VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů

Ing. Peter Jurnečka

Návrhové vzory v paralelných a distribuovaných systémoch

Tézy k dizertačnej práci

Školitel: doc. Dr. Ing. Petr Hanáček

Kľúčové slová

návrhové vzory, paralelné programovanie, refaktoring, statická analýza, BPSL, spôsob zápisu návrhových vzorov

Keywords

design patterns, parallel programming, refactoring, static analysis, BPSL, notation of design patterns

Obsah

1 Úvod

Programovanie paralelných alebo viacvláknových aplikácií sa čím ďalej tým viac rozširuje. Nové technológie, ako sú viacjadrové procesory alebo masívne paralelné procesory grafických kariet sa stali široko dostupnými a použiteľnými aj v bežných počítačoch. Programovanie paralelných systémov však kladie vyššie nároky na znalosti programátorov a tieto vyššie nároky sa ešte násobia pri údržbe a úpravách existujúcich projektov.

Medzi oblasti, v ktorých môže mať každá chyba fatálne následky, patrí letectvo alebo medicína. Bezpečnostné štandardy [15][4] majú v letectve dôležitú úlohu, pretože aj malé zlyhania, môžu mať fatálne následky. Keď hovoríme o softvéri v oblasti letectva, máme hlavne na mysli softvér pre avioniku. Jedná sa o termín používaný pre elektronické systémy používané v prostredí letectva, názov je odvodený od slov letectvo a elektronika. Príklady elektronických systémov používaných v letectve sú systémy riadenia letu (autopilot), navigačné systémy alebo antikolízne systémy. Bezpečnosť softvéru je neoddeliteľnou súčasťou bezpečnosti celého systému.

V medicíne je bezpečnosť zabezpečená pomocou Food and Drug Administration (FDA) validačných štandardov [17] [5], ktorých účelom je posúdenie a validácia softvéru v lekárskych zariadeniach. Normy odporúčajú integráciu správy životného cyklu a riadenia rizík počas vývoja. Vývojár konkrétneho softvéru by si mal stanoviť špecifický prístup a úroveň úsilia, ktoré sa použijú na základe týchto noriem. Na druhú stranu, FDA validačné štandardy neodporúčajú nejaké konkrétne modely životného cyklu a ani špecifické techniky.

Zabránenie chybám je hlavným cieľom softvérových štandardov v spomínaných oblastiach. Jednou z možností ako uľahčiť programátorom prácu, je používanie návrhových vzorov. V súčasnej dobe bolo urobené veľa výskumu v oblasti návrhových vzorov a automatického refaktoringu zdrojových kódov. Avšak dané výskumy sa nevenovali návrhovým vzorom paralelných a distribuovaných systémov.

Spoločnou požiadavkou všetkých týchto štandardov je požiadavka na spoľahlivosť, ktorú možno dosiahnuť pomocou návrhových vzorov. V tejto práci je navrhnutý spôsob zápisu paralelných návrhových vzorov, ktorý umožní ich vkladanie do existujúcich paralelných zdrojových kódov. Navrhovaný spôsob zápisu detekuje nesprávne používanie súbežnosti a synchronizácie a doporučuje vhodné riešenie pomocou príslušného návrhového vzoru. Tento systém je založený na statickej analýze kódu slúžiacej na vyhľadávanie v kóde a formálnom opise paralelných návrhových vzorov.

Hlavným cieľom tejto práce je pomocou kombinácie existujúcich techník a metodík vytvoriť novú metodiku určenú na zápis paralelných návrhových vzorov, ktorá bude využiteľná na automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. Na to, aby bolo možné vytvoriť takúto metodiku je nutné odpovedať na nasledujúce 3 otázky:

1. Ako určiť miesto, kde treba vložiť vzor – analýza kódu.
2. Ako daný vzor reprezentovať, aby ho bolo možné vložiť do kódu – špecifikácia / definícia vzoru.
3. Pomocný problém: ako daný vzor vložiť do kódu – refaktoring.

2 Východiská práce

Medzi východiská práce patria tri nosné témy na ktorých aplikácii je založená celá dizertačná práca. Prvou témou je analýza kódu, ktorá umožňuje definíciu miesta kam je možné vložiť návrhový vzor. Druhou témou je téma návrhových vzorov, ktorá opisuje návrhové vzory a spôsoby ich zápisu. Poslednou nosnou témou je téma refaktoringu. nasledujúce 3 kapitoly prinášajú prehľad týchto nosných tém.

2.1 Analýza kódu

Cieľom tejto kapitoly je uviesť základný prehľad k analýze kódu. Kapitola sa hlavne sústredí na prehľad rôznych prístupov iných autorov. Analýza softvéru sa využíva na sledovanie správania programu, prípadne na získanie užitočných informácií o programe. V podstate sa jedná o proces automatického odvodzovania vlastností správania určitého programu [16]. Tieto vlastnosti môžu zahŕňať tok dát, využitie pamäte, volanie funkcií a pod. V rámci analýzy sa používa množstvo rôznych techník, z ktorých každá používa iný prístup a vedie ku skúmaniu iných vlastností. Podľa základnej povahy týchto techník sa analýza programov delí na dve hlavné časti - dynamickú a statickú analýzu. Okrem dynamickej a statickej analýzy sa kapitola venuje aj témam metrík kódu a vyhľadávaniu v zdrojových kódoch.

Dynamická analýza

Dynamická analýza je založená na spúšťaní analyzovaného programu (väčšinou binárneho kódu). Môže sa jednať iba o jedno spustenie (napr. pri technikách určených na získanie štatistík o programe), alebo o opakované spúšťanie analyzovaného programu. V takom prípade sú výsledné vlastnosti odvodené analýzou všetkých získaných behov [16]. Na rozdiel od statickej analýzy, ktorá obsiahne všetky možné behy programu, je tá dynamická limitovaná množinou reálne vykonaných behov. To môže znamenať v určitých prípadoch nevýhodu, keď je dynamická analýza chybná, pretože nepokryla všetky rôzne možnosti. Na druhej strane má táto skutočnosť aj svoju výhodu, keďže nikdy nedôjde k problému analýzy falošných poplachov (angl. false positives, niekedy aj false alarms), teda chybných varovaní analyzátoru o možnej chybe, ktoré sa môžu vyskytnúť v statickej analýze [8]. Ďalšou z výhod dynamickej analýzy je, že nie je nutné vytvárať nijakú abstrakciu, takže nemusí dôjsť ku strate informácií (aj keď vo väčšine prípadov sa abstrakcia robí kvôli zjednodušeniu a urýchleniu) [8].

Statická analýza

Statická analýza [21] je založená na analýze v čase kompilácie, takže nepotrebuje aby bol zdrojový kód spustiteľný. Existuje mnoho rôznych prístupov k statickej analýze od pomerne jednoduchých, ktoré hľadajú kód podľa vzorov opisujúcich nesprávne postupy po pomerne zložité a niekedy aj úplné analýzy. Medzi najznámejšie metódy statickej analýzy patrí analýza toku dát (data-flow analysis). Abstraktná interpretácia a model checking sú niekedy považované za súčasť statickej analýzy.

Statická analýza, na rozdiel od testovania a dynamickej analýzy, nie je obmedzená na posúdenie správania programu na základe jeho behu. Môže teoreticky pokryť všetky možné správania programu. Statická analýza musí bojovať s exponenciálnym počtom možných scenárov plánovania, čo činí analýzu viacvláknových programov pomerne ťažkú. Z tohto dôvodu existujú rôzne statické analýzy, ktoré využívajú iba približné správanie vlákien. Čím viac aproximácie používajú, tým väčšie množstvo zdrojového kódu dokážu analyzovať, ale za cenu viac falošných poplachov.

Existujúce nástroje na statickú analýzu

Existujú rôzne spôsoby, ako zabezpečiť kvalitu softvéru, vrátane revízie kódu a dôkladného testovania. Chyby softvéru môžu stáť spoločnosti značné množstvo peňazí, najmä keď vedú k zlyhaniu softvéru [24]. Statické analytické nástroje poskytujú prostriedky pre analýzu kódu, bez toho aby musel byť daný kód spustený, čo pomáha zaistiť kvalitnejší softvér v celom procese vývoja. Existuje celý rad spôsobov, ako vykonávať automatickú statickú analýzu [13].Po spustení vývojárom, neustále pri písaní kódu vo vývojovom prostredí, alebo tesne predtým, než sa softvér odošle do systému pre správu verzií. Tieto nástroje umožňujú vývojárom nakonfigurovať, aké druhy chýb majú hľadať, a niekedy dokonca umožňujú definovať nové chybové vzory.

FindBugs [22] je detektor chybových vzorov pre Javu. FindBugs používa množinu ad-hoc techník, ktorých cieľom je vyvážiť presnosť, efektivitu a použiteľnosť. Jednou z hlavných techník, ktoré FindBugs používa, je syntaktické porovnávanie zdrojových kódov so známymi podozrivými kódmi.

