VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘESNÉ HEURISTIKY PRO VKLÁDÁNÍ ŠUMU V NÁSTROJI SEARCHBESTIE

FINE-GRAIN NOISE-INJECTION HEURISTICS FOR SEARCHBESTIE INFRASTRUCTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DAVID KOZÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. BOHUSLAV KŘENA, Ph.D.
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19986/2016/xkozak15

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Kozák David**
Obor: Informační technologie
Téma: **Přesné heuristiky pro vkládání šumu v nástroji SearchBestie**
Fine-Grain Noise-Injection Heuristics for SearchBestie Infrastructure
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s technikami testování vícevláknových programů s využitím vkládání šumu. Detailně pak prostudujte nástroje SearchBestie a RoadRunner.
2. V návaznosti na projektovou praxi dokončete začlenění nástroje RoadRunner pro monitorování Java programů do infrastruktury SearchBestie.
3. Navrhněte nové heuristiky pro vkládání šumu, které by umožnily vkládat šum na přesně definovaná místa v kódu.
4. Navržené heuristiky implementujte a na vhodných příkladech ověřte jejich přínos pro testování vícevláknových aplikací.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího vývoje infrastruktury SearchBestie.

Literatura:

- KŘENA Bohuslav, LETKO Zdeněk a VOJNAR Tomáš. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. *Lecture Notes in Computer Science*. 2012, roč. 2012, č. 7186, s. 177-192. ISSN 0302-9743.
- FIEDOR Jan, DUDKA Vendula, KŘENA Bohuslav, LETKO Zdeněk, UR Shmuel a VOJNAR Tomáš. Advances in Noise-based Testing of Concurrent Programs. *Software Testing, Verification and Reliability*. 2015, roč. 25, č. 3, s. 272-309. ISSN 1099-1689.

Pro udělení zápočtu za první semestr je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křena Bohuslav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se zabývá efektivními způsoby testování vícevláknových programů psaných v jazyce Java. Pro zvýšení šance na odhalení časově závislých chyb se využívá techniky vkládání šumu, kdy dochází ke vložení dalších instrukcí do testované aplikace za účelem prozkoumání netypických proložení událostí. Pro nalezení vhodných konfigurací pro vkládání šumu byl navržen nástroj SearchBestie, který využíval nástroj ConTest pro spouštění testů a instrumentaci bytecodu. Jelikož vývoj ConTestu byl zastaven, bylo třeba najít vhodnou alternativu. V průběhu této bakalářské práce bylo dokončeno propojení SearchBestie s nástrojem RoadRunner, který ConTest nahradil. Dále došlo k navržení a implementaci nových heuristik, které umožňují šum vkládat na přesně zvolená místa v kódu. Experimenty prokázaly, že ve většině případů skutečně přesné heuristiky dosahují lepších výsledků než heuristiky využívající náhodnosti.

Abstract

The main topic of this thesis is testing of concurrent Java programs. Concurrency errors often manifest rarely. To increase a chance to spot such errors, a technique called noise injection can be used. This technique inserts extra instruction into the application under test to disturb the scheduler and thus to explore less common thread scheduling. Because noise injection can be parameterized in many ways, a tool called SearchBestie was created to handle noise-based testing as a search problem. SearchBestie used a tool called Contest for bytecode instrumentation and test execution. However, the development of ConTest has been discontinued and therefore an alternative tool was necessary. In the first part of this thesis, SearchBestie was connected with a tool called RoadRunner, which replaced ConTest. Afterwards new fine-grain noise-injection heuristics have been proposed and implemented. Experiments proved that these heuristics achieve better results in most cases.

Klíčová slova

verifikace softwaru, testování, vícevláknové programování, Java, chyby ve vícevláknových programech, vkládání šumu, metriky pro testování

Keywords

software verification, testing, concurrent programming, Java, concurrency errors, noise injection, metrics for testing

Citace

KOZÁK, David. *Přesné heuristiky pro vkládání šumu v nástroji SearchBestie.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Bohuslav Křena, Ph.D.

Přesné heuristiky pro vkládání šumu v nástroji SearchBestie

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Bohuslava Křeny, Ph.D. Další informace mi poskytli členové skupiny VeriFIT a původní autor nástroje SearchBestie, Ing. Zdeněk Letko, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
David Kozák
30. července 2017

Poděkování

Chtěl bych poděkovat svému školiteli Ing. Bohuslavu Křenovi, Ph.D., za velkou míru trpělivosti a podpory, kterých se mi od něj dostalo. Dále bych chtěl poděkovat Mgr. Bc. Haně Pluháčkové ze skupiny VeriFIT, jejíž pomoc pro mě byla též k nezaplacení. Velké díky patří také původnímu autorovi nástroje SearchBestie, Ing. Zdeňku Letkovi, Ph.D., za čas, který mi poskytl při seznamování se s nástrojem. Na závěr bych chtěl také poděkovat své rodině za podporu a lásku, které mi celý život projevují.

Obsah

1 Úvod **3**

2 Testování a analýza vícevláknových programů v jazyce Java **5**
2.1 Prostředky pro tvorbu vícevláknových aplikací v programovacím jazyku Java 5
2.1.1 Java memory model 6
2.2 Přehled způsobů verifikace softwaru 7
2.3 Přehled typických chyb objevujících se ve vícevkláknových programech . . . 8
2.3.1 Chyby v bezpečnosti 9
2.3.2 Chyby v živosti 11
2.4 Metriky pro testování vícevláknových aplikací 13
2.5 Heuristiky pro testování vícevláknových aplikací 15
2.5.1 Noise seeding heuristiky 16
2.5.2 Noise placement heuristiky 16
2.6 Abstraktní identifikátory pro dynamickou analýzu vícevláknových programů 17
2.6.1 Místo v programu 17
2.6.2 Identifikace vláken 17
2.6.3 Identifikace objektů 18
2.7 Testování jako optimalizační problém 18
2.7.1 Metaheuristické přístupy k prohledávání stavového prostoru 18

3 Nástroje podporující testování vícevláknových aplikací **20**
3.1 SearchBestie 20
3.2 RoadRunner 22
3.3 ConTest 23
3.4 Coverage lib 24

4 Propojení SearchBestie s RoadRunnerem s využitím Coverage lib **25**
4.1 Návrh propojení 25
4.2 Příprava SearchBestie pro komunikaci s RoadRunnerem 26
4.3 Úpravy Coverage lib 27
4.4 Úpravy a rozšíření RoadRunneru 28
4.5 Srovnávací testy RoadRunneru a ConTestu 32

5 Návrh a implementace nových heuristik **35**
5.1 Nově navržené noise placement heuristiky 35
5.2 Návrh implementace 35
5.3 Úpravy v nástroji RoadRunner 36
5.3.1 Sběr informací o instrumentovaných třídách 36

5.3.2 Implementace přesnějšího vkládání šumu 38
5.4 Úpravy v nástroji SearchBestie 38
5.5 Testování nových heuristik 38

6 Závěr **49**

Literatura **51**

A Obsah přiloženého CD **52**

B Uživatelská příručka **54**
B.1 Překlad a spuštění testů 54
B.2 Analýza výstupních souborů 55

Kapitola 1

Úvod

Dlouhodobý trend zvyšování počtu jader v procesorech a jejich snižující se cena již dospěly do bodu, kdy se problematika vícevláknového programování stala velice aktuálním tématem. Nutnost umět programovat paralelně již patří k nezbytným dovednostem každého programátora. Ruku v ruce s výhodami, které vícevláknové programování přináší, přichází ale i nová skupina chyb, které se objevují pouze za velmi specifických okolností, a jsou proto obtížně odhalitelné. Klasifikace různých druhů chyb ve vícevláknových programech a algoritmy pro jejich detekci se proto staly předmětem intenzivního výzkumu. V rámci skupiny VeriFIT na Fakultě informačních technologií postupně vznikaly různé nástroje specializované na testování vícevláknových programů. Pro jazyk C/C++ byla vytvořena Anaconda a pro jazyk Java SearchBestie. Tato práce se zabývá právě rozvojem infrastruktury SearchBestie, jejímž původním autorem je Ing. Zdeněk Letko, Ph.D., který nástroj vyvinul v rámci svého doktorského studia zde na fakultě.

SearchBestie se dá považovat za výsledný produkt mnohaletého studia oblasti testování vícevláknových programů v Javě. Pojďme se nyní zaměřit na jednotlivé kroky, které k jejímu vzniku vedly. Jelikož typickou vlastností chyb ve vícevláknových programech je jejich zřídkavá manifestace, první stádia výzkumu se zaměřovala právě na zvyšování šance, že se chyba projeví. K tomuto účelu byla využita technika vkládání *šumu*, která ovlivňuje činnost plánovače operačního systému s cílem dosáhnout vzácnějších proložení událostí, ve kterých existuje vyšší šance odhalení chyby [7]. VeriFIT začal spolupracovat s firmou IBM, jejíž nástroj ConTest se pro instrumentaci a spouštění testovaných programů ve skupině aktivně využíval. Technika vkládání šumu se též ukázala jako účinný způsob *léčení* dříve odhalených chyb, nicméně vzhledem k problematičnosti nasazení této techniky v praxi[1] se výzkum začal ubírat jiným směrem. S postupem času se experimentovalo s různými heuristikami pro vkládání šumu a množství možných konfigurací spouštěných testů rapidně narůstalo. Tento fakt vzbudil přirozenou zvědavost, zda proces hledání vhodných testových konfigurací lze definovat jako automatizovaně řešený optimalizační problém. Od úvah se přešlo k návrhu a implementaci, jejímž výstupem se stal právě nástroj SearchBestie.

Dr. Letko naneštěstí několik let po získání svého doktorského titulu akademickou půdu opustil. Další problém nastal, když se vývoj ConTestu roku 2011 zastavil, což časem způsobilo nekompatibilnost s novými verzemi Javy. Pro řešení vzniklé situace byla vypsána projektová praxe zaměřující se na údržbu a další rozvoj SearchBestie. Jako hlavní cíl se stanovilo nahradit v dané chvíli již zastaralý nástroj ConTest vhodnou alternativou, za kterou

[1] Technika narazila na problémy v oblasti práva. Otázka, kdo by případně zodpovídal za nesprávně vyléčený program, se jen velmi těžko zodpovídala.

byl zvolen RoadRunner. Tento úkol se ovšem ukázal být z časového hlediska velice náročný a v rámci projektové praxe se naneštěstí nestihl dokončit. Finální úpravy a následné srovnání ConTestu a RoadRunneru se tedy staly prvním důležitým úkolem v rámci bakalářské práce.

Výše zmíněný výzkum na fakultě se mimo jiné zabýval i různými metaheuristickými přístupy k prohledávání stavového prostoru, které by umožňovaly najít optimální testové konfigurace bez nutnosti prohledávat celý stavový prostor. Praktické experimenty skutečně prokázaly, že nasazení genetických algoritmů může při zvolení vhodné fitness funkce přinést oproti náhodnému prohledávání lepší výsledky [7]. Nicméně tyto výsledky bohužel nebyly platné pro všechny testované programy a také trpěly jinými neduhy, například větší časovou náročností. Zároveň s tímto výzkumem bylo zjištěno, že soustředění šumu na jednu určitou proměnnou může oproti náhodnému vkládání šumu přinést výrazně lepší výsledky [7]. Zvážení dvou výše uvedených faktů vedlo ke stanovení druhého hlavního cíle bakalářské práce. Tím se stal návrh a implementace nových heuristik, které by umožňovaly přesně určit, na která místa v programu se šum vloží. Rozhodování se ponechalo v režii SearchBestie. S tím spojené výrazné zvětšení stavového prostoru způsobilo, že genetické algoritmy mohly skutečně projevit svůj potenciál.

Text práce je strukturován následovně. Kapitola 2 se zabývá problematikou tvorby, testování a dynamické analýzy programů psaných v jazyce Java se zaměřením na chyby v paralelismu. V kapitole 3 jsou popsány nástroje, které byly v rámci práce používány a ve většině případů i rozvíjeny. Jmenovitě se jedná a ConTest, RoadRunner, SearchBestie a Coverage lib. Kapitola 4 se zabývá implementací propojení SearchBestie s RoadRunnerem a v závěru poskytuje experimentální srovnání ConTestu a RoadRunneru na sérii testovaných programů. V kapitole 5 je popsán návrh a postup implementace mechanismů pro přesné vkládání šumu. Efektivita těchto nových heuristik je opět ověřena na sérii testovaných programů. V kapitole 6 jsou shrnuty výsledky a uvedeny návrhy na směr budoucího výzkumu.

Kapitola 2

Testování a analýza vícevláknových programů v jazyce Java

Tato kapitola poskytuje přehled znalostí o testování a analýze vícevláknových programů v Javě. Jelikož pro pochopení a odhalování chyb ve vícevláknových programech je důležité rozumět mechanismům, které daný jazyk nabízí, v sekci 2.1 naleznete popis standardního rozhraní pro tvorbu vícevláknových programů v Javě a také nízkoúrovňový Java Memory Model jakožto základní prostředek pro budování složitějších synchronizačních mechanismů a ověřování správnosti programů. V sekci 2.2 naleznete stručný přehled různých způsobů verifikace softwaru a jejich možnosti nasazení v oblasti testování vícevláknových programů. Sekce 2.3 klasifikuje různé chyby objevující se ve vícevláknových programech. V sekci 2.4 naleznete přehled metrik a v sekci 2.5 naleznete přehled heuristik využívaných při testování vícevláknových programů. Sekce 2.6 informuje o metodách pro identifikaci míst ve zdrojovém programu, objektů a vláken jakožto důležitý stavební kámen nutný pro efektivní analýzu pokrytí při testování programů. Sekce 2.7 poskytuje informace o metodách využívaných při hledání vhodných testových konfigurací s využitím optimalizace.

2.1 Prostředky pro tvorbu vícevláknových aplikací v programovacím jazyku Java

Java je populární objektově orientovaný jazyk. Programy v něm psané se překládají do bytecodu[1], který je následně interpretován s využitím běhového prostředí pojmenovaného *Java Virtual Machine* [7]. Při startu programu v Javě je vytvořeno jedno hlavní vlákno, které spouští metodu *main* a několik pomocných vláken, které zajišťují například funkci *garbage collectoru*. Z tohoto faktu vyplývá, že všechny programy v Javě jsou vícevláknové [7]. Pro reprezentace vláken se využívá objektů třídy Thread. Tato třída poskytuje komunikační rozhraní pro práci s vlákny. Nejdůležitějšími metodami jsou *start()*, které spustí vlákno, *join(thread)*, které umožňuje čekat na ukončení běhu vlákna *thread*, *sleep(time)*, které vlákno uspí na daný časový interval, *interrupt()*, které zašle vláknu signál požadující jeho ukončení[2] a metoda *yield()*, jíž se vlákno vzdává procesoru vyvoláním přepnutí kontextu [7].

[1] Jméno bytecodu je odvozeno od faktu, že operační kódy všech instrukcí mají délku jeden byte.

[2] I přesto, že toto pravidlo není v rámci API třídy Thread naimplementováno, metoda Thread::interrupt by měla být využívána pouze na zaslání informace o tom, že se má dané vlákno při nejbližší vhodné příležitosti ukončit. Nicméně vzhledem k faktu, že jazyk sám to nevynucuje, rozhodnutí, jak danou situaci ošetřit, je čistě na programátorovi. V praxi tedy k ukončení vůbec dojít nemusí. [5]

Objekty v Javě disponují monitorem s implicitním zámkem, který může být použit pro účely synchronizace [5]. K tomuto monitoru lze přistoupit s využitím klíčového slova *synchronized*, a to dvěma odlišnými způsoby. Prvním z nich je konstrukce *synchronized(o) {...}*, kde *o* označuje objekt, k jehož monitoru se bude přistupovat. Následující blok kódu bude proveden až po získání daného zámku. Druhá alternativa je označit slovem synchronized celou metodu. V praxi se spíše doporučuje používat první variantu, která označuje pouze blok kódu. Druhá varianta je sice jednodušší, nicméně její využití často *kritickou sekci* chráněnou zámkem činí větší, než je skutečně nutné, což má negativní dopad na výkon programu. Technice využívání implicitního monitoru se také říká implicitní zamykání [5].

Balík *java.util.concurrent* nabízí mimo mnoha dalších užitečných tříd i explicitní zámek. Své využití nalezne v situacích, kdy je potřeba operace obsazení a uvolnění provádět v různých metodách. Součástí tohoto balíku jsou také *thread-safe*[3] kolekce, jejichž využití může značně ulehčit vývoj vícevláknových aplikací.

K synchronizaci lze využít i klíčové slovo *volatile*. Čtení a zápis takto označeného atributu jsou vždy atomické operace a navíc je i zaručena viditelnost změn hodnoty této proměnné mezi vlákny [7]. Své využití tato konstrukce nalezne například pro předání signálu z jednoho vlákna do jiného. Volatilní proměnné jsou často chybně využívány při *check-then-act* či *read-then-write* scénářích. Jednotlivé přístupy k nim sice atomické jsou, nicméně pro tvorbu většího atomického bloku zahrnujícího dva a více přístupů již programátor potřebuje využít jiných synchronizačních prostředků.

2.1.1 Java memory model

Pro pochopení synchronizace na nejnižší úrovni[4] je potřeba porozumět základním mechanismům, které platforma programátorům poskytuje. *Java memory model* definuje vztahy mezi akcemi ve vícevláknovém programu, které jsou vždy platné. Tyto vztahy lze využít k ověřování správnosti programů a také k tvorbě abstraktnějších synchronizačních objektů jako například výše uvedené kolekce v java.util.concurrent. Vztahy v memory modelu Javy jsou reprezentovány *Happens-before* relací, což je částečné uspořádání nad událostmi v programu [5]. Pravidla happens-before jsou zobrazena v seznamu uvedeném níže. Výraz happens before byl v textu přeložen jako *stane se před*.

Pravidla Happens-before

- Každá akce v daném vlákně se stane před všemi událostmi, které následují po ní později.
- Odemknutí zámku na monitoru se stane před jakoukoliv následující operací zamknutí na tomtéž monitoru.
- Zápis do volatilní proměnné se stane před jakýmkoliv následujícím čtením stejné proměnné.
- Volání metody *Thread::start*[5] se stane před jakoukoliv akcí vykonanou spouštěným vláknem.

[3] Pojem thread-safe se užívá k označení objektů, ke kterým lze přistupovat z více vláken bez nutnosti žádné další synchronizace [5].

[4] Tedy z hlediska programovacího jazyka Java.

[5] Pro zjednodušení zápisu metody je využita syntaxe nově dostupná od Javy 8. Jedná se o takzvanou *referenci na metodu*, jejíž formát je {objekt,třída}::metoda.

- Každá akce vykonaná v daném vlákně *t1* se stane předtím, než jiné vlákno *t2* detekuje, že *t1* skončilo. Detekcí je myšleno úspěšné ukončení metody *Thread::join* či volání *Thread::isAlive* vracející hodnotu *false*.
- Volání metody *Thread::interrupt* se stane předtím, než dané vlákno zjistí, že bylo přerušeno.
- Ukončení konstruktoru objektu se stane před spuštěním destruktoru téhož objektu.
- Tranzitivita. Pokud se událost A stala před událostí B a B se stala před událostí C, pak se A stala před C.

2.2 Přehled způsobů verifikace softwaru

V této sekci se nachází stručný popis jednotlivých metod verifikace softwaru a také možností jejich uplatnění při testování vícevláknových aplikací.

Testování

Testování je nejčastější způsob odhalování chyb v programech. Tato metoda se zakládá na následujícím principu. Programátor či tester nejdříve vytvoří *testovací případ*, který je definován svými vstupy a očekávanými výstupy. Pokud při spuštění testu skutečné výstupy neodpovídají očekávaným, tak se buď v programu nebo v testovacím případu nachází chyba. Testování bývá často kombinováno s *analýzou pokrytí*, což je proces sbírání a analýzy jednotlivých *metrik pokrytí*. Každá taková metrika je založena na určité vlastnosti testovaného programu, která se z pohledu testování jeví zajímavá. Jako příklad lze uvést dostupnost určitého řádku či pozorování různých typů interakce mezi vlákny. Každá takto zaznamenaná událost je reprezentována jedním *artefaktem pokrytí*. Task v tomto kontextu reprezentuje vektor hodnot typu *celé číslo* či *řetězec*, jehož konkrétní podoba je dána metrikou. Tento vektor reprezentuje určitou událost v běhu programu. Množina všech artefaktů pokrytí dané metriky se nazývá *doména pokrytí* [7]. Na základě počtu artefaktů lze určit, jak dobře byl program otestován. Metriky pro testování vícevláknových programů jsou podrobněji rozebrány v sekci 2.4.

