměr jde pres hranici 2500 => Pomoc druhé osoby
        (10 min)

      2)  následující operace Velký rozměr > 2000 => Seřízení podpěry
        (5 min)

      3)  následující operace Váha > 25 => Pomoc druhé osoby
        (10 min)

    Druh: Pomůcky
      1)  mění se Přípravek => Změna přípravku
        (10 min)

    Druh: Nástroje
      1)  mění se Horní nástroj => Výměna - horní nástroj
        (10 min)

      2)  mění se Dolní nástroj => Výměna - dolní nástroj
        (15 min)
```

Obrázek C.3: Popis pravidel přestaveb ohraňovacího stroje od odborníka A.

C.2 Odborník B

Popisy přestaveb strojů vzniklých během testování jazyka odborníkem B.

```
Stroj: Frézka
  Kód: f03

  Vlastnosti operace:
    Nástroj : string, Materiál : string, Přípravek : boolean

  Vlastnosti mezi operacemi:


  Přestavby:
    Výměna nástroje => 10 min
    Manipulace s přípravkem => 5 min
    Čištění => 10 min

  Podmínky přestaveb:
    Druh: Všechny
      1)  mění se Nástroj => Výměna nástroje
        (10 min)

      2)  mění se Materiál => Čištění
        (10 min)

      3)  následující operace Přípravek => Manipulace s přípravkem
        (5 min)

      4)  předchozí operace Přípravek => Manipulace s přípravkem
        (5 min)
```

Obrázek C.4: Popis pravidel přestaveb frézky od odborníka B.

```
Stroj: Laser
  Kód: l001

  Vlastnosti operace:
    Tloušťka : int, Délka : int, Materiál : string

  Vlastnosti mezi operacemi:


  Přestavby:
    Výměna hlavy => 8 min
    Čištění => 15 min
    Shanění pomoci => 5 min

  Podmínky přestaveb:
    Druh: Všechny
      1)  Tloušťka jde pres hranici 12 => Výměna hlavy
        (8 min)

      2)  Tloušťka jde pod hranici 10 => Výměna hlavy
        (8 min)

      3)  následující operace Délka > 2800 => Shanění pomoci
        (5 min)

      4)  mění se Materiál => Čištění
        (15 min)
```

Obrázek C.5: Popis pravidel přestaveb laseru od odborníka B.

```
Stroj: Ohraňovák

  Kód: o002

  Vlastnosti operace:
    Horní nástroj : string, Dolní nástroj : string, Přípravek : boolean

  Vlastnosti mezi operacemi:


  Přestavby:
    Výměna horního nástroje => 8 min
    Výměna dolního nástroj => 12 min
    Manipulace s přípravkem => 5 min

  Podmínky přestaveb:
    Druh: Všechny
      1)  mění se Horní nástroj => Výměna horního nástroje
        (8 min)

      2)  mění se Dolní nástroj => Výměna dolního nástroj
        (12 min)

      3)  následující operace Přípravek => Manipulace s přípravkem
        (5 min)

      4)  předchozí operace Přípravek => Manipulace s přípravkem
        (5 min)
```

Obrázek C.6: Popis pravidel přestaveb ohraňovacího stroje od odborníka B.

Příloha D

Obsah přiloženého CD

```
CD
├── changeovers ........................ projekt jazyka Changeovers
├── matrixGenerator ........... projekt generátoru přestavbové matice
├── tests ............................... soubory spojené s testováním
│   ├── ExpertA ...... program vytvořený během testování odborníkem A
│   ├── ExpertB ...... program vytvořený během testování odborníkem B
│   └── StandaloneIDE ............. standalone IDE použité k testování
└── text ............................................ text práce
    ├── pdf ........................................ výsledný soubor
    └── source ............................... zdrojové soubory textu
```