FindBugs je statický analytický nástroj, ktorý skúma triedy alebo súbory JAR. Hľadá potenciálne problémy tým, že porovnáva zostavené aplikácie so zoznamom chybových vzorov. FindBugs používa vzor Visitor na realizáciu functionalít jeho detektorov vzorov.

Napríklad FindBugs kontroluje, či volanie wait(), ktoré sa používa u viacvláknových programov v jazyku Java, je vždy volané v cykle, ktoré je správnym použitím vo väčšine prípadov. V niektorých prípadoch FindBugs tiež používa analýzu toku dát (dataflow) pre kontrolu chýb. Napríklad FindBugs využíva jednoduchú intraprocedurálnu (v rámci jednej metódy) analýzu toku dát na kontrolu NULL ukazovateľov. FindBugs môže byť rozšírený pomocou pridania vlastných detektorov napísaných v Jave.

JLint [2] je open source nástroj na statickú analýzu kódu, ktorý uľahčuje kontrolu kódu a hľadanie chýb v bytekóde jazyka Java, chyby a problémy so synchronizáciou pomocou analýzy toku dát a vytvárania grafu zámkov. Jlint bol vyvinutý Konstantinom Knizhnikom a ďalej rozšírený Cyrille Arthom o dôkladnejšiu kontrolu synchronizácie. Jlint sa skladá z dvoch rôznych programov určených na kontrolu syntaxe a sémantiky. Sémantický verifikátor Jlint predovšetkým extrahuje informácie zo súborov tried Java a využíva informácie o ladení na asociáciu hlásených chýb so zdrojovými kódmi. Vzhľadom k tomu, Java väčšinou dedí C / C ++ syntax, je teda Jlint je schopný overiť syntax pre všetky jazyky z rodiny C jazykov, ako je C, C ++, Objective-C atď. Pôvodne bol tento program nazvaný AntiC, pretože opravoval väčšinu problémov s C gramatikou, ako sú chyby operátorov priority, absencia príkazu break v kóde príkazu switch.

PMD [3], rovnako ako FindBugs a JLint, vykonáva syntaktickú kontrolu zdrojo-

vého kódu programu, ale bez analýzy dátového toku. Okrem detekcie jasne chybného kódu, mnohé z "chýb"ktoré PMD vyhľadá sú štylistické konvencie, ktorých porušenie môže byť podozrivé za určitých okolností. Napríklad, príkaz try s prázdnym blokom catch môže znamenať, že zachytená chyba je nesprávne ošetrená. Vzhľadom k tomu, PMD zahŕňa mnoho detektorov chýb, ktoré sú závislé na štýle programovania. PMD poskytuje možnosť výberu, ktorý detektor alebo skupiny detektorov by mal byť spustený. PMD je ľahko rozšíriteľný pomocou nových detektorov chybových vzorov ktoré môžu byť napísané buď pomocou Java alebo XPath

Checkstyle [1] vytvára syntaktický strom zo zdrojového kódu jazyka Java a vyvolá submoduly nazvané kontroly, pri priechode uzlami stromu. Každý uzol syntaktického stromu definuje token. Návšteva uzla počas priechodu spustí všetky kontroly, ktoré sú konfigurované pre daný token. Napríklad v prípade, že kontrola MethodLength bola nakonfigurovaná ako submodul, potom návšteva uzla s metódou alebo token definície konštruktora spúšťa MethodLength na kontrole počtu riadkov kódu bloku uzla.

Niektoré kontroly, ako FileLength a LineLength sa aplikujú priamo na zdrojové súbory a nezahŕňajú tokeny zo syntaktického stromu. Ostatné kontroly sú spojené s nastaviteľnými sadami tokenov, ktoré vedú ku kontrolám.

StyleCop je voľne šíriteľný statický analyzátor zdrojových kódov pre C# vývojárov, ktorý bol pôvodne vyvinutý spoločnosťou Microsoft. Riadenie a koordinácia projektu StyleCop je riadený .NET komunitou. StyleCop je dobre integrovaný do Visual Studia a varuje vývojárov, ak nenasledujú štandardy jazyka. Kódovacie štandardy sú určené na zlepšenie čitateľnosti, konzistencie, a udržateľnosti. StyleCop je statický analytický nástroj, ktorý poskytuje vývojárom efektívny spôsob, ako sledovať štandard kódovania pomocou definície široko používaného C# štandardu programovania. C# štandard, ktorý definuje StyleCop je široko používaný a mnoho vývojárov ho používa. Štandardy sú o definícii štýlu programovania a písania kódu. Zámerom zriadenia a dodržiavania štandardov písania kódu je, aby bol zdrojový kód čitateľnejší a jednoduchšie udržiavateľný

2.2 Návrhové vzory

Návrhové vzory boli prvý krát použite a popísane Alexandrom, ale nie v kontexte softvérového inžinierstva, ale v architektúre a priestorovom plánovaní. [7, 6] V architektúre sa stretol s odmietnutím, avšak na poli softvérového inžinierstva boli jeho publikácie inspiráciou pre tvorbu podobných zbierok znovu použiteľných vzorov.

V oblasti návrhu mal Alexander mnoho nasledovníkov. Najznámejším bol však Gamma, ktorý už v rámci svojej dizertačnej prace a neskôr v publikácii [20] definoval 23 návrhových vzorov, pri ktorých jasne definoval daný vzor a štandardný prípad jeho použitia. Návrhové vzory sú primárne určené pre objektovo orientovaný vývoj, konkrétne pre fázu podrobného návrhu. Model použitý na popis návrhových vzorov sa líši od publikácie k publikácii, avšak idea opakovaného použitia vzoru v podobnom kontexte zostava, vid nasledujúce definície.

Návrhové vzory v oblasti objektového návrhu softvéru a kvality zdrojového kódu sú definované ako osvedčené postupy riešenia opakujúcich sa všeobecných problémov návrhu, naopak nesprávne postupy sa nazývajú anti-vzory / chybové-vzory. Návrhové

vzory sú zvyčajne definované ako vzťahy medzi jednotlivými objektami softvérového systému, prípadne sú definované priamo na úrovni tried a definujú ich konkrétnu štruktúru tried. Napríklad návrhový vzor dekorátor umožňuje dynamicky pridať funkcionalitu na objekt, jednoduchšie než vopred definovaná funkcionalita prostredníctvom dedičnosti, takže umožňuje pridať funkcionalitu k objektom v čase behu. Tento vzor taktiež znižuje väzbu medzi komponentmi, takže môžu byť modifikované bez ovplyvnenia sa navzájom.

Opakom návrhových vzorov sú anti-vzory, ktoré sú v podstate vzory uplatňované v nevhodnom kontexte. Existujú dva typy Anti-vzorov. Prvý z nich je použitie vzoru v nesprávnom kontexte, druhý je zlý vzor, ktorý možno použiť kdekoľvek. Ďalším príkladom nevhodného kódu sú takzvané pachy v kóde, ktoré ničia kód napríklad dlhými metódami alebo duplicitou kódu. Pachy v kóde sa najčastejšie vyskytujú na úrovni funkcií prípadne tried. Anti-Vzory sú zvyčajne spojené so štrukturálnymi problémami, ako je napríklad nevhodná hierarchia tried. Pachy kódu vznikajú najčastejšie ako chyby implementácie, zatiaľ čo anti-vzory bývajú spojené s nevhodným návrhom. V mnohých prípadoch nie je jednoduché určiť, či sa jedná o pach kódu, alebo o vážnejší anti-vzor.

F. Buschmann vo svojej publikácii [14] opisuje návrhové vzory nasledovne. Vzor definuje ako dvojicu: opakujúci sa problém - riešenie v danom kontexte. Vzor však nie je čisto len popis problému, alebo popis štruktúry riešenia, vzor obsahuje oboje vrátane odôvodnenia, ktoré ich spája. Problém je posudzovaný s ohľadom na možne konflikty v návrhu a dôvody prečo daný problém je problém. Navrhované riešenie je uvádzané v podmienkach jeho očakávanej štruktúry a zahŕňa jasný popis prínosov a nevýhod daného riešenia.

T. Taibi vo svojej publikácii [28] opisuje návrhové vzory nasledovne. Návrhové vzory sú abstrakcie generované z cenných skúsenosti vývojárov ktoré získali pri riešení problémov s ktorými sa opakovane stretli v určitých kontextoch. Návrhové vzory sú podrobne testované a používane v mnohých vývojových a výskumných projektoch, ich opakovane použitie im poskytuje zlepšenie kvality softvéru pri súčasnom znížení času potrebného na vývoj.

Návrhové vzory bývajú často definovane neformálnym spôsobom technickou angličtinou, pripadne polo formálne pomocou UML diagramov. Neformálny zápis sa nedá algoritmicky spracovať a tým pádom slúži len pre vývojárov. Danému problému sa venuje Bayley vo svojej práci a publikáciách [9, 12, 29, 11, 10] nasledovne. Pôvodný účel návrhových vzorov uvedený v [7] je "zachytiť skúsenosť vývojára v podobe ktorú môžu ľudia efektívne používať". Preto sú návrhové vzory definovane vysvetlením všeobecných zásad v neformálnej angličtine a objasnené s formálnymi všeobecnými diagramy tried a konkrétnymi príkladmi kódu. Tato kombinácia je dostatočná pre vývojárov softvéru, ktorý dôkazu odhadnúť, ako aplikovať návrhové vzory pri riešení svojich vlastných problémov. Avšak, takýto zápis spôsobuje problémy pri snahe o automatické úpravy zdrojových kódov. Ak by návrhové vzory boli formalizované, mohli by softvérové nástroje refaktorovať zdrojové kódy v súlade s vybranými návrhovými vzormi.