Z pohledu verifikace vícevláknových programů trpí testování jedním nedostatkem. Vzhledem k nedeterminismu, který je spojen s během vícevláknového programu, totiž jedno spuštění testu s danými vstupy nestačí pro ověření, zda pro ně program pracuje správně [7]. Jelikož se chyby při vícevláknovém programování často objevují pouze za výjimečných okolností, existují techniky snažící se šanci na odhalení chyby zvýšit. Jedna z nich je *systematické testovaní*. Tato technika ovlivňuje plánovač operačního systému takovým způsobem, aby při spouštění testu byla prozkoumána postupně všechna legální proložení mezi vlákny [7]. Nicméně tato technika naráží na problém vysoké výpočetní náročnosti. Jako méně náročná alternativa k tomuto postupu se dá využít vkládání *šumu*. Jedná se o techniku, kdy jsou do testovaného programu na místa specifikovaná buď náhodně či na základě nějaké heuristiky vkládány instrukce ovlivňující plánovač. Tato technika sice umožňuje na rozdíl od systematického testování prozkoumat najednou pouze jedno proložení událostí, nicméně praktické testy ukázaly, že pro odhalování běžných chyb představuje dobrý kompromis mezi vysokou mírou kontroly a výpočetní náročností [7].

Statická analýza

Statická analýza probíhá již v době překladu, a proto často ani nepotřebuje, aby byl zdrojový text spustitelný [7]. Nástrojů statické analýzy existuje mnoho, některé se zaměřují na detekci určitých vzorů v kódu, které jsou považovány za špatné programátorské praktiky, jiné jsou mnohem komplexnějšího charakteru. Do této kategorie lze též zařadit i metody *formální verifikace* jako například *theorem proving* či *model checking* [7].

Výhoda metod formální verifikace tkví v tom, že teoreticky umožňují odhalit všechny chyby v programu a tedy i dokázat jeho správnost. Nicméně jejich použití je již při testování sekvenčních programů vysoce výpočetně náročné a nedeterminismus ve vícevláknovém programování tento problém ještě zhoršuje. Tyto metody se tedy špatně škálují na programy větších rozměrů [7].

Typičtější metody statické analýzy jsou často navržené s využitím velmi vysoké míry abstrakce, která je činí škálovatelné i na programy velkých rozměrů. Nicméně tato vysoká úroveň abstrakce také způsobuje, že daná analýza produkuje velké množství hlášení o domnělých chybách, které ale v testovaném programu doopravdy neexistují, vznikly pouze díky abstrahování se od určitých aspektů testovaného programu.

Dynamická analýza

Dynamická analýza, někdy též označovaná jako *runtime verifikace*, je metoda postavená na sbírání dat o testovaném programu pozorováním jeho běhu. Nástroj poté data zpracovává a získává nové informace, které mohou mimo jiné pomoci i k odhalení chyb, ke kterým v daném běhu nedošlo [7]. Nástroje provádějící dynamickou analýzu většinou fungují následujícím způsobem. Do zkoumaného programu se vloží na úrovni zdrojového kódu, mezikódu, či binárního kódu volání funkcí z API[6] nástroje, ve kterých dochází k výše zmíněné analýze. Z jejich činnosti lze tedy lehce vydedukovat, že dynamické analyzátory určitou mírou zasahují do normálního běhu programu, čímž mohou ovlivnit jeho chování. Tato forma zatížení je označována jako *probe effect* [7].

Shrnutí

Jelikož metody statické analýzy stále trpí špatnou škálovatelností, pro verifikaci vícevláknových programů se využívá hlavně testování, často doplňované analýzou pokrytí a dynamickou analýzou [7]. Touto oblastí se zaobírá i tato bakalářská práce.

2.3 Přehled typických chyb objevujících se ve vícevláknových programech

Vícevláknové programování umožňuje efektivně využívat vícejádrové procesory, což vede k vývoji rychlejších a responzivnějších aplikací. Nicméně mnoho programátorů začne tvořit vícevláknové programy bez hlubšího studia problematiky. Chyby v takových programech se lehce vytvoří, ale jejich odhalení a odstranění může být z hlediska času i nutných znalostí velice náročné [5]. V této sekci jsou jednotlivé chyby rozebrány. Dají se rozdělit do dvou kategorií, *chyby v bezpečnosti* a *chyby v živosti*. Termín *chyba* bude nadále využíván ve významu chyba ve vícevláknovém programu.

[6] API - Application programming interface - rozhraní pro komunikaci programátora s aplikací.

2.3.1 Chyby v bezpečnosti

Tato kategorie zahrnuje chyby, které způsobují, že se v běhu programu stane něco špatného. Jejich odhalování je jednodušší než u chyb v živosti, protože mají konečný protipříklad [5], tedy konečnou posloupnost událostí, které vedou k projevení dané chyby.

Časově závislá chyba nad daty

Jedná se o nejčastější a také nejvíce zkoumanou kategorii chyb. K její identifikaci je potřeba vědět:

- Ke kterým proměnným se při běhu programu přistupuje.
- Jakým způsobem jsou tyto přístupy synchronizovány.

Definice 2.3.1. Běh programu obsahuje časově závislou chybu nad daty, jestliže při něm dojde ke 2 nesynchronizovaným přístupům ke sdílené proměnné a alespoň jeden z nich je zápis.

Příklad 2.3.1. Příklad časově závislé chyby nad daty:

```
int i = 0;

Vlákno 1:                    Vlákno 2:
  i++;                         i++;
```

Typická časově závislá chyba nad daty je uvedena v příkladu 2.3.1. Zde obě vlákna inkrementují hodnotu proměnné *i*. Protože tyto operace nejsou žádným způsobem synchronizované a obě typu čtení i zápis, dochází zde k časově závislé chybě nad daty[7]. Velmi nepříjemným faktem jest, že i jediný nesynchronizovaný přístup k jinak správně synchronizované proměnné vede ke vzniku časově závislé chyby nad daty [5].

Porušení atomicity

Definovat porušení atomicity je náročnější. Jedná se o abstraktnější druh chyby, ke které dojde v situaci, kdy vykonávání určité posloupnosti akcí, která byla zamýšlena jako atomický blok, naruší vlivem nedostatečné synchronizace činnost jiného vlákna. Pro jeho identifikaci je nutno specifikovat:

- Kdy lze dva běhy programu považovat za ekvivalentní.
- Které bloky kódu jsou považovány za atomické.

Definice 2.3.2. Běh programu porušuje atomicitu, pokud není ekvivalentní jakémukoliv jinému běhu, v němž jsou všechny atomické sekce vykonány sériově.

Mezi typické situace, kdy dochází k porušení atomicity, lze zařadit například scénář *check-then-act*. Jedná se o situaci, kdy program na základě podmínky vykoná určitou činnost, jejíž provedení má smysl, pouze pokud daná podmínka stále platí. V sekvenčním programu se

[7] Pozorný čtenář jistě zaznamená, že zde dochází také k porušení atomicity. Pro zajištění správného výsledku by tři dílčí kroky, tedy načtení hodnoty z paměti, zvětšení o 1 a uložení zpět, měly být provedeny atomicky.

jedná o triviální operaci, nicméně ve vícevláknovém prostředí je potřeba z dané části kódu udělat atomický blok. Jedním z klíčových problémů při detekci porušení atomicity je zjištění, jaké bloky kódu mají být atomické. Řešení existuje několik. Nejjednodušší cesta říká, že tuto informaci analyzátoru poskytne programátor buď ručně před spuštěním testu nebo ve formě anotací v testovaném programu. Druhá cesta prosazuje automatické odvozování atomických sekcí na základě informací o běhu programu. Dá se například předpokládat, že pokud ke třem proměnným bylo z jednoho místa v programu přistupováno v rámci atomické sekce, může jít o související proměnné, například souřadnice v trojrozměrném souřadném systému.

Příklad 2.3.2. Příklad porušení atomicity:
Map map;

Vlákno 1:

```
if(!map.contains("key1"))
  map.insert("key1","value1");
```

Vlákno 2:

```
map.insert("key1","value2");
```

V příkladu 2.3.2 naleznete porušení atomicity typu check-then-act. Zde vlákno 1 nejdříve kontroluje, zda se v proměnné *map* již nachází hodnota s klíčem *key1*. Pokud tam taková hodnota ještě není, vlákno 1 ji do *map* vloží. Jelikož tato kontrola a následné vložení netvoří atomickou operaci, může mezi nimi dojít ke *změně kontextu*, při které vlákno 2 do *map* vloží vlastní hodnotu se stejným klíčem.

Nesprávné pořadí

Nesprávné pořadí je kategorie chyb, ke které nastane, pokud v programu není vynucen určitý požadavek na pořadí událostí nezbytný pro správnou funkci programu. Jako takovýto požadavek lze uvést například nutnost otevřít soubor před zápisem do něj. Jedná se o výrazně méně studovanou kategorii chyb než porušení atomicity či časově závislá chyba nad daty[7].

Definice 2.3.3. Běh programu obsahuje chybu nesprávného pořadí, jestliže v jeho běhu nejsou některé instrukce vykonány v očekávaném pořadí.

Uváznutí

Uváznutí neboli deadlock představuje silně studovanou kategorii chyb. Pro jeho definici je potřeba znát:

- Vlákna, která jsou blokována a čekají na určitou událost.
- Množiny událostí, které každé zablokované vlákno mohou odblokovat.

Definice 2.3.4. Běh programu obsahuje množinu S uváznutých vláken, pokud je každé vlákno v S blokováno a čeká na událost, kterou by mohlo vyvolat pouze jiné vlákno z S.

Příklad 2.3.3. Příklad uváznutí:
Lock lockA;
Lock lockB;

Vlákno 1:

```
lockA.lock();
  lockB.lock();
  lockB.unlock();
lockA.unlock();
```

Vlákno 2:

```
lockB.lock();
  lockA.lock();
  lockA.unlock();
lockB.unlock();
```

V příkladu 2.3.3 naleznete jednu z typických situací, kdy může dojít k uváznutí. Vlákno 1 se po úspěšném získání zámku *lockA* snaží získat druhý zámek *lockB*. Vlákno 2 přistupuje ke stejným zámkům v opačném pořadí. Pokud budou tyto instrukce provedeny v pořadí, kdy nejdříve obě vlákna získají první zámek a poté budou získávat druhý, dojde k uváznutí. Vlákno 1 vlastnící *lockA* bude čekat na uvolnění zámku *lockB*, který ale vlákno 2 může uvolnit až po získání zámku *lockA*.

Zmeškaný signál

Zmeškaný signál je další méně studovaná kategorie chyb. Pro jeho identifikaci potřebujeme vědět, který signál měl být doručen kterému vláknu.

Definice 2.3.5. Běh programu obsahuje zmeškaný signál, pokud při něm dojde k zaslání signálu, který není doručen příslušnému vláknu.

Příklad 2.3.4. Příklad zmeškaného signálu:
Object obj;

Vlákno 1:

```
synchronized(obj){
  obj.wait();
}
```

Vlákno 2:

```
synchronized(obj){
  obj.notify();
}
```

Na příkladu 2.3.4 lze demonstrovat chybu zmeškaného signálu. Obě vlákna přistupují k monitoru objektu *obj*. Vlákno 1 za pomocí operace *wait* vyčkává, dokud jiné vlákno nevyvolá operaci *notify* či *notifyAll*. Pokud při běhu programu může tuto operaci vyvolat pouze vlákno 2, dochází ke zmeškanému signálu, protože není zaručeno, že operace *wait* proběhne před *notify*.

2.3.2 Chyby v živosti

Tato kategorie chyb obecně zabraňuje něčemu dobrému, aby se stalo. Identifikace a odstraňování chyb v živosti je výrazně náročnější než odhalování chyb v bezpečnosti, neboť algoritmy s nimi pracující musí prokázat, že určitá událost nemůže nikdy nastat [7]. S definicemi tohoto druhu chyb úzce souvisí problém *hladovění* a *chování nevykazujícího pokrok*.

Definice 2.3.6. Běh programu vykazuje hladovění, pokud v něm existuje vlákno, které čeká buď blokované nebo ve smyčce neustále provádějící nějaký výpočet na událost, která nemusí nikdy nastat.

Problém hladovění zahrnuje i případy uváznutí, zmeškaných signálů a níže definované uváznuté vlákno či uváznutí s aktivním čekáním [7]. V těchto situacích událost, na kterou

vlákno čeká, nemůže nikdy nastat, a proto je daná situace samozřejmě nežádoucí. Ovšem existuje mnoho programů, ve kterých je hladovění do určité míry tolerováno. Předpokládá se totiž, že pokud nějaká událost *může* vždy v praxi s určitou nenulovou pravděpodobností nastat, pak také někdy *nastane*.

Chování nevykazující pokrok

Pro definici tohoto pojmu je potřeba nejdříve určit význam termínu pokrok. Tento pojem obecně vyjadřuje, že vlákno provádí nějakou užitečnou práci neboli že vykonává to, co programátor zamýšlel. Jednotlivé přesné definice se od sebe často liší. Jako pokrok lze například vyjádřit, že vlákno komunikuje s ostatními vlákny [7].

Definice 2.3.7. Program pracující v nekonečné smyčce obsahuje chování nevykazující pokrok, pokud v něm existuje vlákno, které neustále provádí nějaký výpočet, tedy není zablokováno, ale nevykazuje žádný pokrok.

Chování nevykazující pokrok lze popsat jako speciální případ hladovění v programu pracujícím v nekonečné smyčce [7]. Nicméně hladovění je širší termín, tento jev se samozřejmě může objevit i v konečných programech.

Uváznutí s aktivním čekáním

Uváznutí s aktivním čekáním neboli *livelock* je specifický případ chování nevykazujícího pokrok, které může být definováno následujícím způsobem.

Definice 2.3.8. Běh programu pracujícího v nekonečné smyčce obsahuje uváznutí s aktivním čekáním, pokud se v něm nachází množina vláken S taková, že každé vlákno v S běží v nekonečné smyčce, ve které by ale nekonečně dlouho běžet nemělo. Dostat by se z ní ale mohlo pouze pokud by jiné vlákno náležící S svoji nekonečnou smyčku opustilo.

Častým problémem bývá záměna termínů hladovění, chování nevykazujícího pokrok a uváznutí s aktivním čekáním [7]. Tento problém má za důsledek, že se v literatuře objevují algoritmy podle definice určené k odhalování uváznutí s aktivním čekáním, nicméně ve skutečnosti odhalují pouze hladovění či chování nevykazující pokrok. Autoři [7] si nejsou vědomi toho, že by existovaly techniky určené k odhalování uváznutí s aktivním čekáním tak, jak bylo prezentován v definici 2.3.8.

Příklad 2.3.5. Příklad uváznutí s aktivním čekáním:

Lock lockA;
Lock lockB;

Vlákno 1:

```
lockA.lock();
while(1){
// try to get lockB as well
if(lockB.tryLock())
  break;
}
...
lockB.unlock();
lockA.unlock();
```

Vlákno 2:

```
lockB.lock();
while(1){
// try to get lockA as well
if(lockA.tryLock())
  break;
}
...
```

```
lockA.unlock();                    lockB.unlock();
```

Uváznutí s aktivním čekáním lze demonstrovat na příkladu 2.3.5. Vlákno 1 po získání zámku *lockA* vstupuje do smyčky, ze které bude pokračovat dále až po získání druhého zámku *lockB*. Vlákno 2 vykonává stejnou činnost, ale přistupuje k zámkům v opačném pořadí. Při proložení událostí, kdy obě vlákna získají první zámek a poté vstoupí do cyklu, se tento cyklus stává nekonečným a dochází tedy k uváznutí s aktivním čekáním.

Blokované vlákno

Blokované vlákno se může objevit, pokud v běhu programu dojde k blokaci vlákna, které následně čeká na jednu z množiny událostí, která by ho odblokovala. Pro identifikaci blokovaného vlákna je třeba určit tyto odblokující události.

Definice 2.3.9. Běh programu obsahuje blokované vlákno, pokud v něm existuje vlákno čekající na určitou událost, která ale nikdy nenastane.

Příklad 2.3.6. Příklad blokovaného vlákna:
Object o;

Vlákno 1:

```
sychronized(o){
        o.wait();
}
```

Vlákno 2:

V příkladu 2.3.6 naleznete chybu typu blokované vlákno. Vlákno 1 vstoupilo do monitoru objektu *obj* a poté vyvolalo operaci *wait*. Vlákno 2, které mělo nad stejným monitorem vyvolat operaci *notify*, ovšem skončilo předčasně či vlivem nějakého jiného problému tuto operaci nevyvolalo. Vlákno 1 se v tuto chvíli stalo blokovaným vláknem.

2.4 Metriky pro testování vícevláknových aplikací

Při testování je zapotřebí definovat prostředek na měření, jak velká část testovaného programu již byla prozkoumána. Tato informace následně poslouží při rozhodování, zda byl program již otestován do dostatečné míry. Pokud tomu tak není, je nutné míru pokrytí navýšit, čehož se docílí přidáním nových testovacích případů. V případě vícevláknových programů může kvůli jejich nedeterminismu pokrytí zvýšit i spuštění již existujících testů znovu. Pro účel měření míry pokrytí byly navrženy různé metriky pokrytí. Klasické metriky pro testování sekvenčních programů jako například *code coverage* či *condition coverage* se ovšem ukázaly pro testování vícevláknových programů jako nevhodné, neboť se v nich nebere v potaz fakt, že spuštění kódu se stejnými vstupy může potenciálně vést k různým výsledkům [8]. Proto byly navrženy nové metriky zaměřující se na souběžnost událostí a komunikaci mezi vlákny. Některé již dříve používané metriky, jako například *DUpairs*, byly ze stejných důvodů rozšířeny. V [8] byly též navrženy nové metriky postavené na detekčních algoritmech pro různé druhy chyb ve vícevláknových programech. Autoři jednotlivé metriky otestovali a vyhodnotili, že skutečně postihují vícevláknový charakter programu lépe než dřívější metriky a navíc poskytují programátorům více informací o tom, za jakých okolností k chybě došlo. Přehled jednotlivých metrik se nachází v tabulce 2.1. V prvním sloupci se

Metrika	Task pokrytí	Typ
Avio	(pl_1,pl_2,pl_3)	N
Avio*	(pl_1,pl_2,pl_3,var,t_1,t_2)	N
Eraser	(pl_1,state,lockset)	N
Eraser*	(pl_1,var, state, lockset,t_1)	N
GoldiLock	(pl_1,goldiLockSetSC)	N
GoldiLock*	(pl_1,var,goldiLockSetSC,t1)	N
GoodLock	(pl_1,pl_2,l_1,l_2)	N
GoodLock*	(pl_1,pl_2,l_1,l_2,t_1)	N
HBPairs	(pl_1,pl_2,syncObj)	N
HBPairs*	(pl_1,pl_2,syncObj,t_1,t_2)	N
ConcurPairs	(pl_1,pl_2,switch)	E
DUPairs	(pl_1,pl_2,var)	E
DUPairs*	(pl_1,pl_2,var,t_1,t_2)	M
Sync	(pl_1,mode)	E

Tabulka 2.1: Výčet studovaných coverage metrik

nachází jméno metriky, ve druhém podoba jejího artefaktu a ve třetím typ metriky, kde *N* reprezentuje novou metriku navrženou v [8], *E* již existující a *M* modifikovanou. V následujícím textu budou postupně jednotlivé metriky rozebrány. Detailnější informace lze dohledat v [8, 7, 3].

Avio

Avio je metrika postavená na stejnojmenném algoritmu určenému k detekci porušení atomicity. Její základní a rozšířená verze mají formu *(pl_1,pl_2,pl_3)* a *(pl_1,pl_2,pl_3,var,t_1,t_2)*, kde *pl_1* a *pl_2* označují dvě za sebou jdoucí místa v kódu, odkud vlákno *t_1* přistupovalo k proměnné *var*, a *pl_3* označuje místo v kódu, odkud vlákno *t_2* přistoupilo k proměnné *var* mezi událostmi v *pl_1* a *pl_2*. Tento přístup by v případě, že *pl_1* a *pl_2* tvoří atomický blok, znamenal porušení atomicity.

Eraser

Metrika Eraser je založená na stejnojmenném algoritmu pro detekci časově závislých chyb nad daty, který pro ověření správnosti využívá techniku *locksets*. Tato technika pro každou proměnnou počítá průnik množin zámků jednotlivých vláken, které k proměnné přistupovaly, a hlásí chybu, jakmile se tato množina stane prázdná. Jeho základní a rozšířená verze mají podobu *(pl_1,state,lockset)* a *(pl_1,var,state,lockset,t_1)*, kde *pl_1* je místo, kde odkud vlákno *t_1* přistoupilo k proměnné *var*. *Lockset* reprezentuje množinu zámků, které vlákno *t_1* v danou chvíli vlastnilo a *state* je stav konečného automatu, který je využit v algoritmu pro detekci chyb.

GoldiLock

Metrika Goldilock je postavena na vyspělém algoritmu Goldilock určenému k detekci časově závislých chyb nad daty. Jeho základní a rozšířená verze mají formu *(pl_1,goldiLockSetSC)* a *(pl_1,var,goldiLockSetSC,t1)*, kde *pl_1* vyjadřuje místo v kódu, odkud vlákno *t_1* přistoupilo k proměnné *var*. *GoldiLockSetSC* reprezentuje *lockset* vypočítaný algoritmem Goldilock.

Goodlock

Tato metrika je postavena na stejnojmenném algoritmu pro detekci uváznutí. Jeho základní a rozšířená verze mají formu *(pl_1,pl_2,l_1,l_2)* a *(pl_1,pl_2,l_1,l_2,t_1)*, kde pl_2 vyjadřuje místo v kódu odkud vlákno t_1 přistoupilo k zámku l_2, když už v danou chvíli vlastnilo zámek l_1 získaný v pl_1.

HBPairs

Jedná se o metriku postavenou na tvorbě *happens-before* relace. Jeho základní a rozšířená verze mají formu *(pl_1,pl_2,syncObj)* a *(pl_1,pl_2,syncObj,t_1,t_2)*, kde pl_1 označuje místo v programu, které bylo vláknem t_1 navštíveno dříve než místo pl_2 navštívené vláknem t_2. *SyncObj* identifikuje objekt, pomocí kterého bylo možné tyto dva přístupy seřadit.

ConcurPairs

Tato metrika bere v potaz dvě po sobě jdoucí události. Má formu *(pl_1,pl_2,switch)*, kde $pl_{\{1,2\}}$ označuje dvě místa v programu, kde došlo k událostem, a $switch \in \{true, false\}$ nabývá hodnoty *true*, pokud tyto události byly vykonány rozdílnými vlákny a *false* jinak.

DUPairs

Definition Use pairs je metrika zabývající se přístupy k proměnným. Jeho základní a rozšířená verze mají formu *(pl_1,pl_2,var)* a *(pl_1,pl_2,var,t_1,t_2)*, kde pl_1 je místo v programu, ze kterého vlákno t_1 definovalo proměnnou *var*, a pl_2 je místo, odkud vlákno t_2 hodnotu proměnné použilo. Jedná se o rozšířenou metriku, která v základní podobě byla využívána při testování sekvenčních programů.

Synchronization coverage

Tato metrika zkoumá použití synchronizačních primitiv. Má formu *(pl_1,mode)*, kde pl_1 označuje místo v kódu, odkud se k danému primitivu přistoupilo, a mode označuje situaci specifickou pro dané primitivum, například u zámku *mode* ∈ *{synchronization visited, synchronization blocking, synchronization blocked}* *synchronization visited* pouze označuje, že k danému zámku bylo přistupováno, *synchronization blocking* označuje, že vlákno přistupující k zámku způsobilo zablokování jiného vlákna a *synchronization blocked* vyjadřuje, že vlákno přistoupilo k zámku a bylo zablokováno.

2.5 Heuristiky pro testování vícevláknových aplikací

Jak bylo zmíněno výše, chyby ve vícevláknových programech se často objevují pouze za velice specifických okolností. V praxi se může stát, že program, který fungoval správně měsíce či roky, najedou z ničeho nic selže. Navíc chyba, která způsobila pád programu, se při následném ladění nemusí objevit. Proto byly navrženy různé heuristiky za účelem zvýšit množství prozkoumaných proložení událostí během testu a tedy i zvýšit šanci na odhalení chyby [3].

Jedna z možných cest je *deterministické testování*, kdy se s využitím zásahu do JVM program spustí skrz testovací nástroj, který má nad ním úplnou kontrolu. Může tedy přesně určovat, jaké proložení událostí bude prozkoumáno a též postupně systematicky prozkoumávat všechna proložení událostí. Tato technika se v některých ohledech blíží *model checkingu*,

a proto bývá někdy označována jako *runtime model checking*. Trpí tedy stejnými problémy. Množství proložení k prozkoumání je obrovské, což znemožňuje efektivní nasazení této techniky na programy větších rozměrů. Tento problém bývá označován jako *state space explosion* [7].

Jako méně výpočetně náročná alternativa se používá *vkládání šumu*. Tato technika vkládá do testovaného programu na specifická místa instrukce, které ovlivňují rozhodování plánovače operačního systému a můžou tedy vést k novým ještě neprozkoumaným proložením událostí [3]. Právě technika vkládání šumu je využita v nástrojích RoadRunner a ConTest popsaných v sekcích 3.2 a 3.3. Při jejím využití musí být zváženy dvě důležité oblasti. První z nich se zabývá tím, jaký druh šumu do testované aplikace vložit, a druhá zkoumá, na která místa v programu tento šum vkládat. Problematikou druhu šumu se zabývají *noise seeding heuristiky* a problematikou umístění šumu *noise placement heuristiky*.

2.5.1 Noise seeding heuristiky

Tyto heuristiky se zabývají otázkou, jaký druh šumu do testované aplikace vložit. První zkoušenou možností byla *změna priority*. Jak napovídá název, tato technika mění priority vláken testovaného programu. V praktických testech se nicméně ukázala jako neefektivní [7]. Další možností je vložit do aplikace volání funkce *sleep()* pro uspání vláken či *yield()* pro vynucené přepnutí kontextu. Další heuristika postavená na funkci *wait()* funguje následovně. Vlákna aplikace jsou nejdříve nucena získat přístup k monitoru některého objektu testovacího softwaru[8] a poté jsou zablokovány funkcí *wait()* s nastaveným *timeoutem*. Další heuristika *syncYield* opět využívá monitoru, ke kterému nejdříve musí vlákna přistoupit, a následně je na nich vyvolána instrukce *yield()*. V heuristice *busyWait* se do vláken místo synchronizace vkládá výpočet, který je na nějakou dobu zdrží od jejich normální činnosti. Všechny výše uvedené heuristiky vyjma změny priority mohou být parametrizovány *sílou šumu*. Tento parametr určuje, jak silný šum vložit, a jeho konkrétní význam je specifický pro jednotlivé druhy šumu. Například u heuristiky *sleep* určuje dobu spánku a u heuristiky *yield* počet přepnutí kontextu. Při praktických testech se nejlépe osvědčila heuristika *sleep*, nicméně při vysokém počtu běžících vláken dosahovala i heuristika *yield* dobrých výsledků. Výše zmíněné heuristiky lze také náhodně kombinovat. Vzniklá heuristika je označována jako *mixed*.

Jako sofistikovanějších heuristiku lze uvést *HaltOneThread*, která náhodně vybere a uspí jedno vlákno na tak dlouhou dobu, dokud ostatní vlákna ještě mohou pokračovat v činnosti. Heuristika *TimeoutTampering* naopak z aplikace náhodně odebírá či zkracuje dobu volání funkce *sleep()*. Tato heuristika je postavena na pozorování, že uspávání vláken je někdy programátory využito místo synchronizace [3].

2.5.2 Noise placement heuristiky

Tato kategorie heuristik zkoumá, na která místa testovaného programu šum vkládat. Nástroje ConTest i RoadRunner poskytují možnost šum vkládat před a po provedení instrukcí souvisejících s vícevláknovým programováním. Toto nastavení naštěstí nepřináší příliš omezení, neboť vkládat šum na jiná místa se jeví jako neefektivní. Nejjednodušší heuristika *random* je založena na náhodném výběru těchto míst. Sofistikovanější heuristika *sharedVars-all* omezuje místa pro vkládání šumu pouze před a po instrukce pro přístup ke sdíleným proměnným. Její modifikace *sharedVars-one* v každém běhu náhodně vybere pouze jednu pro-

[8] Musí se jednat o nový jinak v testované aplikaci nepoužívaný monitor, jinak by hrozilo uváznutí.

měnnou, která bude šumem ovlivňována. Tato heuristika se ukázala jako poměrně efektivní [3]. Výše zmíněné heuristiky mohou být parametrizovány *frekvencí šumu*, která ovlivňuje pravděpodobnost, s jakou se šum vloží. Před spuštěním testu je definováno číslo v intervalu <0;1000> vyjadřující, jak často se má šum vkládat. Poté se při každé události generuje náhodné číslo ze stejného intervalu. Šum se do programu vloží, pouze pokud je toto nově vygenerované číslo menší než původně zvolená hodnota.

2.6 Abstraktní identifikátory pro dynamickou analýzu vícevláknových programů

Jedna z výzev, kterým se při dynamické analýze vícevláknových programů čelí, je nalezení vhodné reprezentace klíčových entit testovaného programu. Míra abstrakce využitá při této identifikaci výrazným způsobem ovlivňuje celý proces analýzy testovaného programu hlavně při *saturačním* testování a *testování s využitím automatického hledání nejvhodnějších testových konfigurací* popsaném v sekci 2.7. Pokud bude identifikace příliš detailní, množství zaznamenaných artefaktů poroste prudce, aniž by skutečně docházelo k pozorování nového či zajímavého chování testovaného programu. Naopak příliš abstraktní identifikace povede k nízkému počtu zaznamenaných artefaktů [3]. Navíc identifikace pro jeden typ entit často nebývá vhodná pro jinou kategorii.

2.6.1 Místo v programu

Prvním problémem je identifikace míst v testovaném programu. Jedná se o základní stavební kámen, který využívají jak identifikace vláken a objektů, tak i jednotlivé metriky pokrytí [8]. Nástroj ConTest popsaný v sekci 3.3 poskytoval tuto identifikaci jako součást své API. RoadRunner popsaný v sekci 3.2 namísto toho poskytuje v rámci API sérii objektů, ze kterých lze řetězec reprezentující místo v programu vytvořit. Pro účely této práce i dalších experimentů byl zvolen formát *(file:line:offset)*, kde *file* reprezentuje zdrojový soubor, *line* řádek a *offset* index na daném řádku.

2.6.2 Identifikace vláken

JVM sice poskytuje pro vlákna identifikaci ve formě přirozeného čísla, nicméně ta je pro účely dynamické analýzy nepoužitelná. Tato čísla jsou totiž přidělována na základě pořadí, ve kterém se vlákna vytváří, což se mezi jednotlivými běhy liší [3]. Proto bylo potřeba navrhnout jinou formu identifikace. První způsob identifikace využíval datového typu vlákna. Ten se ale ukázal jako neefektivní, neboť vlákna se stejným datovým typem mohou vlivem rozdílných vstupních dat vykonávat rozdílné činnosti. Další forma identifikace byla postavena na dříve definovaném pojmu *místo v programu*. Vychází z předpokladu, že vlákna, jejichž prvních n[9] vyvolaných metod je stejných, mohou být pro účely dynamické analýzy považována za identická. Pro ještě přesnější definici vlákna se využívá dvojice *(type, hash)*, kde *type* vyjadřuje typ vlákna[10] a *hash* je hash z prvních n metod, které dané vlákno volalo [3].

[9] Volba tohoto parametru výrazně ovlivňuje úroveň abstrakce, implicitní nastavení v coverage lib je 10.
[10] V kontextu javy je myšleno jeho datový typ.

2.6.3 Identifikace objektů

Pro identifikaci objektů se osvědčila dvojice *(type,progloc)*, kde *type* vyjadřuje datový typ objektu a *progloc* vyjadřuje místo v kódu, kde byl objekt vytvořen [3].

2.7 Testování jako optimalizační problém

Vzhledem k velkému množství různých konfigurací lze testování vícevláknových programů s využitím vkládání šumu definovat jako *optimalizační problém*. Na obecné rovině se jedná o kategorii úloh, jejímž cílem je nalezení kombinace vstupních parametrů, při které má zkoumaný systém požadované vlastnosti. Konkrétně v případně testování vícevláknových programů dochází k opakovanému spouštění testů a nástroj se snaží nalézt takové konfigurace, při jejichž využití stoupá šance na odhalení chyby nebo pokrytí dle zvolené metriky. Všechny testové konfigurace se reprezentují jako stavy obecně n-rozměrného *stavového prostoru*. Každý parametr patřící do konfigurace testu se zakóduje jako jedna *dimenze*. *Kardinalita* dimenze je dána počtem možných nastavení tohoto parametru. Do testové konfigurace patří jak nastavení pro samotný testovací případ, tak i pro heuristiky vkládající šum. Tento fakt způsobuje, že stavový prostor je poměrně rozsáhlý [7]. Konkrétní podoba stavového prostoru může být například následující: *(noiseFreqvency, noiseStrength, noiseType, haltOneThread, timeoutTamper, threadCount)*. V tomto případě prvních 5 dimenzí reprezentuje nastavení pro vkládání šumu popsané v sekci 2.5 a poslední dimenze odpovídá parametru daného testovacího případu, který určuje počet spuštěných vláken. Při kardinalitě dimenzí (1001,102,6,2,2,10) je počet stavů ve stavovém prostoru roven $1001 * 102 * 6 * 2 * 2 * 10 = 24504480$. Konkrétní stav se reprezentuje vektorem celých čísel, kde každá hodnota odpovídá jednomu parametru testové konfigurace.

2.7.1 Metaheuristické přístupy k prohledávání stavového prostoru

Prohledání všech stavů v rozsáhlém stavovém prostoru je z časového i výpočetního hlediska náročné. V praxi testy musí být dokončeny poměrně rychle, například během jedné noci, aby programátoři ráno po příchodu do práce již měli výsledky testování jejich včerejší práce k dispozici. Z tohoto důvodu byly testovány různé metaheuristické přístupy za účelem nalezení vhodných testových konfigurací bez nutnosti prohledávat celý stavový prostor. Tyto techniky byly pro určení jejich efektivnosti porovnávány s náhodným prohledáváním. V [7] autoři využili pro hledání nejlepších konfigurací *genetické algoritmy*. Výsledky prezentované v této práci ukázaly, že metaheuristické přístupy skutečně mohou být s úspěchem na testování vícevláknových aplikací s využitím vkládání šumu nasazeny. Pro většinu testovacích případů prezentovaných v práci porazily genetické algoritmy náhodný průchod stavovým prostorem z hlediska počtu nalezených chyb i doby nutné k nalezení optimální konfigurace. Pouze u jednoho testovacího případu byl náhodný průchod efektivnější. V daném případě totiž byla *fitness funkce* silně ovlivněna dobou běhu programu. Toto nastavení se poté negativně projevilo při upřednostňování kratších běhů programu před delšími, při kterých ale potenciálně mohlo být objeveno více chyb. Testy tedy ukázaly, že pro efektivní nasazení genetických algoritmů je klíčové zvolit vhodnou fitness funkci [7].

Ačkoliv se prokázalo, že metaheuristické přístupy skutečně šanci na odhalení chyby zvyšují, rozdíl oproti náhodnému prohledávání naneštěstí není zatím dostatečně výrazný. Tato bakalářská práce je motivována právě výše uvedeným faktem a klade si za cíl s nasazením

přesnějších heuristik pro vkládání šumu popsaných v podsekci 5.1 rozšířit stavový prostor a umožnit genetickým algoritmům více projevit svůj potenciál.

Kapitola 3

Nástroje podporující testování vícevláknových aplikací

V této kapitole se nachází popis nástrojů, které implementují techniky popsané v předchozích kapitolách. Konkrétně se jedná o SearchBestie v sekci 3.1, RoadRunner v sekci 3.2, ConTest v sekci 3.3 a Coverage lib v sekci 3.4. S nástrojem ConTest se autor seznámil pouze z teoretického hlediska, ostatní nástroje byly v rámci práce aktivně používány a rozšiřovány.

3.1 SearchBestie

SearchBestie je generický nástroj určený pro řešení *search problémů* či *optimalizačních problémů*, což je kategorie úloh, jejichž cílem je hledání kombinací vstupních parametrů, pro něž daný systém vykazuje žádané vlastnosti [7]. V kontextu testování vícevláknových aplikací lze tyto vlastnosti definovat dvěma způsoby [1]. Prvním způsobem je pochopitelně odhalení chyby. Nicméně vzhledem k faktu, že tato kategorie chyb bývá odhalována jen poměrně zřídka, bylo potřeba zavést mechanismus pro porovnání dvou běhů, při nichž chyba odhalena nebyla. Jedno z možných řešení, které se aktivně využívá, je jako hledanou vlastnost definovat vyšší počet zaznamenaných artefaktů nebo zaznamenání vzácných málo se vyskytujících artefaktů označovaných také jako *vzácné artefakty* [8]. Autorem nástroje je Ing. Zdeněk Letko, Ph.D.

Samotné jméno SearchBestie je složeno se slov Search Based Testing Enviroment. Jak napovídá tento název, cílem SearchBestie není provádět samotné spouštění testů, ale řešení testování jakožto *search* problému [7]. Spouštění a instrumentaci bytecodu testovaného programu zajišťuje externí software, který se do SearchBestie zapojuje jako plug-in. Původní takto využívaný nástroj byl ConTest z firmy IBM [2]. Samotný vývoj SearchBestie probíhal ve spolupráci s vědci z IBM. Ovšem kvůli problémům popsaným v sekci 3.3 již není možné nástroj dále využívat. Proto bylo potřeba najít vhodnou alternativu, za kterou byl zvolen RoadRunner [4].

Architektura SearchBestie se skládá ze čtyř modulů, jmenovitě Manageru, Executoru, Storage a Search [6]. Aby byly zjednodušeny případné úpravy nástroje, sestává každý modul ze dvou částí. První z nich je rozhraní specifikující způsob, jakým daný modul se zbytkem infrastruktury komunikuje. V každém modulu se též nachází jedna nebo více implementací tohoto rozhraní. Tento návrh umožňuje snadné zakomponování externího nástroje do

SearchBestie. Stačí pouze napsat třídu implementující rozhraní pro komunikaci se zbytkem infrastruktury a transformující jednotlivá volání do podoby API externího nástroje[1].

Modul **Storage** realizuje stavový prostor nástroje. Jeden stav v tomto prostoru reprezentuje jednu testovou konfiguraci. Současná implementace umožňuje definovat dimenze čtyř různých datových typů:

- *celé číslo* s intervalem povolených hodnot,
- *booleovský typ*,
- *výčtový typ* s položkami typu *celé číslo*, *boolean*, či *řetězec*,
- *dimenze testů* obsahující test, který bude s danými parametry spuštěn.

Stavy jsou reprezentovány vektory hodnot z příslušných dimenzí. Modul také umožňuje nastavit filtr pro vyloučení stavů reprezentujících nerealizovatelné či nesmyslné konfigurace. Součástí jsou též metody pro ukládání výstupních dat do textového souboru či relační databáze. Do modulu se též ukládají výsledky pro pozdější analýzu.

Executor hodnotí jednotlivé stavy z modulu Storage. Toto ohodnocení buď může být přímo naimplementováno v rámci množiny tříd v Javě nebo dojde ke spuštění externího procesu, který daný stav ohodnotí. V kontextu testování vícevláknových programů dojde ke spuštění externího procesu ConTestu či RoadRunneru, který spustí testovaný program a analyzuje jeho běh. Výsledky jsou po prozkoumání uloženy zpět do Storage. Pro zvýšení programátorského komfortu jsou součástí tohoto modulu i speciální třídy. *RepeatedTest* umožňuje opakované spouštění testu za účelem získání kumulované či průměrné hodnoty. *TimedTest* umožňuje nastavit časový limit pro spouštěný test, po kterém je jeho běh přerušen. Tato třída je například užitečná pro ochranu před situacemi, kdy se testovaný program dostane do nekonečné smyčky s aktivním čekáním nebo když dojde k uváznutí.

Modul **Search** prochází stavový prostor v modulu Storage a určuje, jaké stavy budou dále prozkoumány. V rámci současné implementace jsou v nástroji k dispozici následující techniky:

- noSearch vracející předem definované stavy,[2]
- randomSearch vracející náhodně vybrané stavy,
- sequentialSearch a revSequentialSearch procházející stavový prostor sekvenčně,
- hillClimbingSearch vycházející ze stejnojmenného algoritmu pro hledání lokálního optima,
- prohledávání stavového prostoru s využitím genetických algoritmů.

Hlavním řídícím modulem je **Manager**, jehož činnost může být popsána následovně. Nejdříve načte konfiguraci ze vstupního xml[3] souboru a následně inicializuje všechny ostatní moduly. Poté přejde do hlavní testovací smyčky, ve které postupně spouští jednotlivé testy, dokud není splněna ukončovací podmínka. Na závěr vyexportuje data z modulu Search do výstupního souboru či databáze.