2.3 Refaktoring

Prvé použitie termínu refaktoring v literatúre bolo v práci Opdyka a Johnsona [25, 26], hoci v praxi bola táto technika používaná dávno pred tým. Opdyke definuje refaktoring

ako "reštrukturalizáciu programu zachovávajúcu správanie". Fowler používa podobnú definíciu, ale zdôrazňuje, že od procesu refaktoringu očakávame zlepšenie dizajnu: Refaktoring je proces zmeny softvérového systému takým spôsobom, že sa nemení vonkajšie správanie kódu, ale zlepšuje jeho vnútorná štruktúra [19, 18]. Roberts mení definíciu radikálne tým, že umožňuje refaktoring, ktorý mení správanie programu .

Počas svojho výskumu Opdyke definoval sadu transformácií programov, použiteľných na programy v jazyku C++. V ďalšej svojej práci ukázal, ako by mohli byť dané transformácie použité na definíciu transformácií vyššej úrovne, napríklad na prevod dedičnosti na agregáciu, a naopak [23]. Roberts vo svojom výskume rozšíril prácu Opdyka tým, že poskytol formálny model pre skladanie transformácií, a skúmal využitie dynamických informácií v refaktoringu. Nasledujúce kapitoly sa venujú prístupom k problematike automatického refaktoringu. V mnohých prípadoch sa termín refaktoring nepoužíva, ale podstata daných prác spočíva v transformácii kódu so zachovaním vonkajšieho správania.

Jedným z hlavných prínosov objektovo orientovaného návrhu je dedičnosť. Projektovanie tried a ich hierarchie je však stále náročné, aj keď bolo veľa pokusov o vytvorenie automatickej podpory tejto činnosti. Pravdepodobne najstaršia práca, ktorá sa touto otázkou zaoberala bola od Puna a Windera [27]. Keď dizajnér pridá triedu do hierarchie, môže sa stať, že hierarchia tried spôsobí dedenie nežiaducich vlastností. To znamená, že hierarchia by mala byť nanovo usporiadaná, aby oddelila nežiaduce atribúty od tých, ktoré majú byť zdedené. Pun a Winder ukazujú, ako môže byť tento proces reorganizácie automatizovaný a čiastočne formalizovaný.

3 Návrh spôsobu zápisu návrhových vzorov

Ako už bolo spomenuté v úvode, hlavným prínosom práce je návrh spôsobu zápisu návrhových vzorov, ktorý umožní asistované vkladanie paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov. Zdrojový kód vytvorený s pomocou návrhových vzorov je efektívnejší, čitateľnejší, jednoduchšie spravovateľný a tým pádom aj spoľahlivejší.

Na to aby bolo možné splniť požiadavky uvedené vyššie je potrebné byť schopný pracovať s návrhovým vzorom automaticky, čo znamená že návrhový vzor musí byť definovaný formálne. Druhá požiadavka je, že návrhový vzor musí obsahovať špecifikáciu miesta, kam sa má vložiť. Tretia požiadavka je, že špecifikácia návrhového vzoru musí umožňovať jeho automatické vkladanie do existujúcich zdrojových kódov. Na základe týchto požiadaviek definujme návrhový vzor ako dvojicu (P, d) kde P je množina preconditions, určujúca vhodné umiestnenie vzoru, d je samotný popis vzoru. Množina P vychádza z analýzy kódu opísanej v nasledujúcej kapitole, ktorá umožňuje vytvárať dopyty nad existujúcim zdrojovým kódom, ktorých výsledok je miesto v zdrojovom kóde. Špecifikácia vzoru definuje jeho štruktúru a správanie a je opísaná v kapitole nižšie.

3.1 Analýza kódu

Na začiatok špecifikácie použitého algoritmu, je potrené zhrnúť použitú notáciu. Použitý algoritmus berie ako vstup uzavretý program s hlavnou metódou označenou m_{main}. Definujme $\mathbb{M}$ ako označenie množiny všetkých funkcií, ktoré môžu byť dosiahnuteľné z m_{main}. $\mathbb{M}$ môže byť jednoduchou nadaproximáciou, napríklad vypočítaný analýzy hierarchie tried. Definujme $m_{start} \in \mathbb{M}$ ako označenie pre funckiu Start() triedy `System.Threading.Thread`, ktorá slúži na explicitné vytvorenie nového vlákna. Definujme $\mathbb{I}$ množinu všetkých volaní funkcií v tele každej z metód $m \in \mathbb{M}$. Definujme $\mathbb{H}$ množinu všetkých alokácií objektov v tele každej z metód $m \in \mathbb{M}$. Definujme $\mathbb{V}$ ako označenie množiny všetkých lokálnych premenných referencovanými metódami $m \in \mathbb{M}$. Definujme $\mathbb{F}$ ako označenie množiny všetkých premenných aktuálneho objektu referencovanými metódami $m \in \mathbb{M}$. V prípade polí, sa nahradzujú objekty na indexe samostatnými virtuálnymi premennými s indexom v názve. Definujme $\mathbb{P}$ ako označenie množiny všetkých príkazov použitých v metódach $m \in \mathbb{M}$, do tejto množiny patria volania konštruktorov objektov, volania funkcií, čítanie a zápis premenných z a do pamäte. Predpokladáme, že každá metóda $m \in \mathbb{M}$ môžu byť synchronizované pomocou cez niektorý zo svojich svojich argumentov, čo zapíšeme pomocou sync(o, m), a súčasne neobsahuje žiadne iné synchronizované bloky kódu v tele. Obrázok 1 ukazuje aj relácie použité počas statickej analýzy: (cg) analýza grafu volaní, (pt) analýza ukazatelov, (esc) analýza vlákien a (mhp) analýza súbežnosti akcií.

$(funkcia)$	$m \in M = m_{main}, m_{start}, ...$
$(lokalna\ premenna)$	$v \in V$
$(miesto\ alokacie)$	$h \in H$
$(mnozina\ miest\ alokacii)$	$[h_1 :: ... :: h_n] \in H^n$
$(prikaz)$	$p \in P$
$(abstraktny\ objekt)$	$o \in \mathbb{O} = \mathbb{H}^0 \cup \mathbb{H}^1 \cup \mathbb{H}^2 \cup ...$
$(abstraktny\ kontext)$	$c, t, l \in \mathbb{C} = \mathbb{O} \times \mathbb{M}$
$(graf\ volani)$	$\text{cg} \subseteq (\mathbb{C} \times \mathbb{C})$
$(analyza\ ukazatelov)$	$\text{pt} \subseteq (\mathbb{C} \times \mathbb{V} \times \mathbb{O})$
$(analyza\ vlakien)$	$\text{ta} \subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$
$(analyza\ zamkov)$	$\text{la} \subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$

Obr. 1: Notácia použitá v popise algoritmu analýzy kódu.

Na vytvorenie grafu volaní a analýzu ukazovateľov slúži k-object-sensitive analysis [x14]. Táto analýza je objektovo citlivá *(object sensitive)* čo znamená, že dokáže reprezentovať oddelené inštancie objektov, vytvorené rovnakým kódom, pomocou volaní z potencionálne rôznych inštancií voajúcich objektov. Inštancia objektu, alebo abstraktný objekt zapísaný pomocou $o \in \mathbb{O}$, je konečná množina miest alokácii objektu, ktoré sú zapísané pomocou $[h_1 :: ... :: h_n]$. Prvé miesto alokácie h_1, špecifikuje miesto

v kóde ktoré vytvorilo aktuálnu inštanciu v kontexte objektu ktorý bol alokovaný pomocou inštancie objektu $[h_2 :: ... :: h_n]$ a tak ďalej až po prvý alokovaný objekt. V prípade statických metód sa použije zápis [ ] ktorý znamená žiaden alokovaný objekt nad ktorým sa vykonáva zvolená statická funkcia.

Použitá analýza je aj kontextová *(context sensitive)* čo znamená, že dokáže analyzovať implementáciu metódy v rôznych abstraktných kontextoch. Abstraktný kontext $c \in \mathbb{C}$ je pár (o, m), kde o je inštancia objektu a m je metóda v ktorej použite kľúčového slova `this` vráti objekt m. Pre statické metódy $o = [\ ]$. Táto analýza umožňuje definovať nasledovné relácie:

- cg $\subseteq (\mathbb{C} \times \mathbb{C})$ kontextový graf volaní, obsahuje každú dvojicu $((o_1, m_1), (o_2, m_2))$ takú, že metóda m_1 inštancie objektu o_1 volá metódu m_2 nad inštanciou objektu o_2.