[1] Z hlediska návrhových vzorů je tedy nutno implementovat Adapter.

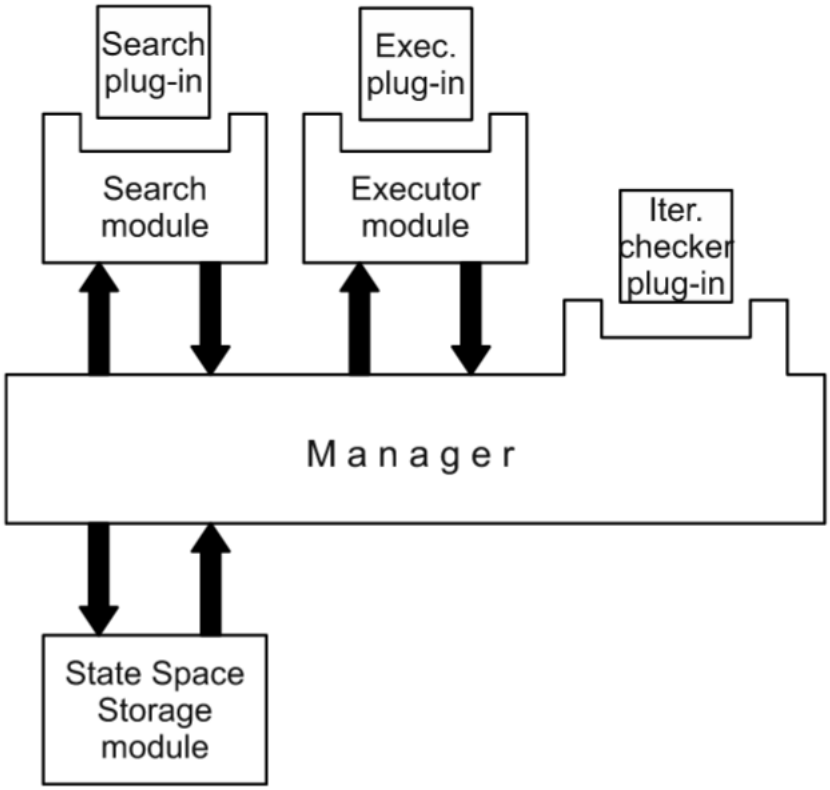
[2] Vhodné pro účely testování opakovaného hodnocení určité množiny konfigurací. V rámci práce byla tato technika také využita pro srovnávání ConTestu a RoadRunneru na předem definované množině konfigurací.

[3] eXtensible Markup Language - značkovací jazyk z rodiny SGML určený pro uchovávání a přenos dat.

Celá architektura je psána genericky, aby pro případnou implementaci dalších testů či integraci nových nástrojů pro instrumentaci již nebylo potřeba zasahovat do jádra SearchBestie. Informace o tom, jaký test se má spustit, a všechny jeho parametry, jsou zadány ve výše zmíněném vstupním xml souboru. Každý nově napsaný test musí implementovat určité rozhraní a korektním způsobem komunikovat s ostatními moduly SearchBestie.

Struktura SearchBestie může být znázorněna obrázkem 3.1.

Obrázek 3.1: Vysokoúrovňový pohled na architekturu SearchBestie [6]



3.2 RoadRunner

Nástroj RoadRunner byl vyvinut ve Spojených státech amerických na univerzitách University of California at Santa Cruz a Williams College jakožto efektivní řešení pro testování vícevláknových aplikací. Zajišťuje nízkoúrovňovou práci s bytecodem a poskytuje čistou API pro programování nástrojů provádějících *dynamickou analýzu*. Autoři zdůrazňují především jeho rychlost a škálovatelnost. RoadRunner byl použit pro testování i rozsáhlých programů jako například IDE Eclipse [4].

RoadRunner provádí instrumentaci bytecodu s pomocí knihovny *asm*. Tato knihovna umožňuje bytecode číst i modifikovat. Roadrunner s využitím technologie *java agent* za běhu programu načítá bytecode testované aplikace a vkládá do něj volání svých vlastních metod. Vzhledem k primárnímu určení nástroje pro testování vícevláknových programů se

instrumentace provádí při událostech, které souvisejí s paralelním programováním. Seznam sledovaných událostí je následující:

- start vlákna,
- vytvoření vlákna,
- přístup k proměnné libovolného datového typu,
- odemknutí a zamknutí zámku,
- metody sleep, wait, notify a join,
- zánik vlákna,
- ukončení aplikace neošetřenou výjimkou[4].

Kromě sledování těchto událostí RoadRunner také shromažďuje metadata o testovaném programu. API nástroje poskytuje objekty reprezentující jednotlivé třídy testovaného programu. Každé vlákno a každý zámek je též reprezentován speciálním objektem. Vzhledem k této poměrně vysoké úrovni abstrakce se programátor může soustředit čistě na implementaci daného analyzátoru bez nutnosti zabývat se manipulací s bytecodem [4].

RoadRunner používá pro jednotlivé analyzátory označení *tools*. Každý analyzátor dědí z bázové třídy *Tool*, která implementuje *callback metody* volané před a po výše zmíněnými událostmi.[5] Tyto *tooly* lze řadit za sebe a spouštět v řetězci, kde první *tooly* mohou sloužit jako filtry, aby jejich následníci, jejichž činností může být nějaká náročná analýza, nemusely zpracovávat události, o kterých je předem zřejmé, že jsou pro danou analýzu nezajímavé. Řetězec analyzátorů bývá označen jako *toolchain*. Tato technika také umožňuje spustit více analyzátorů najednou.

3.3 ConTest

Jelikož nástroj ConTest je chráněn licenčním ujednáním, seznámení s ním proběhlo pouze na teoretické úrovni. Nástroj, jehož jméno je složeno ze slov Concurrency testing, byl vyvinut vědci z IBM za účelem testování vícevláknových aplikací [2].

Nástroj je svým principem podobný RoadRunneru popsanému v 3.2. ConTest provádí instrumentaci bytecodu. Programátor jako v případně RoadRunneru definuje třídu s *callback metodami*, které jsou volány při zaznamenání událostí v testovaném programu. Stejně jako v RoadRunneru je zde pro většinu událostí *callback metoda* před a po jejím provedení.

Oproti RoadRunneru nalezneme rozdíl například v parametrech předávaných těmto metodám. V ConTestu je do metod jako parametr předávaný řetězec *program location*, který tvoří interní mechanismy nástroje a jeho strukturu tedy nelze měnit. Do metod volaných při práci s proměnnými je také jako parametr přímo předávána reference na objekt, k jehož atributu se přistupuje, což zjednodušuje implementaci nástrojů provádějících *dynamickou analýzu*. Za zmínku také stojí, že zatímco RoadRunner rozlišuje pouze přístup k *volatilním* a *nevolatilním* proměnným, ConTest definuje *callbacky* pro jednotlivé datové typy proměnných samostatně. Hlavní parametry ConTestu se nastavují za pomoci konfiguračního souboru, což je z autorova pohledu oproti předávaní přes parametry praktičtější řešení.

[4] Událost není součástí původní verze API RoadRunneru, byla přidána v rámci této práce, viz 4.4.

[5] Pro události přístup k proměnné a zámku je k dispozici pouze jedna metoda.

Už v současné chvíli je totiž počet přepínačů, se kterými je RoadRunner ze SearchBestie spouštěn, větší než 30. Tento počet navíc může potenciálně růst. Pro parametry ConTestu předávané přes konfigurační soubor se používá označení *King properties*.

Projekt IBM, v rámci nějž byl ConTest navržen a naimplementován, naneštěstí roku 2011 skončil a od té doby nástroj nikdo nerozvíjí. Z tohoto důvodu ConTest podporuje pouze Javu verze 6, což znemožňuje testování moderních aplikací. Další využití je též limitováno licenčním ujednáním, které znemožňuje celou infrastrukturu SearchBestie poskytnout jako službu nějakému komerčnímu zákazníkovi.

3.4 Coverage lib

Coverage lib je knihovna pro dynamickou analýzu aplikací v Javě. Autorem je opět Ing. Zdeněk Letko, Ph.D. Knihovna poskytuje třídy pro reprezentaci objektů, vláken a zámků u testovaných programů. Pro jejich identifikaci se využívá techniky popsané v sekci 2.6. V knihovně jsou též implementovány mechanismy pro tvorbu artefaktů jednotlivých metrik popsaných v sekci 2.4.

Knihovna byla původně vyvinuta pro nástroj ConTest, což způsobilo, že některé její části byly na tomto nástroji přímo závislé.

Kapitola 4

Propojení SearchBestie s RoadRunnerem s využitím Coverage lib

Tato kapitola se zabývá návrhem a implementací změn v SearchBestie, RoadRunneru a Coverage lib, jejichž provedení bylo nezbytné pro úspěšné propojení těchto nástrojů. Práce na tomto projektu započala již v předmětu projektové praxe a její dokončení proběhlo následně v rámci této bakalářské práce. Text v kapitole je organizován následovně. Sekce 4.1 obsahuje abstraktní popis změn provedených v jednotlivých nástrojích. Sekce 4.2 popisuje změny provedené v nástroji Searchbestie. V sekci 4.3 se nachází popis změn v Coverage lib. Sekce 4.4 popisuje změny, ke kterým došlo v nástroji RoadRunner. Sekce 4.5 nabízí experimentální srovnání RoadRunneru a ConTestu na sadě testovacích programů z repozitáře VeriFITu.

4.1 Návrh propojení

Díky generickému návrhu nástroje SearchBestie bylo možné se téměř úplně vyhnout jakýmkoliv zásahům do již existujícího kódu. Změny zahrnovaly pouze implementaci nových tříd v rámci modulu Executor, které:

1. převedou vstupní parametry z interního formátu do konfigurace nástroje RoadRunner,
2. spustí nástroj v samostatném procesu,
3. zpracují výsledky testu a předají je zpět do jádra SearchBestie.

Po implementaci výše uvedených změn již SearchBestie může RoadRunner spustit a analyzovat jeho výstupy.

RoadRunner neposkytuje žádný mechanismus pro tvorbu artefaktů pokrytí, pro tyto účely bylo potřeba využít knihovnu Coverage lib. Při jejím návrhu se nicméně nepočítalo s využitím jiného nástroje než ConTest, což způsobilo závislost některých interních mechanismů knihovny přímo na API nástroje. Tyto přímé závislosti bylo pro začlenění Coverage lib do nově vznikající infrastruktury nutno odstínit a poskytnout interně implementaci pro oba nástroje. Na novou verzi knihovny se navíc kladl požadavek umožnit pozdější implementaci efektivního zapínání a vypínání jednotlivých metrik pokrytí za běhu testované aplikace. V rámci jeho realizace došlo ke změně v hierarchii dědičnosti tříd uvnitř knihovny.

Závěrečná fáze směřovala k přizpůsobení RoadRunneru požadavkům kladeným ze strany SearchBestie. V rámci úprav došlo k několika rozšířením. Jako první lze uvést novou callback metodu vyvolanou před přístupem k prvkům pole, neboť stávající implementace neposkytovala všechny potřebné informace. K jejímu provedení bylo nutno zasáhnout do interních mechanismů nástroje provádějících instrumentaci bytecodu. Při návrhu RoadRunneru se nejspíše nepočítalo s nutností umožnit testovacím toolům bezpečné ukončení i při pádu testované aplikace kvůli neošetřené výjimce. Nicméně mechanismy uvnitř Coverage lib neprodukovaly externí výstupy, dokud nedostaly informaci, že došlo k ukončení testu. Řešení tohoto problému spočívalo v přidání další callback metody do API nástroje. RoadRunner pomocí této metody umožní toolům skončit svoji práci a teprve poté se sám ukončí. Poslední výraznější změna spočívala v implementaci chybějících noise seeding heuristik HaltOneThread a TimeoutTampering. Zbývající heuristiky do nástroje v rámci prvních experimentů přidal Ing. Zdeněk Letko, Ph.D., když ještě pracoval na fakultě.

4.2 Příprava SearchBestie pro komunikaci s RoadRunnerem

V rámci implementace došlo k zapouzdření komunikace SearchBestie s RoadRunnerem do tříd *RoadRunnerTest* a *RoadRunnerTask*. K samotnému spuštění procesu se využívá již existující třída *cz.vutbr.fit.sbestie.executor.test.Process* z nástroje SearchBestie. Jméno včetně balíku je uvedeno kvůli zdůraznění, že se nejedná o stejnojmennou třídu ze standardní knihovny jazyka Java. Ta je využita až interně uvnitř výše uvedené třídy Process pro skutečné spuštění externího nástroje[1].

RoadRunnerTest se jakožto třída reprezentující prováděný test přímo specifikuje v konfiguračním xml souboru a obdrží z něj parametry v kolekci typu asociativní mapa, kde klíčem je jméno daného konfiguračního parametru a hodnota odpovídá určenému nastavení. Z této mapy RoadRunnerTest získává dvě základní informace. První je adresář pro uložení výstupních souborů, a druhou je seznam metrik pokrytí zapnutých pro daný test. Méně abstraktní konfigurační záležitosti a samotné spuštění testu je delegováno níže do třídy RoadRunnerTask. Modul Executor poté při žádosti o ohodnocení určitého stavu volá metodu *performTest*. Její hlavní účel tkví ve spuštění testu s využitím RoadRunnerTask, měření doby délky testu a následnému zpracování výstupních souborů.

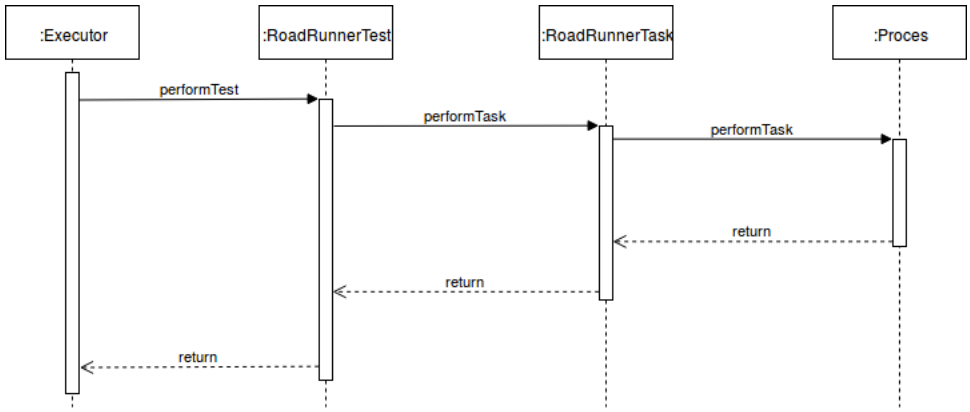
RoadRunnerTask se již stará o méně abstraktní záležitosti spuštění testu. Nejdříve kontroluje vloženou konfiguraci a ověřuje, zda obsahuje všechny potřebné parametry. Pro korektní spuštění je nutno uvést cestu k testované aplikaci a její hlavní třídu obsahující metodu main. Další povinný parametr je pochopitelně příkaz pro spuštění RoadRunneru. Za účelem zachování co nejvyšší míry zpětné kompatibility konfiguračních souborů nemusí být cesta k nástroji RoadRunner přímo uvedena v parametrech, nástroj ji bude hledat i v proměnných prostředí s využitím třídy *Runtime* ze standardní knihovny Javy. Konfigurace může obsahovat i volitelné parametry jako například *timeout* pro ukončení testu po vypršení zadaného časového limitu. Vzhledem k využití další třídy Process se RoadRunnerTask také stará o převod parametrů do podoby požadované touto třídou. Když RoadRunnerTest vyžaduje spuštění testu, volá metodu *performTask*, jejíž současná implementace deleguje spuštění na třídu Process, odchytává případné výjimky a nastavuje speciální výsledek *interrupted* v případě, že dojde k přerušení.

[1] Za povšimnutí zde stojí úroveň abstrakce oddělující jádro modulu Executor od skutečného spuštění externího procesu. Oddělují je od sebe celé tři třídy. Návrh vycházel ze způsobu spuštění ConTestu, kde je využíváno stejné abstrakce.

Třída Process očekává následující parametry. Příkaz pro spuštění procesu *command* nebo stejnojmennou proměnnou prostředí[2]. Process také pátrá po volitelném argumentu timeout, jehož význam byl již objasněn dříve. Další argumenty příkazové řádky při spouštění procesu musí odpovídat regulárnímu výrazu $arg[0-9]*$, kde uvedené číslo vyjadřuje jejich pořadí. Jelikož stejný formát se vyskytoval v konfiguračním xml souboru pro argumenty testovaného programu, docházelo při prvních implementacích ke kolizím. Pro řešení tohoto problému se využil následující postup. Hlavní třída testovaného programu jakožto parametr, který nástroj RoadRunner vyžaduje na posledním místě, je uvedena jako *arg70*. Případné argumenty testovaného programu dostávají interně pořadové číslo 71 a výše. Toto řešení by narazilo na problém, pokud by počet parametrů pro RoadRunner přesáhl hodnotu 70, nicméně k takové situaci by v praxi nemělo dojít[3]. Argumenty neodpovídající žádnému výše zmíněnému formátu se do procesu předávají jako proměnné prostředí.

Komunikaci nových tříd a jejich propojení s modulem Executor vyjadřuje sekvenční diagram 4.1.

Obrázek 4.1: Sekvenční diagram popisující návrh komunikace nových tříd a jejich propojení s jádrem SearchBestie



4.3 Úpravy Coverage lib

Knihovna pro tvorbu artefaktů pokrytí Coverage lib obsahovala několik přímých závislostí na ConTest. Pro propojení s nástrojem RoadRunner muselo dojít k jejich odstínění. Prvním řešeným problém byla identifikace jednotlivých testů. ConTest ji generoval sám a nabízel ji v rámci své API. Každému testu bylo přiřazeno *id* ve formě přirozeného čísla, které by mělo být v rámci jednoho spuštění SearchBestie unikátní. RoadRunner žádný podobný mechanismus nenabízel, a proto se v první verzi úprav namísto toho využilo vlastní generování id implementovaného přímo uvnitř knihovny. Pro tyto účely se využívala metoda *System::currentTimeInMillis* vracející počet uplynulých milisekund od 1.1.1970. Následně autor shledal, že tento proces závislý na identifikaci generované uvnitř knihovny je nee-

[2]Nejdříve dochází ke kontrole parametrů a teprve poté se případně zkoumají proměnné prostředí.

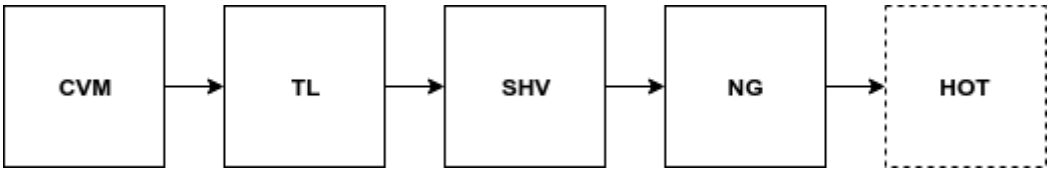
[3]V současnosti dosahuje počet argumentů hodnoty kolem 50 a není důvod předpokládat výrazné zvýšení tohoto čísla. Pokud by k situaci došlo, program se ukončí s chybovým hlášením a správce nástroje může následně tento limit navýšit.

fektivní, a došlo k jeho přesunu přímo do SearchBestie. Také se místo na první pohled náhodných čísel začala generovat posloupnost začínající číslem 1, což zvyšuje přehlednost ve složkách s výstupními soubory.

Hlavní úprava knihovny spočívala ve změně její hierarchie dědičnosti. V rámci dalšího zdokonalování knihovny vzešel požadavek umožnit metriky pokrytí zapínat a vypínat za běhu programu. Byť se tento požadavek týkal hlavně třídy RRPlugin v RoadRunneru, muselo kvůli němu dojít k následujícím úpravám uvnitř knihovny. RoadRunner vyžadoval možnost uchovávat reference na třídy reprezentující jednotlivé metriky v kolekci, ve které by se dalo v případě potřeby metriky přidávat a odebírat. Jelikož vyjma třídy pro metriku GoldiLock všechny ostatní metriky dědily ze společné třídy CoverProducer, stačilo přidat vztah dědičnosti i mezi třídami CoverProducer a Goldilock[4]. Třída CoverProducer nicméně neimplementovala potřebné metody, k jejichž volání nad jednotlivými instancemi subtříd docházelo. Proto bylo nutno je do třídy přidat a metody v subtřídách označit jako redefinované. Jelikož volání těchto nových metod přímo ve třídě CoverProducer je nežádoucí, obsahuje jejich tělo vyvolání výjimky upozorňující na chybné použití objektu. Tato implementace se nejednou osvědčila při dokončování třídy RRPlugin, protože umožnila rychle odhalit chybné volání metod.

4.4 Úpravy a rozšíření RoadRunneru

Většina práce při propojování nástrojů se odehrála právě v RoadRunneru. Jelikož požadavky ze strany SearchBestie byly poměrně pestré, jejich implementace se rozdělila do několika toolů propojených v řetězci označených souhrnně jako *toolchain*. Podoba současného toolchainu je znázorněna na obrázku 4.2. *CVM* vyjadřuje tool jménem *RRPlugin* určený ke sběru dat pro produkci metrik pokrytí. *TL* neboli *ThreadLocalTool* odfiltrovává všechny *thread-local* události, tedy přístupy k proměnným a zámkům pouze z jednoho vlákna. Tool *SHV* implementuje noise placement heuristiky *sharedVars-one*, *all*. Tímto filtrem projdou pouze události, které potenciálně mohou být šumem při dané noise placement heuristice ovlivněny. *NG* neboli *NoiseGenerator* se stará o generování šumu. Konkrétně podporuje noise seeding heuristiky yield, sleep, wait, syncYield, busyWait a mixed. Poslední tool *HOT* implementuje heuristiku HaltOneThread. Do toolchainu se tedy zahrnuje pouze při zapnutí dané heuristiky. Pozorný čtenář jistě zaznamená, že tento návrh postrádá heuristiku TimeoutTampering. Ta je totiž skutečně implementována v jiné části kódu a zapíná se za pomoci přepínače při spouštění RoadRunneru. V následujících subsekcích naleznete podrobnější popis práce na jednotlivých toolech z toolchainu a i další úpravy, které RoadRunner podstoupil.



Obrázek 4.2: Toolchain využívaný při spouštění RoadRunneru

[4]Tento fakt byl sice spojen s určitým plýtváním zdrojů, protože Goldilock dědit z CoverProduceru nepotřeboval, nicméně tato daň zaplacená na výkonu, která se nejspíše téměř neprojevila, vizte srovnávací testy v sekci 4.5, umožnila čistou implementaci metod třídy RRPlugin.

Příprava Toolů

První tool z toolchainu jménem RRPlugin se stará o produkci artefaktů pokrytí. Jeho činnost je silně závislá na knihovně Coverage lib. S využitím tříd z knihovny reprezentuje proměnné, objekty, vlákna a zámky testovaného programu. Po vytvoření této abstraktní reprezentace předá informace do knihovny, která interně produkuje jednotlivé artefakty. Druhý tool TL nebyl vyvinut v rámci této práce. Patří do základní sady toolů poskytovaných RoadRunnerem. Tooly SHV a NG byly z velké části naprogramovány Ing. Zdeňkem Letkem, Ph.D. při jeho experimentech s projektem. Nicméně v rámci práce byly v těchto toolech provedeny určité úpravy jako například odstranění občas se objevujících blokovaných vláken při použití šumu wait se sílou 0[5].

Implementace noise seeding heuristik HaltOneThread a TimeoutTampering

Závěrečný tool HOT implementuje heuristiku HaltOneThread. Oproti ostatním noise seeding heuristikám se jedná o složitější typ šumu. Pseudokód algoritmu implementovaného ve třídě HaltOneThreadTool naleznete níže v ukázce 4.1. Tool si udržuje přehled o všech běžících vláknech pomocí vektoru booleovských proměnných. Hodnota *true* vyjadřuje, že vlákno s PID odpovídajícím indexu ve vektoru je aktivní. Jakmile dojde k zaznamenání libovolné události, hodnota ve vektoru na příslušném indexu se nastaví na *true*. Tool při svém vzniku odstartuje vlákno, které v pravidelných intervalech nastaví všechny hodnoty ve vektoru na *false*. Toto vlákno také jednou za čas vybere náhodně jedno z vláken testovaného programu a uspí ho. V tomto stavu zůstane až do chvíle než vlákno toolu zjistí, že mezi dvěma jeho kontrolami nedošlo k nastavení jediné hodnoty ve vectoru na *true*. Poté je uspané vlákno opět probuzeno.

[5] Volání metody wait s timeoutem nula je ekvivalentní volání metody bez timeoutu, vlákno se už nikdy neprobudí.

Algoritmus 4.1: Noise seeding heuristika HaltOneThread

```
begin class HaltOneThreadTool
        vector knownThreads;

        /**
        * called periodically
        */
        void checkThreads(){
                if(noThreadIsSleeping() and theTimeHasCome()){
                        pickOneThreadAndSuspendIt();
                        return;
                }

                if(noThreadMarkedAsActive() and someThreadIsSleeping())
                        wakeTheSleepingThread();

                markAllThreadsAsInactive();
        }

        /**
        * called when any threadActivity is encountered
        */
        void someThreadActivitySpotted(PID id){
                maskThreadAsActive(id);
        }

        ...
        housekeeping and callback methods
        ...

end class
```

Kromě HaltOneThread bylo potřeba do RoadRunneru doimplementovat i heuristiku TimeoutTampering. Vzhledem k faktu, že metody v rámci API RoadRunneru tool pouze informují o tom, že vlákno bude/bylo uspáno, tato heuristika musela být umístěna přímo do nástroje. Konkrétně se nachází ve třídě *RREventGenerator*, která slouží jako vstupní bod pro všechny callbacky vložené do bytecodu testovaného programu. Pro metody třídy Thread funguje instrumentace tak, že původní volání metody nahradí voláním veřejné statické metody třídy RREventGenerator, v tomto konkrétním případě metody *sleep*. Uvnitř těla metody dojde nejdříve k informování toolů, že dané vlákno bude uspáno, poté se vlákno uspí a na závěr metoda informuje tooly, že vlákno bylo probuzeno. Pro implementování timeoutTamperingu stačilo obalit skutečné volání Thread::sleep uvnitř RREventGenerator::sleep podmínkou ovlivnitelnou přes přepínač a volání povolit pouze pokud je metrika vypnutá. Jednalo se tedy spíše o problém nalezení vhodného místa v kódu, kam metriku začlenit, samotná implementace již byla rychlá.

Informace o polích

Na první pohled čistá a přehledná API nástroje RoadRunner skrývala jedno temné zákoutí. Jednalo se o přístupy k polím. RoadRunner pro tyto účely neposkytoval žádnou speciální metodu. Rozlišoval pouze metody *access* a *volatileAccess* vyvolané při přístupu k volatilním a nevolatilním proměnným. Do metod se jako parametr předával objekt subtřídy odvozené z třídy *AccessEvent*, který v atributu *target* obsahoval referenci na objekt. Konkrétně docházelo k předáváním dvou typů objektů, a to *FieldAccessEvent* a *ArrayAccessEvent*. Význam atributu *target* závisel právě na typu proměnné, ke které se přistupovalo. V případě obyčejného atributu obsahoval referenci na objekt, k jehož atributu se právě přistupovalo. V případě přístupu k poli ale obsahoval místo toho referenci na pole jako takové. Coverage lib ovšem potřebuje pro identifikaci referenci na objekt, jemuž daný atribut patří, a také jméno atributu. *FieldAccessEvent* obě tyto informace poskytoval. V případě *ArrayAccessEvent* nicméně nebyla k dispozici ani jedna. Pro odhalení důvodů bylo potřeba jít velmi hluboko až k samotné instrumentaci bytecodu uvnitř nástroje. Při přístupu do pole totiž není reference na objekt již potřeba, stačí reference na pole jako taková. Implementovat mechanismus přístupu k poli výše uvedeným způsobem byla tedy příjemná zjednodušující cesta.

Pro řešení této situace došlo k přidání nové metody *beforeArrayAccess* do abstraktní třídy Tool a do třídy RREventGenerator. Dále bylo třeba upravit instrumentaci metod pro práci s poli a vložit do bytecodu nové instrukce pro získání reference na objekt, k jehož atributu typu pole se bude přistupovat. Následně byla do bytecodu vložena instrukce pro volání *RREventGenerator::beforeArrayAccess*. Přes tuto metodu se k toolům dostanou stejné informace jako v případě přístupu ke klasickému atributu.

Informace o výjimkách

Původní implementace RoadRunneru řešila situaci, kdy testovaný program havaroval vlivem neošetřené výjimky, vypsáním chybové hlášky a ukončením programu. Tato technika nicméně neumožňovala toolům, aby mohly dokončit rozpracovanou práci a uložit výstupy. Jelikož například u toolu CVM je uložení výstupů i při pádu aplikace nezbytné, došlo k úpravě RoadRunneru. Do abstraktní třídy tool byla přidána nová metoda *killedByException*, která tool informuje o ukončení aplikace a přes parametr mu předá výjimku, která pád způsobila. Původní ukončovací schéma bylo rozšířeno o volání této nové metody před ukončením RoadRunneru.

Další úpravy nástroje RoadRunner

V rámci implementace došlo také k dalším drobným úpravám nástroje. První změna zahrnovala rozšíření mechanismu předávání booleovských argumentů přes příkazovou řádku. V rámci definice booleovského argumentu mu byla přiřazena implicitní hodnota. Při spuštění RoadRunneru s uvedeným argumentem ve formátu *-jméno* se tato hodnota invertovala. Původní varianta neumožňovala předat parametr v klasickém formátu *-jméno=hodnota*. Vzhledem k faktu, že ze strany SearchBestie se tento formát generoval mnohem snadněji[6], byla jeho podpora do nástroje doplněna.

[6] Důvodem bylo, že formát *-jméno=hodnota* umožňoval nastavit novou hodnotu bez nutnosti znát implicitní nastavení přepínače. Bylo by tedy potřeba procházet všechny parametry a určovat v závislosti na jejich implicitní hodnotě, zda má daný přepínač SB uvádět či nikoliv.

Název programu	Velikost	Typ chyby
Airlines	0.2 kLOC, 3 třídy	časově závislá chyba nad daty, porušení atomicity
Cache4j	1.7 kLOC, 66 tříd	žádná známá chyba
Crawler	1.2 kLOC, 19 tříd	porušení atomicity
Elevator	1.2 kLOC, 12 tříd	časově závislá chyba nad daty, porušení atomicity
HEDC	12.7 kLOC, 747 tříd	žádná známá chyba
Rover	5.4 kLOC, 82 tříd	porušení atomicity, uváznutí

Tabulka 4.1: Výčet testovacích programů využitých při srovnávání ConTestu a RoadRunneru

V rámci další změny se zavedl méně striktní způsob zpracování argumentů. Původní implementace RoadRunneru při setkání s neznámým přepínačem ukončila nástroj s chybovým hlášením. Pro spuštění z příkazové řádky se jedná o logické a smysluplné omezení. Nicméně z důvodu co nejlepší zpětné kompatibility konfiguračních souborů[7] došlo k zavedení méně striktního systému, kdy RoadRunner pouze na neznámý přepínač upozorní, ale neukončí svoji činnost.

4.5 Srovnávací testy RoadRunneru a ConTestu

Po dokončení implementace propojení nastala fáze testování. Primárním cílem bylo ověřit, zda toto propojení mezi nástroji SearchBestie a RoadRunner funguje správně, aby bylo možno tuto kombinaci začít používat na další experimenty. Sekundárním cílem bylo zjistit, jak si nástroj RoadRunner vede ve srovnání s nástrojem ConTest. Tyto testy proběhly na sérii testovacích programů z repozitáře VeriFITu. Jednalo se o šest programů různých velikostí obsahujících hlavně časově závislé chyby nad daty, porušení atomicity a v některých případech i uváznutí. Přehled těchto programů naleznete v tabulce 4.1.

Pojďme uvedené programy prozkoumat důkladněji[8].

- Airlines je jednoduchý testovací případ simulující portál pro rezervaci letenek v letadle. Práce s proměnnou vyjadřující počet zabraných míst není chráněna žádným způsobem synchronizace, takže může dojít k rezervaci většího počtu míst, než odpovídá skutečné kapacitě letadla.

- Cache4j je program simulující cache paměť typu LRU. Implementace je synchronizována pomocí jednoho globálního zámku.

- Crawler je program poskytnutý IBM. Jedná se o jádro programu zasazené do testovacího prostředí simulující skutečné užití produktu. Systém vytváří skupinu vláken, která postupně obsluhují přicházející požadavky. Chyba v programu se objevuje při ukončovací sekvenci, ale jenom velice zřídka (přibližně 15 krát z 10000 běhů).

- Elevator obsahuje *real-time* diskrétní simulaci. Jednotlivá vlákna reprezentují výtahy, které komunikují s centrální řídicí deskou za pomoci synchronizace s využitím zámků.

- HEDC reprezentuje jádro aplikace implementující paralelní prohledávání internetových archivů. Jedná se o největší použitý testovací program.

[7] V původních konfiguračních souborech pro ConTest se vyskytovalo velké množství parametrů, které RoadRunner nevyžaduje. Jejich přepisování by bylo časově náročné.

[8] Pro bližší informace navštivte http://www.fit.vutbr.cz/research/groups/verifit/benchmarks/

- Rover je experimentální platforma od NASA navržená pro autonomní řízení vozidel při zkoumání povrchu planety Mars.

Testy se dají rozdělit do dvou fází. V první se spustilo testování s využitím náhodného prohledávání stavového prostoru a omezením na 1000 iterací. Pro druhou fázi zaměřující se na přesnější srovnání nástrojů došlo k výběru fixních 20-ti konfigurací a každá z nich byla prozkoumána 100krát. Jednalo se o prvních 20 konfigurací prozkoumaných při náhodném prohledávání v prvním testu. Cílem tohoto druhého testu bylo zjistit, zda nástroje při stejné konfiguraci budou podobně úspěšné. V následujících tabulkách naleznete výsledky těchto experimentů. Pro každý benchmark jsou uvedeny testy obou nástrojů. Lepší výsledek je zobrazen tučně. Hodnotila se tři kritéria, doba běhu testu, směrodatná odchylka od doby běhu testu a počet nalezených chyb. Jako chyba je vyhodnocena situace, kdy se program ukončí vlivem neošetřené výjimky nebo když dojde k překročení maximální doby vyhrazené na jeden testovací běh. Tento případ se týká hlavně testovacích případů, ve kterých se nachází chyba typu uváznutí. Pokud některý běh testu mnohonásobně překračuje obvyklou dobu běhu, situace je vyhodnocena jako uváznutí. V tabulce 4.2 naleznete výsledky testů při prozkoumávání 1000 náhodných konfigurací. RoadRunner výrazně překonal ConTest u benchmarku Airlines, a to jak v počtu chyb, tak i v rychlosti běhu. Na druhou stranu u benchmarků Elevator a Rover naopak v počtu chyb zaostává. V době běhu programu je ale RoadRunner lepší u všech benchmarků vyjma HEDC. V následující tabulce 4.3 se nachází výsledky pro 20 fixních konfigurací. I zde si RoadRunner vedl výrazně lépe u benchmarku Airlines, ale naopak hůře u benchmarků Elevator a Rover. Testy byly spuštěny vícekrát, ale výše popsaný vzor se mírnými rozdíly pravidelně opakoval.

Rozdílné výsledky ConTestu a RoadRunneru mohou být například důsledkem rozdílných technologií použitých na instrumentaci bytecodu a také celkově rozdílnou implementací. Contest byl navržen přímo na detekci chyb ve vícevláknových programech. Oproti tomu RoadRunner je nástroj obecnější umožňující vytvářet různé dynamické analyzátory. Dále je také nutné vzít v potaz nedeterminismus vícevláknových programů. Detekce chyby s využitím testování je stále do určité míry věcí náhody.

Testy prokázaly, že původní myšlenka na propojení nástrojů SearchBestie a RoadRunner byla validní. Toto propojení lze vyhodnotit jako fungující. Díky tomuto kroku byla odstraněna závislost na nástroji ConTest, což otevírá dveře novým možnostem v oblasti výzkumu a případně i nasazení nástrojů v praxi.

Benchmark	Nástroj	Doba běhu [ms]	Odchylka	Počet chyb
Airlines	RoadRunner	**2 483**	**42**	**542**
	ConTest	6 996	90	119
Cache4J	RoadRunner	**116 662**	**340**	0
	ConTest	163 570	372	0
Crawler	RoadRunner	**2 786**	**41**	0
	ConTest	2 839	48	0
Elevator	RoadRunner	**9 595**	84	0
	ConTest	10 947	**60**	**14**
HEDC	RoadRunner	13 688	**41**	0
	ConTest	**5 777**	78	0
Rover	RoadRunner	**7 930**	**46**	10
	ConTest	16 440	115	**102**

Tabulka 4.2: Tisíc náhodně vybraných stavů

Benchmark	Nástroj	Doba běhu [ms]	Odchylka	Počet chyb
Airlines	RoadRunner	**2 635**	**42**	**1 096**
	ConTest	8 945	109	275
Cache4J	RoadRunner	61 465	288	0
	ConTest	**32 805**	**233**	0
Crawler	RoadRunner	**2 736**	**40**	0
	ConTest	2 739	48	**2**
Elevator	RoadRunner	**8 004**	76	0
	ConTest	11 436	**59**	**16**
HEDC	RoadRunner	13 609	**38**	0
	ConTest	**9 246**	94	0
Rover	RoadRunner	**7 332**	**39**	0
	ConTest	11 654	84	**100**

Tabulka 4.3: Dvacet zvolených stavů

Kapitola 5

Návrh a implementace nových heuristik

Tato kapitola se zabývá návrhem a implementací přesných heuristik pro vkládání šumu. V sekci 5.1 naleznete seznam heuristik, které byly v rámci této práce vyvinuty. V sekci 5.2 se nachází abstraktní návrh řešení implementace těchto heuristik. Sekce 5.3 popisuje již konkrétní změny provedené v nástroji RoadRunner, který nyní mimo jiné umožňuje vypsat seznam všech míst, která instrumentoval, a to na čtyřech různých úrovních abstrakce. Sekce 5.4 popisuje změny provedené v nástroji SearchBestie. Sekce 5.5 poskytuje experimentální srovnání nových heuristik s původními založenými na náhodnosti.

5.1 Nově navržené noise placement heuristiky

Původní noise placement heuristiky nabízejí pouze omezené možnosti, jak kontrolovat vkládání šumu. Rozhodovací proces navíc využívá pravděpodobnosti a generování pseudonáhodných čísel. Nelze tedy přesně určit množinu míst, na která má být šum vložen. Navrhnout a naimplementovat nové heuristiky je hlavním cílem této práce. Rozdíl oproti původním heuristikám tkví v tom, že nové heuristiky umožňují přesně určit, která místa budou šumem ovlivněna. Jejich přehled naleznete v tabulce 5.1. První heuristikou je *perProgLoc*, jejíž činnost lze popsat následovně. Nejdříve se zjistí všechna místa testovaného programu, která jsou vhodná pro ovlivnění šumem. V rámci konfigurace testu se následně určí, která konkrétní místa budou skutečně ovlivněna. Další heuristikou je *perObject*, při které dochází ke specifikaci na vyšší úrovni abstrakce. Zde již tester nevolí jednotlivá místa v kódu, ale konkrétní sadu proměnných a zámků. O trochu abstraktnější heuristika *perClass* umožňuje specifikovat pouze určité třídy, jejichž instance budou šumem ovlivněny. Nejabstraktnější heuristikou je *perEventSet*, kdy dochází k ovlivnění pouze určitého typu události či událostí, například pouze přístupy ke sdíleným proměnným či synchronizace s využitím zámků. Všechny výše uvedené heuristiky lze parametrizovat frekvencí šumu.

5.2 Návrh implementace

Po analýze mechanismů pro instrumentaci bytecodu došlo k následujícímu návrhu. Do nástroje RoadRunner byl přidán nový přepínač, při jehož aktivaci nástroj do souboru *locations.progloc* vypíše seznam všech míst, která instrumentoval. Konkrétní heuristiku určí hodnota parametru tohoto přepínače. Před spuštěním SearchBestie se takto nejdříve získá

Heuristika	Úroveň abstrakce	Počet možných nastavení
perProgLoc	místa v kódu	nejvyšší
perObject	proměnné a zámky	střední-více
perClass	třídy testované aplikace	střední-méně
perEventSet	události	nejmenší

Tabulka 5.1: Výčet nově navržených heuristik pro vkládání šumu

seznam všech míst, kam lze potenciálně vložit šum. Každé místo se reprezentuje jako jedna booleovská dimenze ve stavovém prostoru, kde hodnota *true* odpovídá vložení šumu. Třída RoadRunnerTask bude upravena, aby při spuštění RoadRunneru přes speciální textový soubor předala seznam míst k ovlivnění šumem. Cesta k souboru a heuristika se specifikuje s využitím přepínače. Do toolchainu RoadRunneru se vloží nový filtrovací nástroj, který dále propustí pouze události odpovídající obsahu tohoto souboru. Vizualizaci tohoto návrhu pro heuristiku *perProgLoc* naleznete na obrázku 5.1.