- pt $\subseteq (\mathbb{C} \times \mathbb{V} \times \mathbb{O})$ výsledok analýzy ukazovateľov obsahuje množinu trojíc (c, v, o) takých, že lokálna premenná v môže ukazovať na inštanciu objektu o v kontexte c.

- ta $\subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$ výsledok analýzy vlákien obsahuje množinu trojíc (c, p, o) takých, že príkaz p inštancie objektu o je vykonávaný v kontexte c.

- la $\subseteq (\mathbb{C} \times \mathbb{P} \times \mathbb{O})$ výsledok analýzy zámkov obsahuje množinu trojíc (c, p, o) takých, že príkaz p inštancie objektu o je vykonávaný v kontexte c.

Relácie grafu volaní a analýzy ukazovateľov sa naplnia nasledovným spôsobom. Analýza začína naplnením kontextu statickej metódy `Main` ([ ], Main) a statických konštruktorov použitých tried ([ ], Class.ctor). Analýza predpokladá kladné celé číslo priradené ku každému miestu alokácie objektu, nazvané k-hodnota daného miesta. Uvažujme akékoľvek miesto alokácie $v = \mathbf{new}^h ...$ kde $h \in \mathbb{H}$ a $v \in \mathbb{V}$, v metóde m ktorá je dosiahnuteľná v kontexte (o, m). Potom analýza pridá trojicu $((o, m), v, h \oplus_k o)$ do relácie pt, kde $h \oplus_k o$ značí konečnú neprázdnu usporiadanú množinu miest alokácii objektu, ktorej prvý prvok je h a ďalej obsahuje zoradených $k-1$ miest alokácii v poradí v akom boli alokované. Pre príklad z obrázku **??**, ktorého kontext ([ ], A) obsahuje alokáciu dvoch premenných t1 = $\mathbf{new}^{h_1}$ a t2 = $\mathbf{new}^{h_2}$ pribudnú nasledovné relácie do pt (([ ], A),v1,[h_1]) a (([ ], A),v2,[h_2]).

Ak $n(...)$ je volanie statickej metódy v dosiahnuteľnom kontexte (o, m), tak analýza prída dvojicu $((o, m), ([\ ], n))$ do cg, za predpokladu, že kontext $([\ ], n)$ je dosiahnuteľný. Ak $v.n(...)$ je inštančné volanie metódy, tak volaná metóda závisí od aktuálneho typu objektu v. Každá trojica $((o, m), v, [h_1 :: ... :: h_n]) \in$ pt môže označovať rôzne metódy v rôznych kontextoch. Analýza preto pridáva $((o, m), [h_1 :: ... :: h_n], n')$ do cg, kde n' označuje cieľ volania n nad objektom alokovaným pomocou h_1. Analýza predpokladá, že kontext $([h_1 :: ... :: h_n], n')$ je dosiahnuteľný. V našom príklade z obrázku **??**, ktorého kontext ([ ], A) obsahuje volanie dvoch funkcií, `t1.Start()` a `t2.Start()`, analýza pridá nasledové dvojice do cg (([ ],A),([h_1],m_{start})) a (([ ],A),([h_2],m_{start})).

Výstupom analýzy vlákien je relácia, definovaná pomocou trojice (c, p, o), v ktorej príkaz p abstraktného objektu o je dosiahnuteľný v kontexte c. Kedže každé vlákno je v .Net vykonávané nad inštanciou objektu typu `System.Threading.Thread`, môžeme

každé unikátne vlákno označiť pomocou inštancie objektu t.j. kontextu v ktorom je vykonávané.

Majme metódu m s unikátnym počiatočným príkazom p v abstraktnom kontexte c', volanú z metódy n kontextu c. Relácia pt, neobsahuje ani jednu lokálnu premennú v z metódy m. Ak má metóda neprázdnu množinu vstupných parametrov, tak relácia pt, obsahuje zjednotenie relácii pt pre všetky argumenty analyzovanej metódy. Abstrakcia haldy a relácia ta, obsahuje zjednotenie haldy a relácie ta zo všetkých príkazov volajúcich metódu m. Ak $m = m_{start}$ čo značí, že m je metóda `Start()` objektu typu `System.Threading.Thread`, ktorá je zodpovedná za vytvorenie nového vlákna, tak všetky príkazy nasledujúce po volaní metódy m v metóde n a kontexte c je označený ako dosiahnuteľný z novo vytvoreného vlákna.

Analýza zámkov je naplnená rovnakým algoritmom ako analýza vlákien. S tým rozdielom, že sa sleduje držanie a uvoľňovanie zámkov. V našom výskume, pre zjednodušenie uvažujeme iba zámok typu monitor, ktorého zamykanie je definované na obrázku 2.

```
MONITORENTER(mon) =
  if mon = Null then FAILUP(ArgumentNullException)
  elseif lockOwner(mon) = self then
    lockCount(mon, self) := lockCount(mon, self) + 1
    YIELDUP(Norm)
  elseif lockOwner(mon) = None ∧ Empty(readyQueue(mon)) then
    LOCK(self, mon)
    lockCount(mon, self) := 1
    YIELDUP(Norm)
  elseif interruptRequested(self) then THROWINTERRUPTEDEXCEPTION
  else
    readyQueue(mon) := readyQueue(mon) · [self]
    monObj(self) := mon
    execState(self) := Syncing
    YIELDUP(Norm)
```

Obr. 2: Definícia zamykania použitého zámku.

Na základe takto získaných dát, dokážeme definovať miesta v kóde s nesprávnou synchronizáciou medzi vláknami, prípadne s nesynchronizovaným prístupom ku zdieľaným premenným. Tieto dva prípady sú najčastejšími dôvodmi na vloženie návrhového vzoru a preto sú popísané v tejto práci. Samozrejme existujú aj iné dôvody na vloženie paralelných návrhových vzorov do existujúcich zdrojových kódov, ale špecifikácia príslušných anti-vzorov sa dá získať jednoduchým rozšírením vyššie popísaného algoritmu. Špecfikáciou miesta v kóde na ktoré by sa mal aplikovať návrhový vzor je $p \in \mathbb{P}$ ktoré vieme jednoznačne definovať pomocou výrokov predikátovej logiky nad reláciami cg, pt, ta a la.

3.2 Špecifikácia vzoru

Špecifikácia vzoru využíva údaje zistené počas analýzy na definíciu miesta kam sa má vložiť vzor. Na to aby sme dokázali vložiť vzor potrebujeme minimálne jeden bod ktorého sa môžeme chytiť. Týmto bodom je miesto s vadným kódom a je označené značkou l. Nasledujúci text uvádza použitú formálnu špecifikáciu vzoru, ktorá je rozšírením existujúceho jazyka BPSL [28]. Rozšírenie spočíva v pridaní možnosti označovať metódy, atribúty, premenné a objekty značkami l, ktoré umožňujú priradiť k daným bodom zo vzoru precondition, ktorá určuje kam je vhodé daný vzor vložiť. Ďalej je jazyk rozšírený o množinu aktívnych vlákien TA a zámkov LA použitých v danom vzore.

Formálna špecifikácia návrhového vzoru používa existujúci formálny jazyk BPSL (Balanced Pattern Specification Language) [28], ktorý umožňuje špecifikáciu návrhových vzorov jak z pohľadu štruktúry, tak aj z pohľadu správania. Tento formálny jazyk sa radí do kategórie formálnych systémov, ktoré vhodným spôsobom kombinujú už existujúce formálne mechanizmy a prispôsobujú ich tak, aby ich bolo možné použiť na opis návrhových vzorov. K definícii štruktúry používa predikátovú logiku 1. rádu, k definícii správania temporálnu logiku akcie. na to aby mohol byť tento jazyk použitý pre potreby tejto práce, bolo nutné ho rozšíriť o informácie o vláknach a zámkoch. Toto rozšírenie je opísané v špecifikácii štruktúry.

Špecifikácia štruktúry

BPSL využíva k špecifikácii štruktúry vzoru podmnožinu predikátovej logiky 1. rádu (najmä premenné a predikáty), pretože vzťahy medzi účastníkmi vzoru môžu byť ľahko vyjadrené ako predikáty. Temporálne relácie medzi účastníkmi vzoru a ich správanie umožňuje definovať pomocou podmnožiny temporálnej logiky akcie (používa najmä akcie a premenné popisujúce stav). BPSL tak vhodným spôsobom kombinuje dva formálne systémy a vďaka tomu môžeme jedným jazykom definovať správanie i štruktúru vzoru. BPSL uvažuje ako hlavný stavebné prvky, ktoré odrážajú entity a relácie:

1. Triedy, atribúty, metódy, objekty, netypované hodnoty, ktoré nazýva primárnymi entitami.
2. Trvalé a dočasné relácie.
3. Akcie.
4. Všetky ďalšie prvky musia byť odvodené z trvalých relácií alebo primárnych entít.