Pojďme se na tento obrázek zaměřit detailněji. Nejdříve dojde k vygenerování souboru se seznamem všech míst v testovaném programu. Tento soubor vygeneruje RoadRunner a v obrázku je identifikován jako *allLocations.progloc* a naleznete jej vpravo nahoře. Informace z tohoto souboru jsou poté uloženy do konfiguračního souboru pro SearchBestie *config.xml* jako booleovské dimenze, jak můžete vidět vlevo nahoře. Tato konfigurace je načtena modulem *Manager* do SearchBestie, kde jsou dimenze uloženy do stavového prostoru v modulu *Storage*. Před spuštěním testu je konfigurace předána třídě *RoadRunnerTask*, která místa v programu, co mají být ovlivněna šumem, uloží do souboru znázorněného v obrázku jako *toInsert.progloc*. Poté dojde ke spuštění nástroje RoadRunner a jeho filtrovací nástroj *LocationFilterTool* tento soubor načte a na základě něj filtruje pouze události, které mají být šumem ovlivněny.

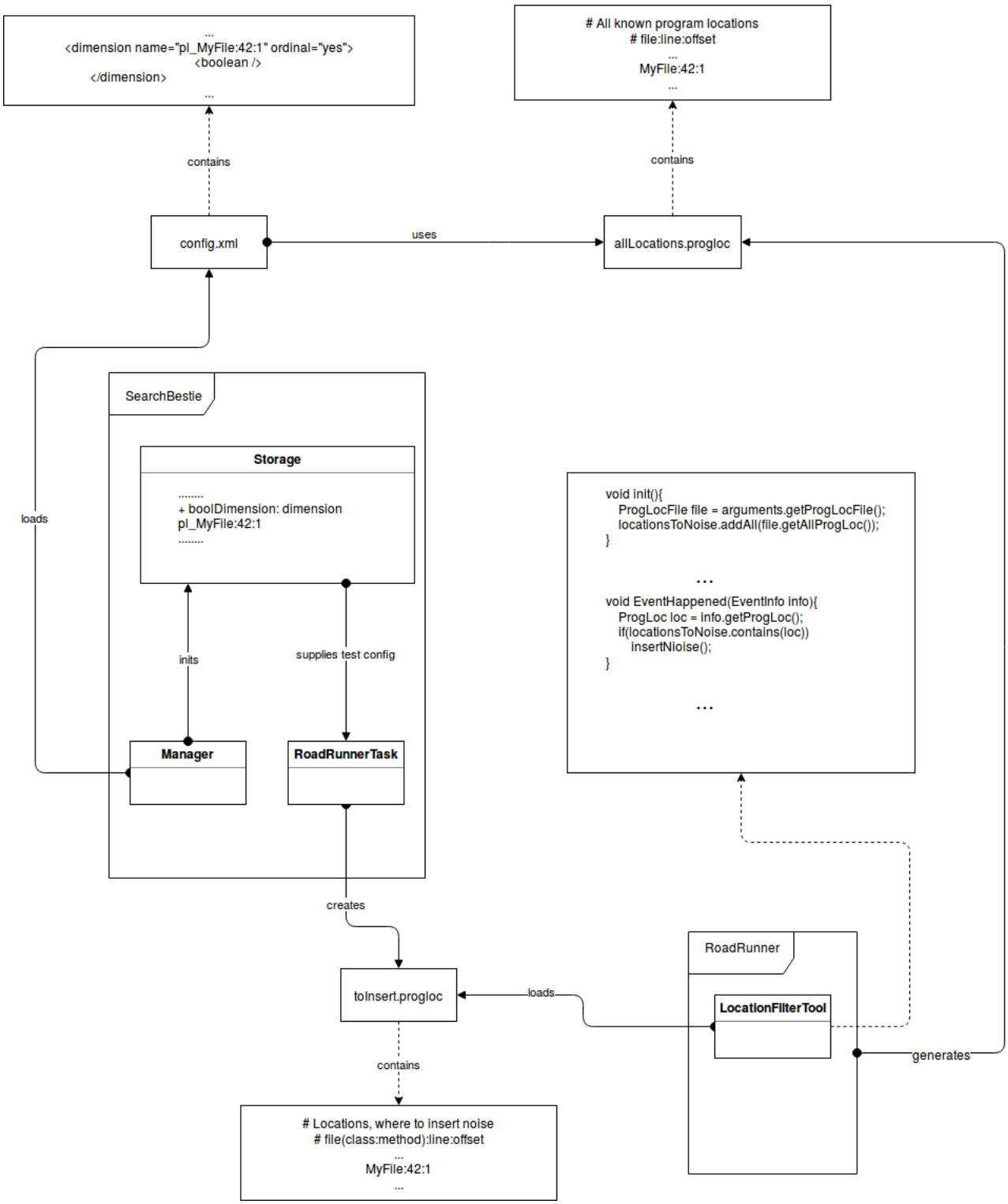
5.3 Úpravy v nástroji RoadRunner

Úpravy v nástroji RoadRunner se dají rozdělit do dvou samostatných fází. První fáze zahrnovala upravení tříd provádějících instrumentaci bytecodu tak, aby seznam všech instrumentovaných míst uložily do souboru *locations.progloc*. Druhá fáze obsahovala implementaci tříd umožňujících přesné vkládání šumu do testovaného programu.

5.3.1 Sběr informací o instrumentovaných třídách

Pro sběr informací byla vytvořena nová třída *ProglocCollector*, jejíž činnost zapíná přepínač *collectProgloc*. Tento přepínač je výčtového datového typu, jež nabývá hodnot dle dříve definovaných přesných noise placement heuristik pro vkládání šumu prezentovaných v sekci 5.1[1]. Třída obsahuje metody volané při instrumentaci přístupu k proměnným a metodám testovaného programu. Interně si udržuje seznam všech zaznamenaných míst v kolekci. Po skončení běhu programu se tento seznam uloží do souboru.

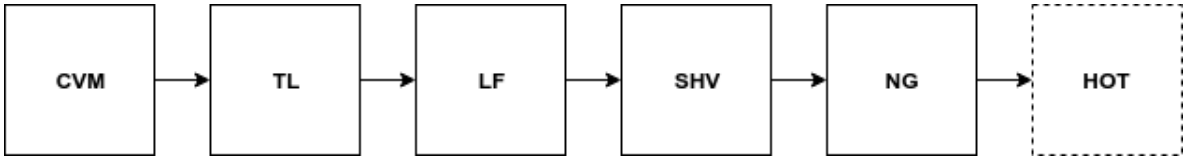
[1] Kromě heuristiky perEventSet, k jejíž činnosti není potřeba žádné informace sbírat, neboť tato heuristika cílí vždy na určitý typ událostí, které již v API RoadRunneru jsou poskytovány.



Obrázek 5.1: Návrh komunikace pro heuristiku perProgLoc

5.3.2 Implementace přesnějšího vkládání šumu

Původní plán implementace přesnějšího vkládání šumu zahrnoval *parciální instrumentaci*. Při této technice dochází k ovlivnění pouze předem daných míst v kódu. Tato metoda se jevila jako efektivní, neboť v místech mimo toto určení nedocházelo ke zbytečným volání metod analyzátoru. Nicméně od tohoto nápadu se nakonec upustilo z důvodu, že některé nástroje jako například *RRPlugin* potřebují informace o všech událostech, jinak nemohou produkovat všechny artefakty pokrytí. Ve výsledku by tedy stavy s malým počtem instrumentovaných míst mohly při fitness funkci zaměřené právě na množství artefaktů pokrytí působit nezajímavě, a to i v případě, kdy by byla instrumentována přesně ta správná místa k potenciálnímu odhalení chyby. Nový návrh tedy přesnější vkládání šumu zahrnul do nového toolu *LocationFilterTool*, zkráceně *LF*, který se začlenil do toolchainu RoadRunneru mezi tooly pro sběr dat a tooly pro vkládání šumu. Konkrétní pořadí toolů naleznete na obrázku 5.2.



Obrázek 5.2: Toolchain využívaný při spouštění RoadRunneru s přesným vkládáním šumu

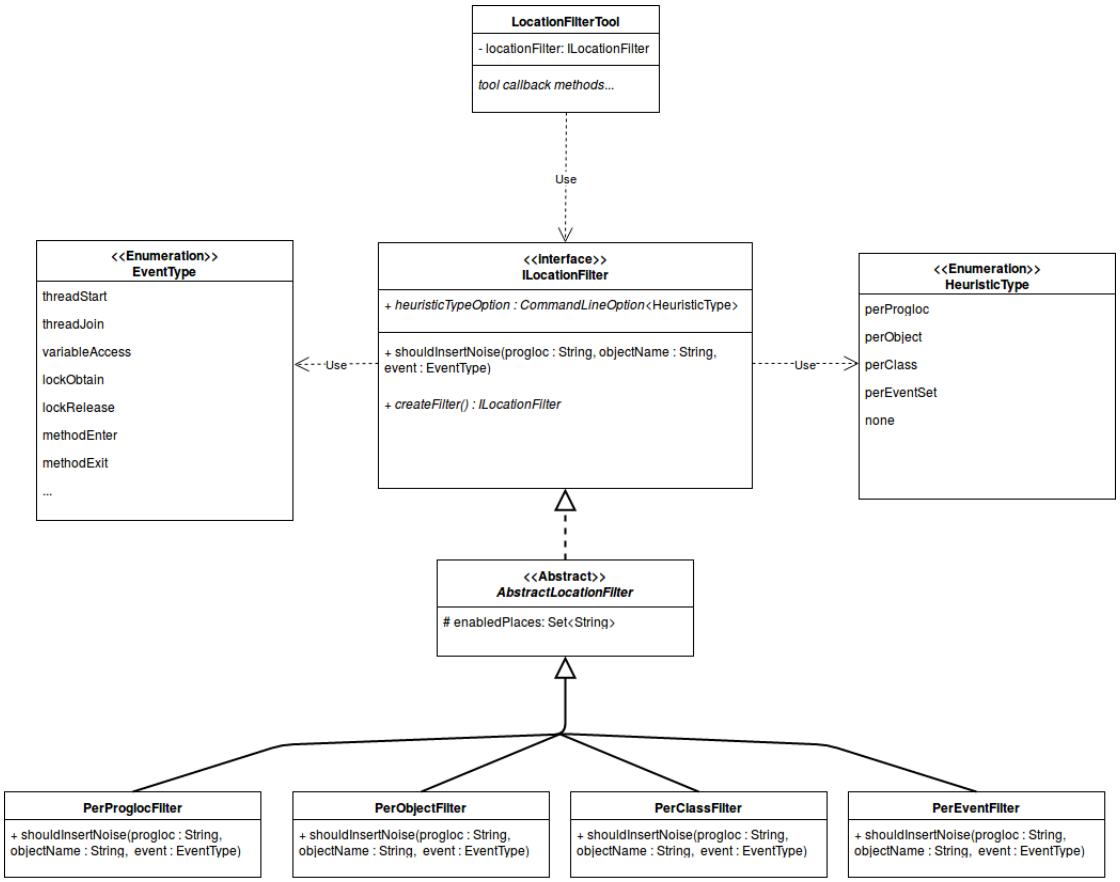
Vizualizaci návrhu pro přesné vkládání šumu naleznete v diagramu tříd 5.3. Z abstraktní třídy *AbstractLocationFilter* dědí čtyři konkrétní třídy filtrující místa v programu, každá pro jednu heuristiku. Tato abstraktní třída implementuje rozhraní *ILocationFilter* a její tovární metoda *createFilter* vrací jeden ze čtyř konkrétních filtrů v závislosti na nastavení přepínače RoadRunneru *heuristicTypeOption*. *ILocationFilter* dále využívá dvou pomocných výčtových typů. *HeuristicType* reprezentuje jednotlivé heuristiky a *EventType* reprezentuje typy událostí, na základě kterých lze filtrovat při využití heuristiky perEventSet.

5.4 Úpravy v nástroji SearchBestie

Díky generickému návrhu nástroje se úpravy uvnitř SearchBestie zredukovaly na zásah do jediné třídy RoadRunnerTask. Před startem SearchBestie nejdříve dojde ke spuštění skriptu, který spustí nástroj RoadRunner pro získání seznamu míst v testovaném programu a tato místa poté vloží do konfiguračního xml souboru jako nové booleovské dimenze. Pro odlišení od ostatních budou mít všechny tyto dimenze prefix *pl_*. Třída RoadRunnerTask v rámci metody *performTask* hodnotu všech dimenzí typu místo v programu s hodnotou true uloží do souboru *locations.progloc*, ze kterého je poté LocationFilterTool v nástroji RoadRunner načte.

5.5 Testování nových heuristik

V této sekci naleznete informace o srovnávacích testech mezi nově navrženými heuristikami pro přesné vkládání šumu a původními heuristikami využívajícími náhodnosti. Pro ověření byly využity testovací programy Airlines, Crawler, Elevator a Rover, jejichž stručnou charakteristiku naleznete v sekci 4.5. Tyto čtyři programy byly z výše uvedených programů vybrány pro jejich relativně krátkou dobu běhu, což umožnilo zrychlit testovací proces. Pro

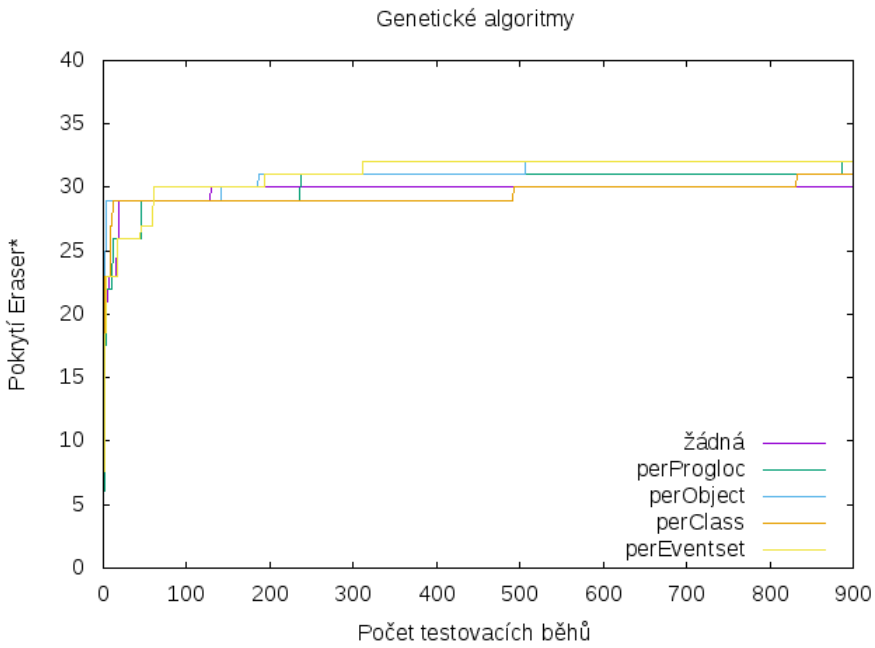


Obrázek 5.3: Digram tříd pro implementaci přesnějšího vkládání šumu

Heuristika	Algoritmus	Doba běhu [ms]	Odchylka	Počet chyb	Eraser*
žádná	ec	**17 171**	**59**	**152**	**30**
	rnd	17 281	63	103	29
perProgLoc	ec	**11 856**	**41**	**711**	32
	rnd	27 633	130	145	32
perObject	ec	**11 338**	**35**	**743**	32
	rnd	26 012	135	353	32
perClass	ec	**11 530**	**36**	**765**	31
	rnd	26 574	129	208	31
perEventSet	ec	**11 615**	**38**	**724**	**32**
	rnd	26 916	131	145	31

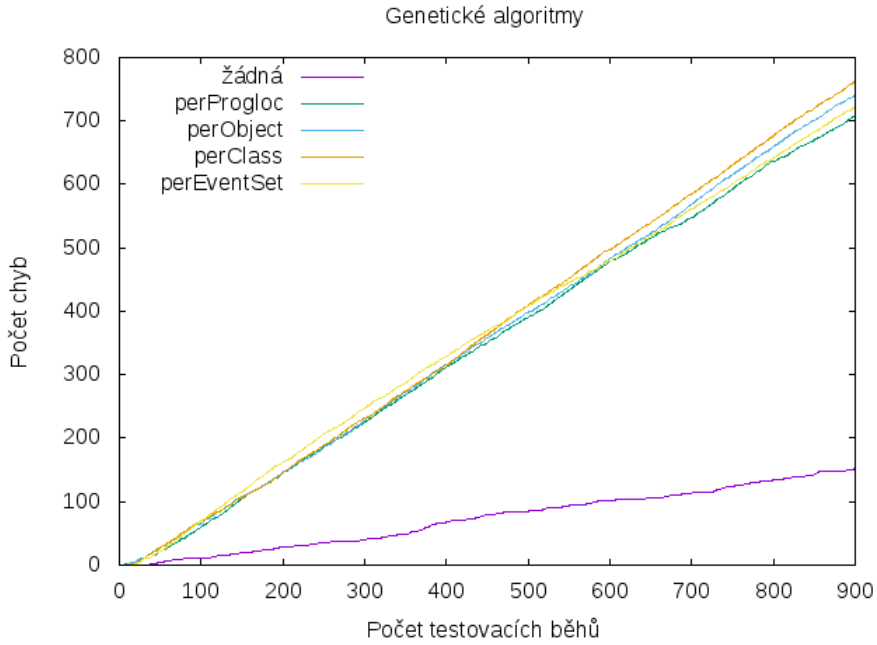
Tabulka 5.2: Srovnání na testovacím programu Airlines

každý program bylo spuštěno 10 konfigurací. Ke čtyřem nově navrženým heuristikám byla spuštěna i jedna konfigurace bez přesných heuristik. Těchto 5 konfigurací bylo spuštěno se dvěma různými algoritmy prohledávání stavového prostoru, s náhodným prohledáváním a genetickými algoritmy. Z mnoha možných konfigurací genetických algoritmů byla zvolena verze SOGA, tedy Single Objective Genetic Function, s 50-ti generacemi o 20-ti jedincích. Toto nastavení bylo již dříve používáno při experimentech například v [7]. Celkově tedy bylo spuštěno 40 konfigurací. Pro každý testovací program se v této sekci nachází tabulka zobrazující průměrnou délku běhu, odchylku od ní, počet běhů, kdy došlo k odhalení chyby, a kumulovaný počet artefaktů metriky Eraser*.

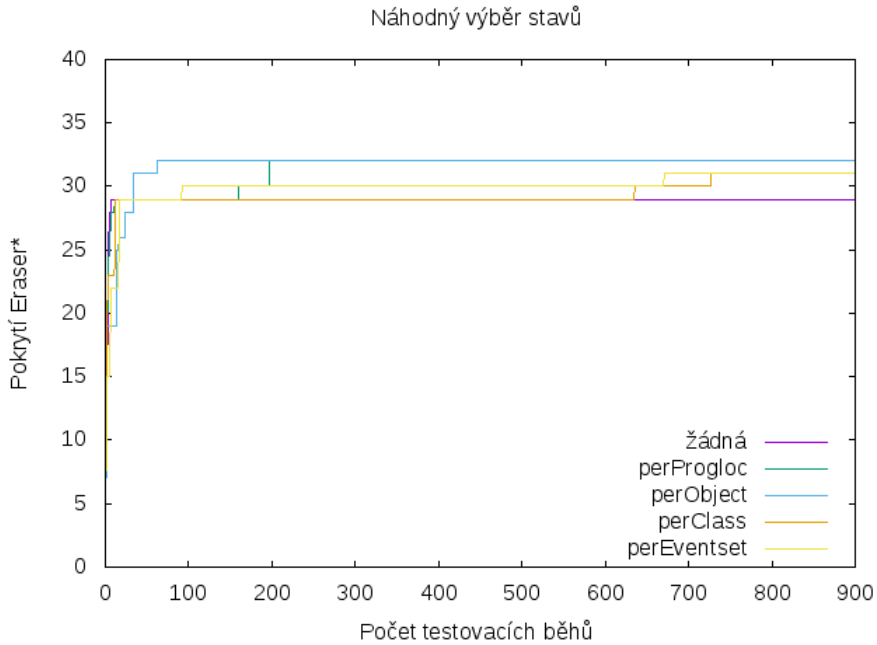


Obrázek 5.4: Metrika Eraser* na testovacím programu Airlines s genetickými algoritmy

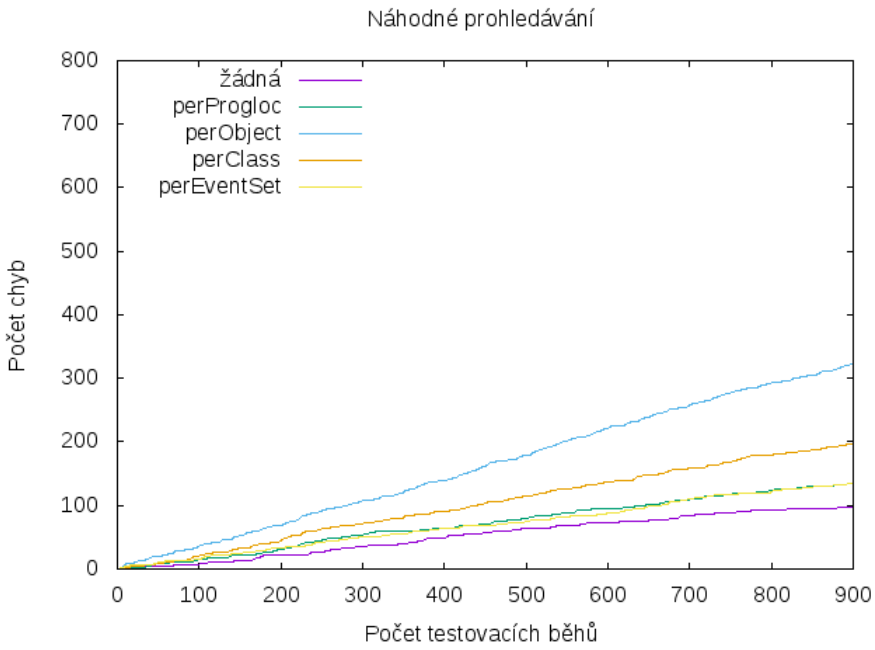
V tabulce 5.2 a v grafech 5.4, 5.6, 5.5 a 5.7 naleznete výsledky testů na testovacím programu Airlines. Z tabulky lze vyčíst, že genetické algoritmy překonaly náhodné prohledá-



Obrázek 5.5: Počet chyb objevených při testování programu Airlines s genetickými algoritmy



Obrázek 5.6: Metrika Eraser* na testovacím programu Airlines s náhodným prohledáváním



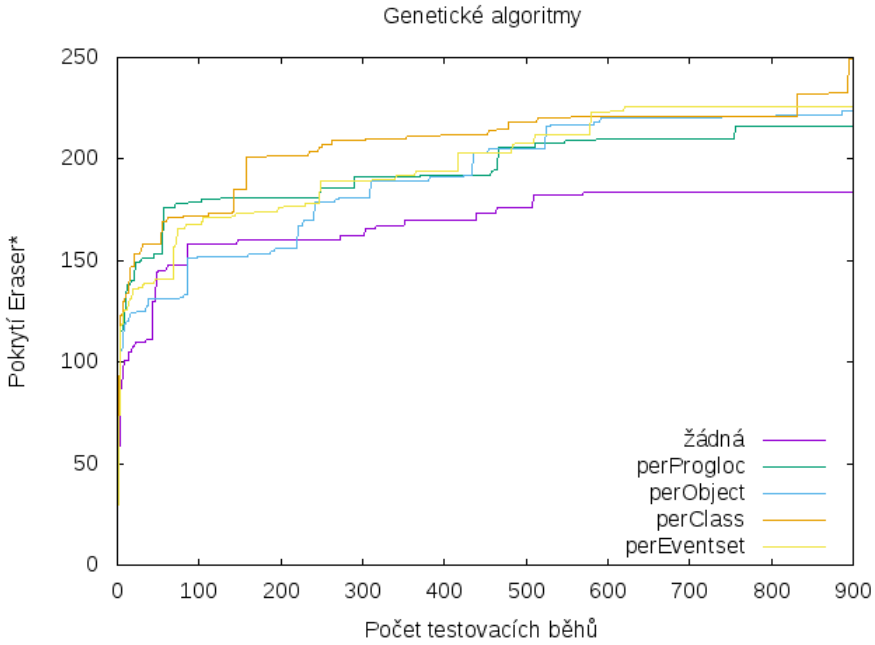
Obrázek 5.7: Počet chyb objevených při testování programu Airlines s náhodným prohledáváním

vání, a to velmi významně. Můžete také zaznamenat nečekaný výsledek ohledně délky běhu, kdy genetické algoritmy opět překonaly náhodné prohledávání. V oblasti kumulovaného pokrytí dosahovaly všechny testy podobných výsledků, jak je patrno hlavně z grafů. Nejvíce chyb bylo nalezeno při kombinaci heuristiky perClass s genetickými algoritmy. Všechny nově navržené heuristiky byly v počtu nalezených chyb výrazně lepší než původní implementace.

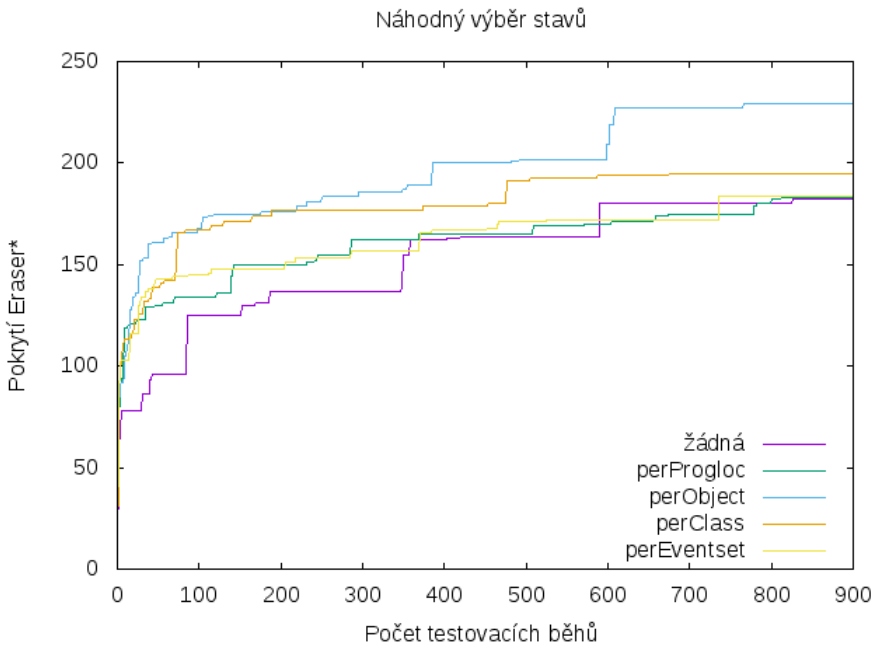
U testovacího programu Crawler nedošlo k odhalení žádných chyb, proto lze porovnávat pouze dobu běhu programu a metriky pokrytí. V rychlosti opět genetické algoritmy překonaly náhodné prohledávání. Genetické algoritmy dosáhly lepší kumulované metriky pokrytí ve všech případech kromě metriky perObject. Nejvyšší kumulované pokrytí dosáhla heuristika perClass v kombinaci s genetickými algoritmy. Všechny nově navržené heuristiky byly

Heuristika	Algoritmus	Doba běhu [ms]	Odchylka	Počet chyb	Eraser*
žádná	ec	**18 182**	56	0	**184**
	rnd	19 195	56	0	182
perProgLoc	ec	**14 667**	**34**	0	**216**
	rnd	17 390	49	0	183
perObject	ec	**14 570**	**34**	0	224
	rnd	16 163	45	0	**235**
perClass	ec	**14 796**	**34**	0	**249**
	rnd	17 249	53	0	200
perEventSet	ec	**14 535**	**34**	0	**226**
	rnd	17 689	52	0	187

Tabulka 5.3: Srovnání na testovacím programu Crawler



Obrázek 5.8: Metrika Eraser* na testovacím programu Crawler s genetickými algoritmy

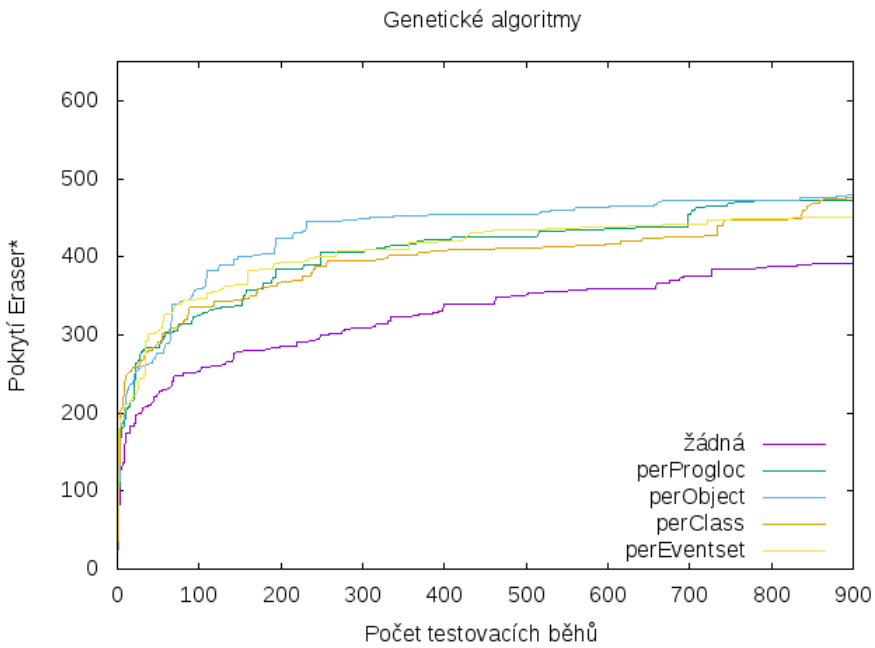


Obrázek 5.9: Metrika Eraser* na testovacím programu Crawler s náhodným prohledáváním

Heuristika	Algoritmus	Doba běhu [ms]	Odchylka	Počet chyb	Eraser*
žádná	ec	21 580	80	0	**392**
	rnd	**20 898**	**79**	0	370
perProgLoc	ec	21 114	**72**	0	473
	rnd	**20 615**	81	0	**633**
perObject	ec	**20 185**	**78**	0	**479**
	rnd	22 302	91	0	437
perClass	ec	22 238	**75**	0	**475**
	rnd	**21 431**	87	0	452
perEventSet	ec	21 035	**72**	0	451
	rnd	**20 972**	84	0	**519**

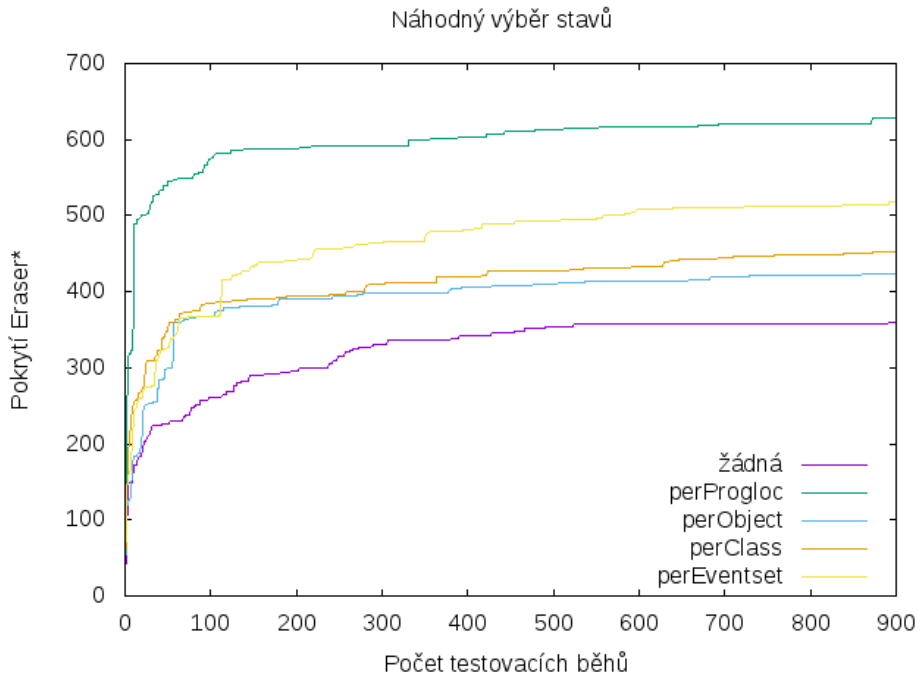
Tabulka 5.4: Srovnání na testovacím programu Elevator

v počtu chyb i kumulovaném pokrytí lepší než původní implementace. Data můžete nalézt v tabulce 5.3 a grafech 5.8 a 5.9.



Obrázek 5.10: Metrika Eraser* na testovacím programu Elevator s genetickými algoritmy

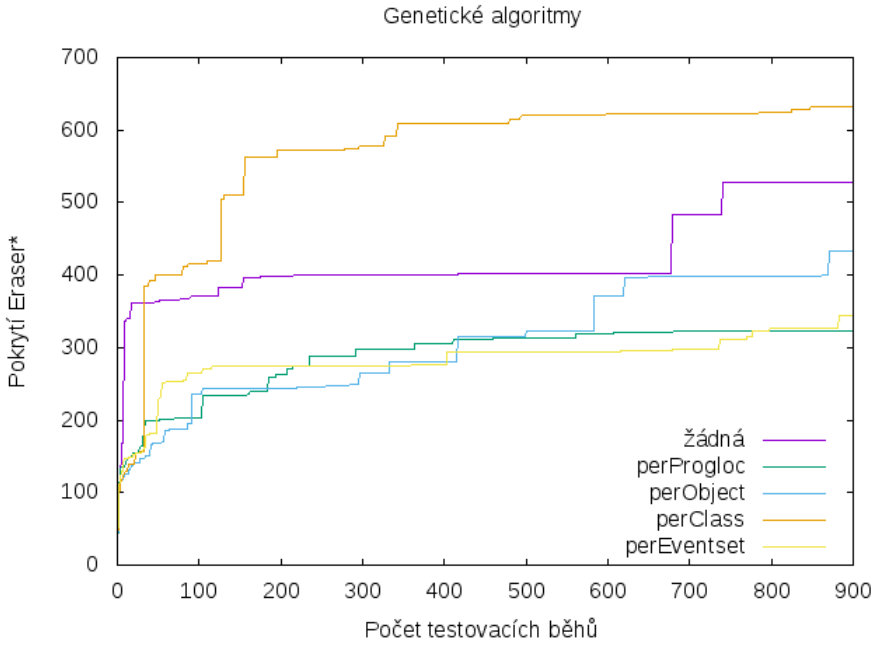
Stejně jako v předchozím případě, ani u testovacího případu Elevator nedošlo k odhalení žádných chyb. Kratší průměrnou dobu běhu programu mělo náhodné prohledávání ve všech metrikách kromě perObject. Kumulované pokrytí zde dosahovalo v jednotlivých metrikách výrazně rozdílných hodnot. I u tohoto testovacího případu lze říci, že nově navržené heuristiky překonaly původní implementaci. U heuristik perObject a perClass dosáhly lepšího kumulovaného pokrytí genetické algoritmy, naopak u perProgLoc a perEventSet dosáhlo lepšího výsledku náhodné prohledávání. V kumulovaném počtu artefaktů opět nově navržené heuristiky překonaly původní implementaci, nejlepšího výsledku dosáhla heuristika perProgLoc. Data jsou k dispozici v tabulce 5.4 a grafech 5.10 a 5.11.



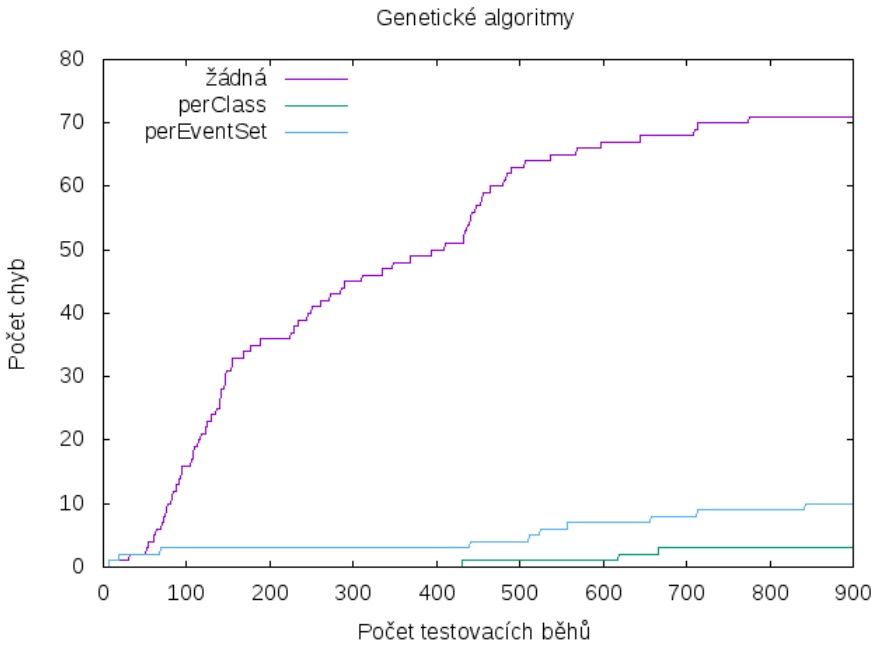
Obrázek 5.11: Metrika Eraser* na testovacím programu Elevator s náhodným prohledáváním

Heuristika	Algoritmus	Doba běhu [ms]	Odchylka	Počet chyb	Eraser*
žádná	ec	23 839	**59**	**71**	**527**
	rnd	**22 799**	63	58	480
perProgLoc	ec	**20 283**	**48**	0	323
	rnd	22 053	61	0	**419**
perObject	ec	19 954	**42**	0	**434**
	rnd	**19 370**	47	0	337
perClass	ec	**21 274**	**48**	3	632
	rnd	21 818	61	**13**	**691**
perEventSet	ec	**21 062**	**51**	**10**	345
	rnd	21 668	60	5	**625**

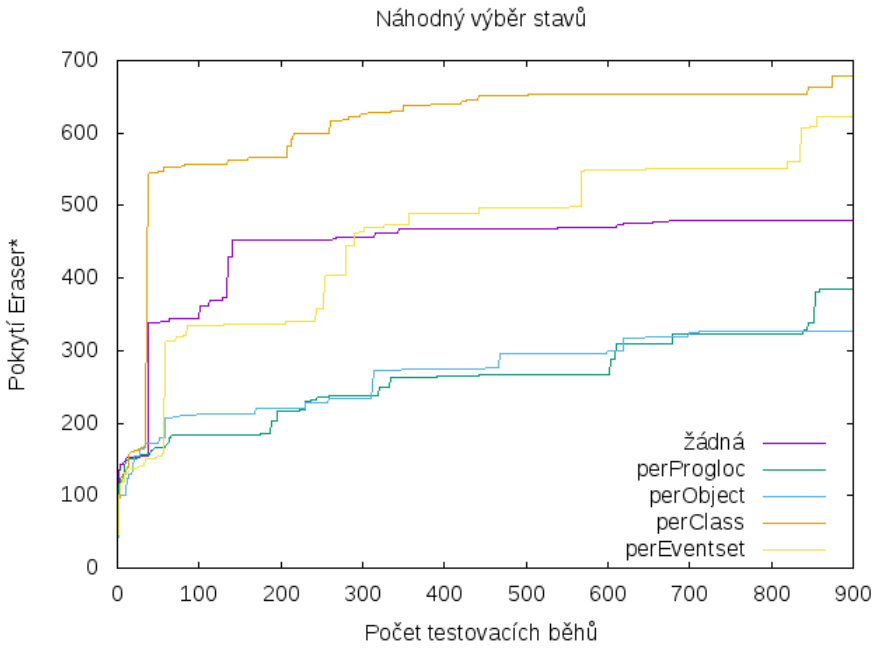
Tabulka 5.5: Srovnání na testovacím programu Rover



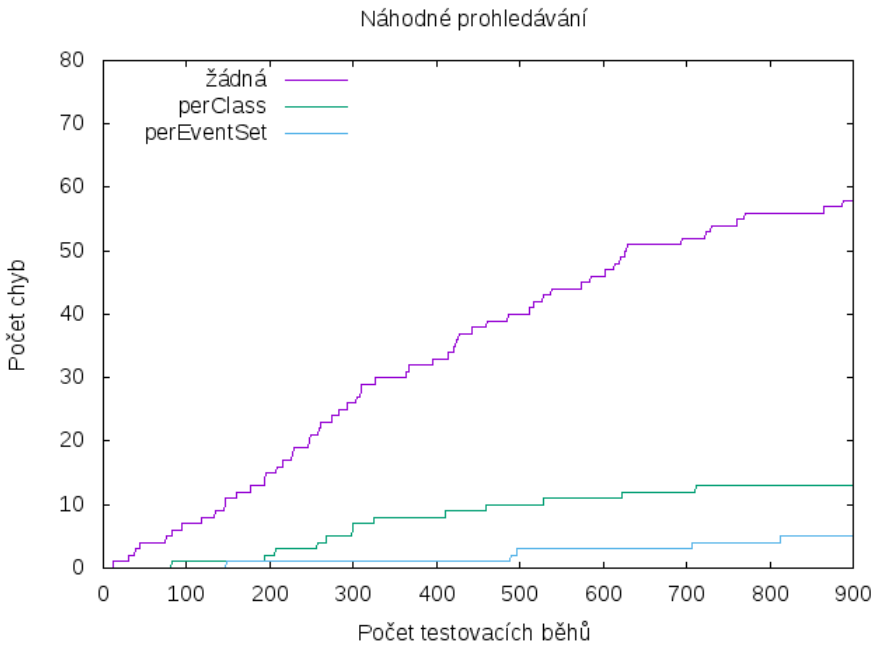
Obrázek 5.12: Metrika Eraser* na testovacím programu Rover s genetickými algoritmy



Obrázek 5.13: Počet chyb objevených při testování programu Rover s genetickými algoritmy



Obrázek 5.14: Metrika Eraser* na testovacím programu Rover s náhodným prohledáváním



Obrázek 5.15: Počet chyb objevených při testování programu Rover s náhodným prohledáváním

Testovací případ Rover byl výjimkou, u které dosáhla dobrých výsledků původní implementace využívající náhodnosti. Při náhodném prohledávání i genetických algoritmech došlo k nalezení většího počtu chyb než u kterékoliv z nově naimplementovaných heuristik. Výsledky u tohoto testovacího případu jsou velmi nejednoznačné, což opět dokazuje, že vícevláknové programy jsou plné nedeterminismu. Z nově navržených heuristik dosáhla dobrých výsledků heuristika perClass v kombinaci s náhodným prohledáváním i genetickými algoritmy. Data naleznete v tabulce 5.5 a grafech 5.12 a 5.14. Grafy počtu chyb naleznete v 5.13 a 5.15.