Primárne entity jazyk chápe ako základné prvky štruktúry návrhových vzorov, pričom uvažuje možnosť objektov a netypovaných hodnôt vyskytovať sa ako parametre metód. Trvalé relácie definuje jazyk ako relácie, ktoré sa po vytvorení už nemôžu meniť, zatiaľ čo dočasné relácie sa môžu meniť v priebehu správania vzoru. Akcie jazyk opisuje ako atomické jednotky činnosti, kde zostavenie viac takýchto jednotiek definuje správanie vzoru.

K definícii štruktúry sú využité prvky predikátovej logiky 1. rádu, najmä premenné, logické spojky, existenčný kvantifikátor a predikáty. Premenné reprezentujú primárne

entity, zatiaľ čo predikáty reprezentujú trvalé relácie medzi nimi. Nech triedy sú označené premennou C, atribúty A, metódy M, objekty O, netypované hodnoty V, označené body programe L, vlákna programu TA a zámky programu LA, potom tabuľka 1 popisuje možné trvalé relácie medzi účastníkmi vzťahu realizované ako predikáty.

Názov	**Doména**	**Význam**
DefinedIn	M × C	Metóda je definovaná v konkrétnej triede
	A × C	Atribút je definovaný v konkrétnej triede
ReferenceTo one (many)	C × C	Trieda obsahuje referenciu na jednu (viacej) inštancií triedy
Inheritance	C × C	Prvá trieda dedí z druhej
Creation	M × C	Metoda vytvára nové inštancie triedy
	C × C	Jedna z metód prvej triedy vytvára inštanciu druhej triedy
Invocation	M × M	Prvá metóda volá druhú metódu
Argument	C × M	Argumentom metódy je referencia na triedu
	V × M	Argumentom metódy je hodnota
Instance	O × C	Objekt je inštanciou danej triedy
Label	M × L	Metóda je označená značkou
	V × L	Hodnota je označená značkou
	O × L	Objekt je označený značkou
	A × L	Atribút je označený značkou

Tabuľka 1: Možné trvalé relácie medzi účastníkmi vzťahu realizované ako predikáty BPSL.

Špecifikácia správania

U niektorých návrhových vzorov nestačí samotná definícia ich štruktúry, ale tiež je veľmi dôležitá definícia ich správania, tj. popis, ako účastníci vzoru spolu spolupracujú. BPSL opisuje správanie vzoru pomocou podmnožiny temporálnej logiky akcie. Jazyk opisuje správanie vzoru ako nekonečnú sekvenciu stavov, ktorými entity vzoru prechádzajú. Každý stav možno chápať ako kolekciu hodnôt stavových premenných (napr. hodnoty atribútov tried) a dočasných vzťahov medzi objektami.

Dvojica po sebe idúcich stavov sa nazýva prechod. Systém sa na začiatku nachádza v počiatočnom stave a postupne, ako sa vykonávajú jednotlivé akcie, prechádza jednotlivými stavmi. Akcia sú vyberané nedeterministicky a každá má stanovenú vstupnú podmienku, ktorá musí byť splnená, aby sa akcia uskutočnila. BPSL naviac oproti definícii temporálnej logiky akcie rozširuje sémantiku akcií. V BPSL sa počas vykonávania akcií nielen menia hodnoty stavových premenných, ale aj dočasné vzťahy medzi objektmi (môžu vznikať nové, zanikať existujúce).

Dočasné relácie (TR) BPSL definuje ako dvojicu $TR(C1[cardinality1], C2[cardinality2])$, kde TR je názov relácie, $C1$, $C2$ sú triedy a kardinality reprezentujú počet inštancií daných tried, ktoré sú spolu vo vzťahu. Kardinalitou môže byť konkrétny rozsah nezáporných celých čísel (zapísaný v tvare [m ... n]) alebo [*], čo znamená akýkoľvek počet. V akciách sa potom môžu vyskytovať zápisy v tvare napr. $TR(o1, o2)$, čo znamená, že objekt o1 je vo vzťahu s objektom o2, $\neg TR(o1, o2)$, čo znamená, že objekt o1 nie

už ďalej vo vzťahu s objektom o2, $TR(o1, C2)$ znamená, že objekt o1 je vo vzťahu so všetkými inštanciami triedy C2.

Akcie v BPSL sú definované ako zoznam vstupných parametrov, vstupné podmienky a tela (ako sa má zmeniť stav systému vykonaním akcie). Akcia A môže byť napr. Definovaná nasledovne: $Ao1, o2, p) : TR(o1, o2) \wedge o1.x \neq p \rightarrow \neg TR(o1, o2) \wedge o1.x' = p$, kde $o1$ je objekt triedy C1, $o2$ je objekt triedy C2, p je vstupný parameter akcie a x je atribút triedy C1. Je teda zrejmé, že výraz $TR(o1, o2) \wedge o1.x \neq p$ je vstupná podmienka akcie (ak nie je splnená, akcia sa nevykoná) a zvyšok výrazu je samotné telo akcie, x' znamená hodnotu atribútu x po vykonaní akcie. Ako už bolo povedané vyššie, objekty a hodnoty, ktoré sa akcie zúčastňujú, sú vybrané nedeterministicky. Preto sa akcia A vykoná pre všetky objekty tried C1 a C2, ktoré sú spolu v relácii.

BPSL nepoužíva vzájomne disjunktné množiny premenných v trvalých a temporálnych reláciách a akciách. Naopak kombinuje tieto množiny dohromady, tj. niektoré premenné, ktoré sa vyskytujú v definícii trvalých relácií, sa vyskytujú aj v definícii dočasných relácií a akcií. BPSL teda pozerá na návrhový vzor ako na kolekciu entít (tried, atribútov, metód, objektov, hodnôt), relácií (dočasných a trvalých) a akcií medzi nimi. Z formálneho hľadiska BPSL definuje návrhový vzor ako model $M =< E, R >$, kde E je univerzum entít a R je množina relácií (temporálnej / permanentných a akcií).

3.3 Príklad zápisu paralelného návrhového vzoru

V nasledujúcom texte bude zapísaný návrhový vzor Thread Safe Interface, popísaný vyššie v kapitole **??** pomocou navrhnutého spôsobu zápisu. Ako už bolo spomenuté na začiatku tejto kapitoly, tak bolo potrebné rozšíriť špecifikačný jazyk BPSL o novú množinu L ktorá obsahuje značky (labels) použité pri popise miest kam sa má daný vzor vložiť. Ďalej boj jazyk rozšírený o množinu preconditions, ktoré určujú miesto v analyzovanom zdrojovom kóde, kam je vhodné daný návrhový vzor vložiť. Prepojenie medzi definíciou miesta kam sa má vzor vložiť a popisom vzoru, umožňuje automatické vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov.

Tanuľka 2 obsahuje formálnu špecifikáciu návrhového vzoru Thread Safe Interface. V prvej časti tabuľky sú definované entity vystupujúce v systéme. V tomto prípade sa jedná o dve triedy *Client* a *Server*, ďalej atribut *serviceLock* ktorý slúži na zabezpečenie vzájomnej výlučnosti. V množine metód M musia existovať prvky *client_thread*, *service*, *service_imp*. Množina konkrétnych objektov O musí obsahovať dva objekty s a c. Množina premenných V je prázdna. A množina označení kódu obsahuje jeden prvok *label*.

Duhá časť tabuľky obsahuje množinu trvalých relácii. Trieda *Server*, musí obsahovať atribút *serviceLock*. Trieda *Client* musí obsahovať funkciu *client_thread*. Treida *Server*, ďalej obsahuje funkcie *service* a *service_imp*. Trieda *Client* si drží referenciu na tredu *Server*. Inštancia treidy *Client* je pomenovaná c a inštancia triedy *Server* je pomenovaná s. Nakoniec metóda *service_imp* je označená značkou *label*. Táto značka nám umožňuje priradiť tejto metóde precondition (výsledok analýzy kódu) ktorá určuje, kam by bolo vhodné aplikovať opisovaný návrhový vzor.

Tretia a štvrtá časť tabuľky definujú pomocou dočasných relácii správanie vzoru. V opisovanom vzore existujú dve dočasné relácie pomenované *Locked* a $Waiting$. Relácia

$\exists\ Client, Server \in C;$ $\exists\ serviceLock \in A;$ $\exists\ client\_thread, service, service\_imp \in M;$ $\exists\ c, s \in O;$ $V = \emptyset$ $\exists\ label \in L$
$DefinedIn(serviceLock, Server) \wedge$ $DefinedIn(client\_thread, Client) \wedge$ $DefinedIn(service, Server) \wedge$ $DefinedIn(service\_imp, Server) \wedge$ $ReferenceToOne(Client, Server) \wedge$ $Invocation(service, service\_imp) \wedge$ $Instance(c, Client) \wedge$ $Instance(s, Server)$ $Label(service\_imp, label)$
$Locked(Server[1], serviceLock[1])$ $Waiting(service[0..*], serviceLock[1])$
$Init : \neg Locked(s, serviceLock) \wedge \neg Waiting(s, serviceLock)$ $Service(s) : \neg Locked(s, serviceLock) \rightarrow Locked'(s, serviceLock) \vee$ $Locked(s, serviceLock) \rightarrow Waiting'(s, serviceLock)$
$\exists\ p \in service\_imp \wedge p = label;$ $\exists\ t_1, t_2 \in TA\ \bullet\ \{t_1, t_2\} \subseteq (\mathbb{C} \times p \times \mathbb{O});$ $\nexists\ lock \in LA\ \bullet\ \{lock\} \subseteq (\mathbb{C} \times p \times \mathbb{O});$

Tabuľka 2: Príklad formálneho zápisu paralelného návrhového vzoru pomocou navrhnutého spôsobu zápisu.