U Airlines, Elevatoru a Crawleru dosahovaly lepších výsledků nově navržené heuristiky, zatímco u Roveru je překonala původní implementace. Z nových heuristik dosahovala napříč všemi testovacími případy dobrých výsledků heuristika perClass. Nicméně výsledky pro jednotlivé testovací programy se lišily. Volba správné heuristiky tedy opět záleží na konkrétním testovaném programu, nelze jednu zvolit univerzálně. Ačkoliv Rover celkové statistiky zhoršuje, lze říci, že nově navržené heuristiky dosáhly dobrých výsledků a jejich rozšíření či efektivnější integrování do procesu verifikace by mohlo přinést příznivé výsledky.

Kapitola 6

Závěr

Tématem této bakalářské práce bylo testování vícevláknových aplikací v jazyce Java. Jelikož se chyby ve vícevláknových programech objevují pouze zřídka a za nestandardních okolností, existují techniky, jejichž cílem je šanci na odhalení chyb zvýšit. V případě této práce byla využita technika vkládání šumu, kdy jsou do běhu testovaného programu vkládány další instrukce, jako například uspání vlákna či přepnutí kontextu. V rámci dřívějšího výzkumu se pro instrumentaci bytecodu za účelem vkládání šumu a spuštění testů využíval nástroj ConTest od firmy IBM. Nicméně možných konfigurací pro testování s využitím vkládání šumu existuje velké množství. Proto byl na fakultě vyvinut nástroj SearchBestie, který testování řeší jako optimalizační problém. Původní autor nástroje, Ing. Zdeněk Letko, Ph.D., fakultu opustil a nástroj ConTest se již před několika lety přestal udržovat. Proto bylo potřeba nalézt alternativní řešení, za které byl zvolen nástroj RoadRunner. Na propojení SearchBestie a RoadRunneru jsem začal pracovat již při své projektové praxi a v rámci první částí zadání bakalářské práce jsem jej dokončil. Pro generování artefaktů pokrytí jsem nejdříve upravil a nyní aktivně využívám knihovnu Coverage lib. Po dokončení tohoto propojení nastal čas na srovnávací testy. Vyšlo najevo, že nástroj RoadRunner pro většinu testovaných programů výrazně překonává původně využívaný nástroj ConTest v oblasti rychlosti. Co se týče počtu nalezených chyb, výsledky nebyly jednoznačné, každý nástroj vynikal u jiných benchmarků. Nicméně závislost SearchBestie na Contestu byla úspěšně odstraněna a nyní je možné SearchBestie s RoadRunnerem používat v dalším výzkumu a případně i v praxi na skutečných projektech.

Po dokončení tohoto propojení se přešlo na návrh a implementaci nových heuristik umožňujících přesnější vkládání šumu. Pro tyto účely opět došlo k rozšíření nástroje RoadRunner, nyní za účelem získání seznamu všech instrumentovaných míst na čtyřech různých úrovních abstrakce. Do řetězce analyzátorů uvnitř nástroje se přidal nový nástroj filtrující všechny události, které šumem nemají být ovlivněny. Po dokončení implementace došlo k sérii experimentů srovnávajících nové heuristiky s původními založenými na náhodnosti. Ukázalo se, že tyto heuristiky v kombinaci s metaheuristickým průchodem stavového prostoru v podobě *genetických algoritmů* skutečně přináší ve většině případů pozitivní výsledky. Dokončení těchto experimentů také znamenalo splnění posledního bodu programu a tedy i splnění všech cílů stanovených při zadávání práce.

Nad rámec tohoto zadání se dá zařadit implementace čtyř různých přesných noise placement heuristik, ačkoliv původní návrh vyžadoval pouze jednu (odpovídající heuristice perProgLoc). Navíc také došlo k implementaci chybějících heuristik TimeoutTampering a HaltOneThread, které nově využívaný nástroj RoadRunner na rozdíl od ConTestu neposkytoval. Dále se také v rámci implementace rozšířila API RoadRunneru o dvě nové callback

metody. První z nich nástroje informuje, že testovaná aplikace havarovala kvůli neošetřené výjimce. Druhá metoda je volaná před přístupem k proměnné typu pole. V rámci tohoto rozšíření bylo třeba zasáhnout až do interních mechanismů nástroje provádějících instrumentaci bytecodu.

Jako další směr navrhuji využít metody *statické analýzy* pro určení míst k ovlivnění šumem. Kromě této automatické techniky by jistě pozitivní přínos mohla poskytnout i definice míst zkušeným programátorem po komplexní analýze kódu. Další výzkum by se též mohl zabývat novými pokročilejšími heuristikami, které by určovaly místa pro vložení šumu s využitím informací o minulých bězích programu či zaznamenaných artefaktech pokrytí. Pro zkrácení v některých případech velmi dlouhých běhů testů navrhuji využít *dynamické vkládání šumu*, které se spustí, až když program dospěje do určitého předem definovaného bodu, a může se případně i opakovaně vypínat a zapínat, aby docházelo pouze k nezbytně nutné míře ovlivnění testovaného programu. Dále doporučuji vyhodnocení nových heuristik na větší sadě programů. Zajímavé by jistě bylo i experimentovat s různým nastavením genetických algoritmů či využití AdaBoost, algoritmu pro získávání znalostí z dat.

Literatura

[1] Avros, R.; Hrubá, V.; Křena, B.; aj.: Boosted Decision Trees for Behaviour Mining of Concurrent Programs. In *Proceedings of MEMICS'14*, NOVPRESS s.r.o., 2014, ISBN 978-80-214-5022-6, s. 15–27.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10687

[2] Edelstein, O.; Farchi, E.; Goldin, E.; aj.: Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, ročník 15, 2013: s. 3–5.

[3] Fiedor, J.; Hrubá, V.; Křena, B.; aj.: Advances in noise-based testing of concurrent software. *Softw. Test. Verif. Reliab*, ročník 25, 2015: str. 272–309, doi:10.1002/stvr.1546.

[4] Flanagan, C.; Freund, S. N.: The RoadRunner dynamic analysis framework for concurrent programs. *Program Analysis for Software Tools and Engineering (PASTE)*, 2010.

[5] Goetz, B.; Peierls, T.; Bloch, J.; aj.: *Java Concurrency in Practise*. Addison-Wesley Professional, 2006, ISBN 0321349601, 403 s.

[6] Křena, B.; Letko, Z.; Ur, S.; aj.: A Platform for Search-based Testing of Concurrent Software. *PADTAD*, ročník 10, 2010.

[7] Křena, B.; Letko, Z.; Vojnar, T.: *Analysis and Testing of Concurrent Programs*. Faculty of Information Technology BUT, 2012, ISBN 978-80-214-4464-5, 136 s.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10229

[8] Křena, B.; Letko, Z.; Vojnar, T.: Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. *RV*, ročník 11, 2012.

Příloha A

Obsah přiloženého CD

V této sekci se nachází popis přiloženého CD. Obsah je rozdělen na dvě části. V první se nachází seznam souborů, se kterými přijde uživatel přímo do styku. Ve druhé naleznete seznam podpůrných souborů, které jsou využívány interně.

Hlavní soubory a adresáře

- README.md - instrukce pro práci s CD, velmi podobné manuálu, ale v anglickém jazyce,
- thesis - zdrojové kódy v latexu pro tuto práci,
- thesis.pdf - pdf verze této práce,
- repositories - zdrojové kódy pro SearchBestie, RoadRunner a Coverage lib,
- config - konfigurační soubory pro SearchBestie a ecj,
- benchmarks - testovací programy, na kterých byly prováděny experimenty, popis naleznete v 4.1,
- test_data/rr_ct_cmp - data ve formátu csv, ze kterých byly generovány tabulky v sekci 4.5,
- test_data/fine_noise - data ve formátu csv, ze kterých byly generovány tabulky a grafy v sekci 5.5,
- check.sh - skript, který ověří, zda jsou nainstalovány všechny potřebné nástroje pro běh testů,
- compile.sh - skript, který přeloží nástroje v repositories a vygeneruje archiv searchbestie.tar.gz,
- demonstrate.sh - skript, který spustí demonstrační testy na programu Airlines,
- start_fine_noise_test.sh - skript, který spustí kompletní experimenty s novými heuristikami,
- start_rr_cmp_test.sh - skript, který spustí RoadRunnerovskou část experimentů využitých při porovnávání ConTestu a RoadRunneru,

- output - adresář, do kterého se uloží výstupní soubory testů,
- clean.sh - skript smaže obsah adresáře output.

Pomocné soubory a adresáře

- io/dkozak/scripts/UpdateEC.class - program přidávající nové dimenze do konfiguračního souboru pro ecj,
- insertProgLoc.py - program přidávající nové dimenze do konfiguračního souboru pro SearchBestie,
- parseTestParam.py - program načítající parametry z konfiguračního souboru pro SearchBestie,
- runEC_simple.sh - skript, který spustí testování s využitím ecj v demonstračním testu,
- runEC_fine_noise.sh - skript, který spustí testování s využitím ecj pro testy nových heuristik,
- kill_test_procs.sh - skript, který ukončí všechny zbývající testovací procesy, pokud některé stále běží,
- init.sh - skript, který se využívá při inicializaci SearchBestie po rozbalení archivu searchbestie.tar.gz.

Příloha B

Uživatelská příručka

V této sekci naleznete návod, jakým způsobem pracovat s jednotlivými repozitáři, požadavky nutné ke spuštění nástrojů a také popis způsobu, jakým pracují skripty přiložené na CD.

B.1 Překlad a spuštění testů

Pro správné spuštění demonstračního testu je potřeba operační systém Linux, Java verze 8 a Python verze 3. Python je potřeba pouze pro spuštění skriptů *insertProgLoc.py* a *parseTestParam.py*, ostatní projekty vyžadují pouze Javu a Linux. Pro přeložení projektů jsou zapotřebí nástroje *ant* a *gradle*. Zda je váš systém nastavený správně můžete ověřit pomocí skriptu *check.sh*.

SearchBestie, RoadRunner i Coverage lib naleznete v repozitářích typu git na serveru perchta.fit.vutbr.cz. Jejich přesné adresy jsou uvedeny v seznamu níže. Pro získání přístupových práv je třeba kontaktovat administrátora tohoto serveru. Ve složce repositories se nacházejí ve chvíli odevzdávání práce aktuální klony z těchto repozitářů.

Adresy jednotlivých repozitářů

- SearchBestie - git@perchta.fit.vutbr.cz:8022/jct/searchBestie.git,
- RoadRunner - git@perchta.fit.vutbr.cz:8022/jct/roadrunner.git,
- Coverage lib - git@perchta.fit.vutbr.cz:8022/jct/coverage.git.

Po naklonování repozitářů je možné projekty přeložit. Jako první by se měla přeložit Coverage lib, a to příkazem *gradle jar*. Vzniklý archiv coverage.jar je třeba nakopírovat do složky repositories/roadrunner/jars. Nyní je možné přeložit RoadRunner příkazem *ant*. Na závěr lze přeložit SearchBestie příkazem *gradle fatjar*. Jelikož tento proces je plně automatizovatelný, tuto činnost vykoná skript *compile.sh*, který všechny tři projekty přeloží a sestaví je do archivu searchbestie.tar.gz. Tento archiv v sobě obsahuje SearchBestie, RoadRunner, Coverage lib, všechny knihovny, na kterých jsou tyto nástroje závislé, a i všechny pomocné skripty a programy. Měl by tedy být použitelný v libovolném prostředí splňujícím výše uvedené podmínky.

Pro správnou funkci demonstračního skriptu je třeba nenarušovat adresářovou strukturu obsahu CD. Nejdříve příkazem *./compile.sh* přeložte projekt. Poté příkazem *./demonstrate.sh* spusťte demonstrační testování. Skript spustí tři testy nad testovacím programem

Airlines. První test demonstruje funkčnost propojení mezi SearchBestie a RoadRunnerem. Spustí tedy test bez nových heuristik a provede 10 iterací při náhodném prohledávání stavového prostoru. Druhý test demonstruje funkčnost nové metriky perObject a opět spustí 10 iterací při náhodném prohledávání stavového prostoru. Třetí test demonstruje funkčnost propojení mezi ejc, SearchBestie a RoadRunnerem a spustí krátký test se třemi generacemi po 5-ti jedincích. Výsledky testů naleznete v adresářích *output/rnd*, *output/rnd_fine_noise* a *output/ecj_fine_noise*. Celý test generuje velké množství výstupních souborů s rozdílným formátem a smyslem. Pro rychlou analýzu doporučuji vyhledat soubory csv s postfixem _res v adresářích sbestie_output. Z nich lze relativně přehledně vyčíst výsledky testů a také se dobře zpracovávají za pomoci skriptů.

K dispozici též máte skripty, které spustí celý testovací proces využitý pro generování dat pro tuto práci. Skript *start_rr_cmp_test.sh* spustí srovnávací testování využité v sekci 4.5 pro porovnávání ConTestu a RoadRunneru. Jedná se pouze o část experimentů nad nástrojem RoadRunner, ConTest nemohl být kvůli licenci přiložen. Skript *start_fine_noise_test.sh* spustí srovnávací testy nových heuristik využité pro sekci 5.5. Nutno ovšem podotknout, že spuštění těchto testů je náročné z časového hlediska i na zdroje v PC. Doba běhu testu se může v některých případech pohybovat až v řádu dní a u testů nových metrik může dojít k současnému spuštění až 80 procesů. Dojde totiž ke spuštění 20-ti testů se SearchBestie, 20-ti testů s ecj a každý z těchto procesů pravidelně startuje RoadRunner. Proto je doporučeno tyto testy spustit pouze na výkonném serveru. Množství a velikost výstupních souborů, které tyto testy produkují, je též velká, v případě posledního spuštění testů nových heuristik dosáhla výstupní složka velikosti 3,5 GiB.

Pokud chcete projekt umístit do jiného adresáře, postupujte prosím následovně. Po umístění archivu v adresáři vašeho výběru jej rozbalte příkazem *tar -xf searchbestie.tar.gz*, přesuňte se příkazem *cd searchbestie* do rozbalené složky a inicializujte ji spuštěním příkazu *./init.sh*. Skript vytvoří složku bin se skripty *startSB.sh*, který spustí SearchBestie, a *startRR.sh*, který spustí RoadRunner. Hlavním účelem těchto skriptů je vyřešit problémy s nastavováním proměnné *classpath*, což bývá ošemetné. SearchBestie také očekává, že jí bude v proměnné prostředí *command* předána cesta ke skriptu pro spuštění RoadRunneru. Tento problém je také vyřešen ve skriptu *startSB.sh*. Nyní již můžete s oběma nástroji pracovat podle jejich manuálů. Je ovšem nutno podotknout, že všechny skripty na CD jsou připraveny pro defaultní adresářovou strukturu a nebudou proto po této změně okamžitě fungovat, bude třeba je též upravit.

B.2 Analýza výstupních souborů

V této sekci naleznete informace, jakým způsobem lze vyhodnocovat výstupní data, která SearchBestie produkuje. Pojďme se nejdříve podívat na strukturu výstupní složky jednoho experimentu.

Obsah výstupního adresáře jednoho experimentu

- sbestie_output - složka se soubory obsahující informace o celém experimentu,
- avio - složka s artefakty pokrytí pro metriku Avio,
- duPairs - složka s artefakty pokrytí pro metriku DUPairs,
- eraser - složka s artefakty pokrytí pro metriku Eraser,

- exception - složka s informacemi o neošetřených výjimkách, ke kterým při běhu aplikace došlo,
- goldiLock - složka s artefakty pokrytí pro metriku Goldilock,
- goodLock - složka s artefakty pokrytí pro metriku GoodLock,
- hbPairs - složka s artefakty pokrytí pro metriku HBpairs,
- log - složka s výstupním souborem z jednoho běhu RoadRunneru, generována automaticky, pro analýzu celého experimentu nezajímavá,
- objEnumSet.txt - identifikované objekty za všechny běhy testu,
- threadEnumSet.txt - identifikovaná vlákna za všechny běhy testu.

Každá zapnutá metrika pokrytí je reprezentována jednou složkou. V ní naleznete soubory typu *.cover* pro každý běh. Tyto soubory obsahující seznam artefaktů pokrytí za daný běh. Některé metriky také produkují soubory *.err* se seznamem chyb, které objevily. Nejdůležitějším adresářem je *sbestie_output*. Všechny ostatní soubory a adresáře jsou generovány RoadRunnerem a Coverage lib, ale *sbestie_output* je generovaný přímo SearchBestie a obsahuje souhrnné informace o experimentu. Nejdůležitější informace naleznete v již výše zmíněném souboru s názvem obsahujícím *_all.csv*. Vyjma hlavičky odpovídá každý řádek tohoto souboru jednomu testovacímu běhu. Klíčové sloupce jsou *CTexception*, tedy počet neodchycených výjimek, *testDuration*, tedy délka experimentu, a *SBInterrupt*, který nabývá hodnoty 1, pokud byl test ukončen ze SearchBestie kvůli nadměrně dlouhému běhu. Kumulované pokrytí pro danou metriku je možné získat z analýzy jejích *.cover* souborů.

Kromě znalosti struktury výstupních souborů jednoho experimentu je samozřejmě důležité i vědět, kdy tyto soubory hledat. Kořenový adresář pro výstupní data je *output*. Po spuštění skriptu *demonstrate.sh* v něm naleznete adresáře *rnd* s daty pro 10 náhodných iterací bez nových heuristik, *rnd_fine_noise* s daty pro 10 náhodných iterací s heuristikou perObject a *ecj_fine_noise* s daty pro test s genetickými algoritmy. Po spuštění skriptu *start_rr_cmp_test.sh* v adresáři *output* naleznete adresáře s názvy ve formátu *sbestie_testprogram_random* a *sbestie_testprogram_random_20*, kde *testprogram* odpovídá konkrétnímu testovacímu programu. Varianta bez postfixu 20 je 1000 iterací náhodného prohledávání, varianta s postfixem 20 je 20 fixních stavů, každý prozkoumaný 100krát. Po spuštění skriptu *start_fine_noise_test.sh* v adresáři *output* naleznete adresáře s názvy ve formátu *testprogram_heuristika* pro jednotlivé kombinace testovacích programů a heuristik.