Locked umožňuje definovať počiatočný stav, kedy prístup k implementácii funkcie *service_imp* nieje blokovaný zámkom a súčasne relácia $Waiting$ popisuje, že žiadne volanie funcke *service_imp* nieje zablokované. V prípade volania funkcie $Service(s)$ sa zmení stav objektu na *Locked* alebo volajúce vlákno prejde do stavu $Waiting$.

Posledná časť tabuľky opisuje miesto kam je vhodné vložiť návrhový vzor pomocou výrazov predikátovej logiky nasledovne. Existuje príkaz p funkcie *service_imp*, a tento príkaz vystupuje v predošlej špecifikácii pod názvom *label*. V relácii výsledkov analýzy vlákien programu TA existujú relácie t_1 a t_2 také, že príkaz p je v nich obsiahnutý. Neexistuje žiaden záznam v relácii výsledkov analýzy zámkov LA taký, že príkaz p je v ňom obsiahnutý.

4 Využitie navrhnutého spôsobu zápisu

Navrhnutý spôsob zápisu paralelných návrhových vzorov bol využitý pri návrhu systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov, ktorý sa skladá z troch častí. Analyzátora kódu, ktorý je zodpovedný za spracovanie zdrojového kódu a jeho reprezentáciu pomocou štruktúry Searchable Code Model, ktorá je výsledkom tejto práce. Druhým modulom je modul vyhodnocovania kvality kódu, ktorý má na vstupe Searchable Code Model analyzovaného zdrojového kódu a

množinu špecifikácii návrhových vzorov, zapísaných vo formáte XML, ktorý vychádza z metodiky popísanej v predošlej kapitole. Tento modul vyhodnotí predložený kód oproti definovaným návrhovým vzorom a navrhne používateľovi vhodné návrhové vzory na zvýšenie kvality spracovávaného zdrojového kódu. Posledným modulom systému je modul refaktoringu, ktorý na základe špecifikácií návrhových vzorov vykoná príslušný refaktoring zdrojového kódu. Nasledujúci text zjednodušene opisuje navrhnutý spôsob zápisu implementovaný v navrhnutom systéme.

4.1 Zápis návrhových vzorov

Jedným z hlavných vstupov aplikácie je opis vzoru. Tento opis by mal byť nezávislý na platforme, ľahko rozšíriteľný, prenositeľný medzi aplikáciami, zrozumiteľný používateľom, pretože budú v tejto forme tvoriť definície ďalších vzorov, ale tiež ľahko validovateľný, aby aplikácia mohla spracovávať len správne definície. Preto by bolo vhodné zapisovať definície návrhových vzorov vo formáte XML, ktorý spĺňa všetky vyššie uvedené požiadavky. Validita XML dokumentov je zabezpečená pomocou XML schémy (napr. XSD), ktorú je možné použiť na kontrolu, či je definícia vzoru validná, alebo nie.

Popis vzoru sa skladá z dvoch hlavných častí: entít (značené pomocou <entities>) a vzťahov medzi nimi (značené pomocou <relations>). Za popisom vzoru nasleduje časť dopytov (značená pomocou <preconditions>). Časť <entities> obsahuje definíciu entít, ktoré sa vyskytujú v návrhových vzoroch. Sú tu najmä uložené informácie o atribútoch, metódach, vlastnostiach, konštruktore. Pri každej časti možno definovať viac modifikátorov. Pri každom elemente v množine <entities> je možné definovať atribut label, ktorý slúži ako väzobný bod pre množinu preconditions. Časť <relations> obsahuje definície relácií medzi entitami. Sú tu najmä uvedené informácie o asociáciách, dedičnosti, realizácii a pod. Každá táto relácia "spájaëntity, ktoré sú popísané v predchádzajúcej časti, a umožňuje určiť ich kardinalitu. Časť <precoditions> umožňuje definovať množinu vyhľadávacích dotazov do Searchable Code Model, ktoré umožnia definovať miesta v analyzovanom zdrojovom kóde, kam je možné vkladať popisovaný návrhový vzor.

Aby opis vzoru spĺňal požiadavky na aplikáciu opísanú vyššie, je nutné, aby bol dostatočne abstraktný. Len vďaka tomu je potom možné vyhľadať podobné štruktúry vzoru v kóde, ktoré nie sú úplne rovnaké so vzorom, ale zároveň dostatočne relevantné. Toho je možné dosiahnuť tak, že v popise vzoru sú názvy entít (tried, rozhrania, abstraktných tried a pod.) A dátové typy metód, atribútov, vlastností uvedené len v abstraktnom poňatí.

Dátové typy sú uvedené všeobecným označením napr. Typ 1. Sú navyše očíslované, aby bolo možné viaceré typov v rámci entity (napr. návratové typy metód v triede), ale zároveň pri zachovaní možnosti definície príslušnosti vlastnosti (atribútu alebo metódy) k triede. Algoritmus nájdenia vhodných entít (popísaný nižšie) potom neberie do úvahy konkrétny "typ"triedy alebo metódy, ale jeho abstraktnú variantu. Algoritmus je toto schopný zohľadniť pri vyhľadávaní podobných entít v zdrojovom kóde. Vďaka tomu je popis návrhového vzoru dostatočne všeobecný, ale zároveň umožňuje plne špecifikovať všetky vlastnosti vzoru.

4.2 Vkladanie návrhových vzorov

Kedže vkladanie návrhových vzorov je riešené len ako dodatočný problém, tak vkladanie vzorov je riešené pomocou jednoduchého mapovania existujúcej štruktúry kódu na návrhový vzor. Druhým možným riešením, ktoré nieje podrobnejšie študované je vytvorenie jednoúčelových transformácii / operácii refaktoringu ktoré priamo zavedú príslušný vzor. Nasledujúci text opisuje metódu navrhovania vkladanie vzorov pomocou mapovania štruktúry kódu a vzoru.

Aby bolo možné efektívne a správne vyhľadávať podobné entity vzoru v zdrojových kódoch, porovnávať ich a hodnotiť ich z hľadiska relevantnosti, je nutné, aby zdrojové kódy aj definícia návrhových vzorov boli v rámci algoritmu reprezentované rovnakou uniformnou štruktúrou. Zdrojové kódy sú načítané pomocou parsera použitého programovacieho jazyka, ktorého výstupom je abstraktný syntaktický strom (AST) kódu. Parsovanie kódov má aj ďalšiu výhodu. Súčasne s načítaním kódu parser vykonáva aj validáciu, lebo ak načítaný kód nie je validný, nemožno zostaviť korektne AST pre daný jazyk. Vďaka tomu sa po korektnom načítaní zdrojových kódov (a teda aj po úspešnom zostavení AST) prevedie už načítaná definícia vzoru taktiež na AST, ktorý je reprezentovaný zhodnou dátovou štruktúrou ako AST zdrojových kódov.

Ďalšou významnou časťou logiky aplikácie je algoritmus nájdenia vhodných entít. Jeho vstupom je popis vzoru a zdrojový kód, obaja sú reprezentované rovnakou štruktúrou. Cieľom je potom vyhľadať pre každú entitu vzoru zodpovedajúce entitu v zdrojovom kóde a jej ohodnotenie. Výsledkom algoritmu je zoznam podobných štruktúr zdrojového kódu k jednotlivým entitám vzoru. Tento algoritmus začína z bodov definovaných v množine preconditions, t.j. entít kódu, ktoré vyhľadávací algoritmus opísaný v predošlej kapitole analýza kódu označil ako kandidátov na vloženie daného vzoru.

4.3 Príklad zápisu vzoru pomocou navrhnutého systému

Majme popis návrhového vzoru Thread-Safe Interface z obrázku 3.

Zápis vzoru uvedený na obrázku 3 ukazuje návrhový vzor Thread-Safe Interface, v ktorom na zabezpečenie správnej synchronizácie, klientské vlákna volajú iba funkcie verejného rozhrania. Funkcia rozhrania získa potrebný zámok a zavolá príslušnú implementačnú funkciu ktorá sa už nestará o zamykanie a môže slobodne volať iné implementačné funkcie. Uviaznutie (self-deadlock) nemôže nastať pretože zámok sa získava iba raz na začiatku v rozhraní a pri rekurzívnom volaní funkcií je zámok získavaný iba raz, čo taktiež zvyšuje výkon programu.

Najdôležitejšou časťou použitého zápisu je element `<preconditions>` spoločne s atribútom `label` ktoré umožňujú definovať miesto v existujúcom zdrojovom kóde kam je vhodné daný vzor vložiť, súčasne s návrhom riešenia pomocou návrhového vzoru. V prípade vzoru Thread-Safe Interface, uvdený dopyt vracia všetky funkcie objektu, ku ktorým pristupuje viac ako jedno vlákno a súčasne daná funkcia nieje chránená žiadnym zámkom. Tento dopyt je zapísaný pomocou notácie Microsoft LINQ.

Ak tento vzor aplikujeme na príklad z obrázku 4, tak navrhnutý systém označí funkciu `service` značkou "label". V kroku 2 modul refaktoringu zistí, že funkcia ozna-

čená značkou "label"má byť premenovaná na `service_imp` a namiesto nej má byť vytvorená funkcia `service` obsahujúca kód zabezpečujúci vzájomnú výlučnosť. V poslednom kroku je zmenený názov funkcie `service` zmenený na pôvodný názov funkcie označenej značkou "label".

```
<?xml version="1.0" encoding="windows-1250"?>
<pattern name="Thread-Safe Interface" xmlns="http://www.w3.org">
    <entities>
        <class>
            <name>Class1</name>
            <attributes>
                <attribute>
                    <modifiers>
                        <modifier>private</modifier>
                        <modifier>readonly</modifier>
                    </modifiers>
                    <name>attribute1</name>
                    <type>object</type>
                </attribute>
            </attributes>
            <methods>
                <method label="label" originalName="method1">
                    <modifiers>
                        <modifier>private</modifier>
                    </modifiers>
                    <name>method_impl1</name>
                </method>
                <method>
                    <modifiers>
                        <modifier>public</modifier>
                    </modifiers>
                    <name>method1</name>
                    <code>
                    <![CDATA[lock(this) {this.method_impl1}]]>
                    </code>
                </method>
            </methods>
        </class>
    </entities>
    <relations></relations>
    <preconditions>
        <precondition label="label">
        <![CDATA[CodeStats.Types
            .SelectMany(type => type.Functions)
            .Where(func => ((func.Threads.Count > 1)
            && (GetLocks(func).Count() == 0));]]>
        </precondition>
    </preconditions>
</pattern>
```

Obr. 3: Príklad špecifikácie vzoru.

```
class client {
        server s = new server();
    void main(){
        thread t1 = new thread(() => thread_main);
        t1.start();
        thread t2 = new thread(() => thread_main);
        t2.start();

        thread.joinall(t1, t2);
    }
    void thread_main(){
        s.service();
    }
} // end client
class server {
    int x = 0;
    void service() {
        x++;
    }
} // end server
```

Obr. 4: príklad analyzovaného kódu.

5 Záver

Táto dizertačná práca predstavuje spôsob zápisu paralelných návrhových vzorov, ktorý umožňuje využitie tohoto zápisu pre automatické vkladanie návrhových vzorov do existujúcich zdrojových kódov. Existujú rôzne výskumy, ktoré sa venujú problematike automatického zavádzania návrhových vzorov do existujúcich zdrojových kódov, avšak žiaden z existujúcich výskumov sa nevenuje problematike zavádzania paralelných návrhových vzorov. Prvá časť práce prináša pohľad na použité témy analýzy kódu, návrhových vzorov a refaktoringu v kapitolách 2.1, 2.2, 2.3

Hlavným prínosom práce je návrh spôsobu zápisu návrhových vzorov umožňujúci asistované vkladanie paralelných návrhových vzorov do existujúcich paralelných zdrojových kódov, ktorý je vytvorený pomocou kombinácie existujúcich techník a metodík. Na to je potrebné byť schopný pracovať s návrhovým vzorom automaticky, čo znamená že návrhový vzor musí byť definovaný formálne. Druhá požiadavka je, že návrhový vzor musí obsahovať špecifikáciu miesta, kam sa má vložiť návrhový vzor. Tretia požiadavka je, že špecifikácia návrhového vzoru musí umožňovať jeho automatické vkladanie do existujúcich zdrojových kódov. Na základe týchto požiadaviek definujme návrhový vzor ako dvojicu (P, d) kde P je množina preconditions, určujúca vhodné umiestnenie vzoru a d je samotný popis vzoru. Množina P vychádza z analýzy kódu, ktorá umožňuje vytvárať dopyty nad existujúcim zdrojovým kódom, ktorých výsledok je miesto v zdrojovom kóde. Špecifikácia vzoru definuje jeho štruktúru a správanie. Navrhnutý spôsob zápisu je formálne popísaný v kapitole 3.

Na základe dát získaných z analýzy (ktorá je popísaná v kapitole 3.1), je možné definovať miesta v kóde s nesprávnou synchronizáciou medzi vláknami, prípadne s nesynchronizovaným prístupom ku zdielaným premenným. Tieto dva prípady sú najčastejšími dôvodmi na vloženie návrhového vzoru a preto sú popísané v tejto práci. Samozrejme existujú aj iné dôvody na vloženie paralelných návrhových vzorov do existujúcich zdrojových kódov, ale špecifikácia príslušných anti-vzorov sa dá získať jednoduchým rozšírením vyššie popísaného algoritmu. Špecfikáciou miesta v kóde na ktoré by sa mal aplikovať návrhový vzor je $p \in \mathbb{P}$ ktoré vieme jednoznačne definovať pomocou výrokov predikátovej logiky nad reláciami cg, pt, ta a la.

Špecifikácia vzoru využíva údaje zistené počas analýzy na definíciu miesta kam sa má vložiť vzor. Na to, aby bolo možné vložiť vzor je potrebný minimálne jeden bod v kóde. Týmto bodom sa rozumie miesto s nevhodným kódom. K definícii štruktúry sú využité prvky predikátovej logiky 1. rádu, najmä premenné, logické spojky, existenčný kvantifikátor a predikáty. Premenné reprezentujú primárne entity, zatiaľ čo predikáty reprezentujú trvalé relácie medzi nimi. U niektorých návrhových vzorov nestačí samotná definícia ich štruktúry, ale tiež je veľmi dôležitá definícia ich správania, tj. popis, ako účastníci vzoru spolu spolupracujú. Použitý jazyk opisuje správanie vzoru pomocou podmnožiny temporálnej logiky akcie. Jazyk opisuje správanie vzoru ako nekonečnú sekvenciu stavov, ktorými entity vzoru prechádzajú.

Spôsob zápisu paralelných návrhových vzorov bol použitý na návrh systému na vkladanie návrhových vzorov do existujúcich paralelných zdrojových kódov ktorý je popísaný v kapitole 4. Navrhnutý systém sa skladá z troch častí. Prvou časťou je analyzátor existujúcich kódov, ktorý pripraví informácie o zdrojových kódoch, vrátane informácie o vláknach a zámkoch. Druhou časťou je modul vyhodnocovania kvality

analyzovaných kódov, ktorý dokáže navrhnúť vhodné paralelné návrhové vzory. Treťou časťou je modul refaktoringu, ktorý sa postará o zavedenie návrhového vzoru do existujúcich kódov bez zmeny ich funkčnosti.

Navrhnutý systém bol experimentálne overený na vybranej množine paralelných návrhových vzorov zo zbierky POSA [14], ktoré sa zaoberajú súbežným behom a synchronizáciou súbežne bežiacich vlákien v programe. Navrhnutý systém je možné zlepšovať, napríklad podrobnejším štúdiom a implementáciou rôznych metód refaktoringu, alebo použitím iných metód analýzy zdrojových kódov. Taktiež je možné preštudovať možné rozšírenia navrhnutého spôsobu zápisu paralelných návrhových vzorov.

Využitie navrhnutého spôsobu zápisu je v oblasti bezpečnostných štandardov, najmä v oblasti letectva, zdravotníckej a vojenskej techniky. Navrhnutý spôsob zápisu je možné použiť aj k zvyšovaniu kvality existujúcich zdrojových kódov, keďže zdrojové kódy založené na návrhových vzoroch sú prehľadnejšie a jednoduchšie spravovateľné.

Tématika spoľahlivosti a bezpečnosti je hlavou témou autorových publikácií a ostatných vedeckých výstupov. Opisovaný spôsob zápisu paralelných návrhových vzorov a návrh systému, ktorý ho využíva, boli prezentované v publikáciach na konferenciách a vedeckých časopisoch: [Pub1, Pub2, Pub3, Pub4]. Využitie paralelných algoritmov na útoky silou bolo prezentované v časopise [Pub7]. Téma spoľahlivosti je spoločnou témou aj dvoch zverejnených úžitkových vzorov [Pat1, Pat2].

Literatúra

[1] checkstyle project page. 2016.
URL `http://checkstyle.sourceforge.net/`

[2] Jlint project page. 2016.
URL `https://sourceforge.net/projects/jlint/`

[3] pmd project page. 2016.
URL `https://pmd.github.io/`

[4] Administration, F. A.: Advisory Circular 20-115B. `http://goo.gl/C6d1k`, 1993 [cit. 2014-08-23].

[5] Administration, F. D.: Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices. `http://goo.gl/JqkYr`, 2005-05-11 [cit. 2014-08-23].

[6] Alexander, C.: *The Timeless Way of Building*. číslo zv. 8 in Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series, Oxford University Press, 1979, ISBN 9780195024029.
URL `https://books.google.cz/books?id=H6CE9hlbO8sC`

[7] Alexander, C.; Ishikawa, S.; Silverstein, M.: *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series, OUP USA, 1977, ISBN 9780195019193.
URL `https://books.google.cz/books?id=hwAHmktpk5IC`

[8] Ball, T.: The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, London, UK, UK: Springer-Verlag, 1999, ISBN 3-540-66538-2, s. 216–234.
URL http://dl.acm.org/citation.cfm?id=318773.318944

[9] Bayley, I.: Formalising Design Patterns in Predicate Logic. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2884-8, s. 25–36, doi:10.1109/SEFM.2007.22.
URL http://dx.doi.org/10.1109/SEFM.2007.22

[10] Bayley, I.: Formalising Design Patterns in Predicate Logic. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2884-8, s. 25–36, doi:10.1109/SEFM.2007.22.
URL http://dx.doi.org/10.1109/SEFM.2007.22

[11] Bayley, I.; Zhu, H.: On the Composition of Design Patterns. In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, Washington, DC, USA: IEEE Computer Society, 2008, ISBN 978-0-7695-3312-4, s. 27–36, doi:10.1109/QSIC.2008.32.
URL http://dx.doi.org/10.1109/QSIC.2008.32

[12] Bayley, I.; Zhu, H.: Specifying Behavioural Features of Design Patterns in First Order Logic. In *Proceedings of the 2008 32Nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '08, Washington, DC, USA: IEEE Computer Society, 2008, ISBN 978-0-7695-3262-2, s. 203–210, doi:10.1109/COMPSAC.2008.67.
URL http://dx.doi.org/10.1109/COMPSAC.2008.67

[13] Bessey, A.; Block, K.; Chelf, B.; aj.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, ročník 53, č. 2, Únor 2010: s. 66–75, ISSN 0001-0782, doi:10.1145/1646353.1646374.
URL http://doi.acm.org/10.1145/1646353.1646374

[14] Buschmann, F.; Henney, K.; Schmidt, D.: *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007, ISBN 0470059028, 9780470059029.

[15] C., H.: DO-178B safety certification and other software security tools drive avionics software designs. http://goo.gl/Z8zniy, 2011-05-19 [cit. 2014-08-23].

[16] Fleury, E.; Point, G.; Vincent, A.: Binary Program Analysis: Theory and Practice.
URL http://www-verimag.imag.fr/ async/CCIS/talk 13/Fleury.pdf

[17] Food; Administration, D.: General Principles of Software Validation; Final Guidance for Industry and FDA Staff. http://goo.gl/HjIKb, 2002-01-11 [cit. 2014-08-23].

[18] Fowler, M.: Refactoring. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, TOOLS '00, Washington, DC, USA: IEEE Computer Society, 2000, ISBN 0-7695-0774-3, s. 437–.
URL `http://dl.acm.org/citation.cfm?id=832261.833315`

[19] Fowler, M.; Beck, K.: *Refactoring: Improving the Design of Existing Code*. Component software series, Addison-Wesley, 1999, ISBN 9780201485677.
URL `https://books.google.cz/books?id=1MsETFPD3I0C`

[20] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994, ISBN 9780321700698.
URL `https://books.google.cz/books?id=6oHuKQe3TjQC`

[21] Gosain, A.; Sharma, G.: *Static Analysis: A Survey of Techniques and Tools*. New Delhi: Springer India, 2015, ISBN 978-81-322-2268-2, s. 581–591.

[22] Hovemeyer, D.; Pugh, W.: Finding Concurrency Bugs in Java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[23] Johnson, R. E.; Opdyke, W. F.: Refactoring and Aggregation. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, London, UK, UK: Springer-Verlag, 1993, ISBN 3-540-57342-9, s. 264–278.
URL `http://dl.acm.org/citation.cfm?id=646897.709889`

[24] McConnell, S.: *Code Complete*. DV-Professional Series, Microsoft Press, 2004, ISBN 9780735619678.
URL `https://books.google.cz/books?id=QnghAQAAIAAJ`

[25] Opdyke, W. F.: *Refactoring Object-oriented Frameworks*. Dizertační práce, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[26] Opdyke, W. F.: Refactoring Object-oriented Frameworks. Technická zpráva, Champaign, IL, USA, 1992.

[27] Pun, W.; Winder, R.: Automating Class Hierarchy Graph Construction.

[28] Taibi, T.; Ngo, D. C. L.: Formal Specification of Design Patterns - A Balanced Approach. *Journal of Object Technology*, ročník 2, č. 4, Červenec 2003: s. 127–140, ISSN 1660-1769, doi:10.5381/jot.2003.2.4.a4.

[29] Zhu, H.; Bayley, I.; Shan, L.; aj.: Tool Support for Design Pattern Recognition at Model Level. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, COMPSAC '09, Washington, DC, USA: IEEE Computer Society, 2009, ISBN 978-0-7695-3726-9, s. 228–233, doi:10.1109/COMPSAC.2009.37.
URL `http://dx.doi.org/10.1109/COMPSAC.2009.37`

Životopis

Osobné údaje

Meno:	Peter Jurnečka
Národnosť	slovenská
Dátum narodenia	17. marca 1986
E-mail:	`ijurnecka@fit.vutbr.cz`
Web:	`www.fit.vutbr.cz/ ijurnecka`
Linkedin:	`https://www.linkedin.com/in/peter-jurnecka`

Vzdelanie

od 2009	Fakulta informačních technologií VUT v Brně, doktorský študijny program Výpočetní technika a informatika
2007 – 2009	Fakulta informačních technologií VUT v Brně, magisterský študijny program Informační technologie, DP: Analyzátor protokolov riadený pravidlami
2004 – 2007	Fakulta informatiky a informačných technologií STU v Bratislave, bakalársky študijny program Informatika
2000 – 2004	Gymnázium Jána Bosca, Nová Dubnica, Slovensko

Publikácie

5 publikácie v DBLP
7 publikácií v Scopus
1 publikácia v recenzovanom neimpaktovanom priodiku (DSM 1211-8737)
1 publikácia v recenzovanom impaktovanom priodiku. IF: 0,872 (Springer 1863-1703)
5 publikácií na medzinárodnej konferencii
8 citácií

Produkty

AnalyzeThis: Analyzátor protokolů řízený pravidly„ software, 2010

Zapísané úžitkové vzory

Zariadenie na zvýšenie spoľahlivosti (MTBF) elektrotechnických výrobkov. 2011. Slovensko. 126-2010, MPT:G01R 13/28. Přihlášeno 07.09.2010. Zapsáno 29.09.2011.

Zariadenie na meranie a vyhodnocovanie obsahu vyšších harmonických v napätiach, resp. prúdoch elektrických zariadení. Slovensko. 169-2010, MPT:G01R 23/16. Přihlášeno 15.11.2010. Zapsáno 21.10.2011

Projekty

Pokročilé bezpečné, spolehlivé a adaptivní IT, VUT v Brně, FIT-S-11-1, 2011-2013
Matematické a inženýrské metody pro vývoj spolehlivých a bezpečných paralelních a distribuovaných počítačových systémů, GAČR - Doktorské granty, GD102/09/H042, 2009-2012

Abstrakt

V tejto práci je opísaný návrh spôsobu zápisu a práce s paralelnými návrhovými vzormi, ktorého prínosom je možnosť navrhovania automatických oprav existujúcich paralelných zdrojových kódov pomocou refaktoringu. Na to, aby bolo možné navrhovaný spôsob zápisu využiť je potrebné, aby táto práca pokrývala oblasti statickej analýzy kódu, formálneho zápisu paralelných návrhových vzorov a refaktoringu. Statická analýza kódu umožňuje porozumieť existujúcim paralelným zdrojovým kódom a definovať miesta, kam sa má vložiť návrhový vzor. Formálny zápis návrhového vzoru umožňuje automaticky aplikovať daný vzor do existujúceho zdrojového kódu. Nakoniec refaktoring umožňuje upraviť existujúci zdrojový kód bez zmeny funkčnosti. Prvá časť práce sa venuje popisu súčasného stavu v týchto troch oblastiach t.j. analýze kódu, návrhovým vzorom a refaktoringu. Druhá časť práce sa venuje opisu metodiky a experimentálnemu overeniu jej nasadenia.

Abstract

This Ph.D. thesis describes proposed notation and method for working with parallel design patterns, which allowes proposing of automatic corrections to existing parallel source code with help of refactoring. In order to define the proposed notation, this work must cover areas of static code analysis, formal description of parallel design patterns and refactoring. Static code analysis is used to analyse the existing parallel source code for definition of places where you want to insert specified design pattern. Formal description of design pattern allows you to automatically apply the pattern to the existing source code. Finally, refactoring allows you to edit an existing source code without changing its functionality. The first part is devoted to the description of the current status in these three areas e.g. code analysis, design patterns and refactoring. The second part is devoted to a description of the methodology and experimental verification of its deployment.