VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

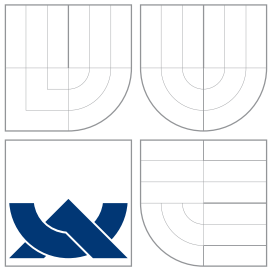
STATICKÁ ANALÝZA JAVA PROGRAMŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

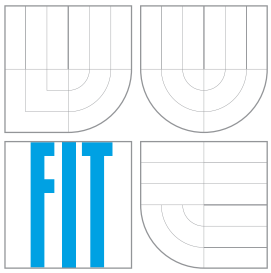
AUTOR PRÁCE
AUTHOR

PAVEL VYVIAL

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STATICKÁ ANALÝZA JAVA PROGRAMŮ

STATIC ANALYSIS OF JAVA PROGRAMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVEL VYVIAL
AUTHOR

VEDOUCÍ PRÁCE — ING. BOHUSLAV KŘENA, PH.D.
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Vyvial Pavel**
Obor: Informační technologie
Téma: **Statická analýza Java programů**
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s principy statické analýzy.
2. Prostudujte nástroje použitelné pro statickou analýzu Java programů. Zaměřte se především na nástroje použitelné na úrovni Java byte-kódu.
3. Použijte nástroj FindBugs (případně další nástroje) pro ověření vlastností Java programů pro potřeby projektu SHADOWS.
4. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího vývoje.

Literatura:

- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005. ISBN 3-540-65410-0.
- Schwartzbach, M.I.: Lecture Notes on Static Analysis, BRICS, Department of Computer Science, University of Aarhus, Denmark, 2006.
- FindBugs [online]. Poslední změna: 2007-08-11. Dostupné na URL: http://findbugs.sourceforge.net/

Při obhajobě semestrální části projektu je požadováno:

- První dva 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 2007
Datum odevzdání: 14. května 2008

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Originál licenční smlouvy je uložen v archivu FIT VUT v Brně.

Abstrakt

Projekt SHADOWS se zabývá vývojem software, který je schopen automaticky opravovat chyby v programech. Po každé opravě je třeba zjistit, zda opravná akce úspěšně odstranila chybu a nezanesla do kódu chybu novou, mnohem závažnější. Ve své bakalářské práci se zabývám právě takovýmto dokazováním korektni opravy kódu v konkurentních systémech. Některé z chyb v konkurentních systémech mohou být opraveny automatickým přidáním synchronizace. Při takovémto opravování chyb je potřeba zkontrolovat zda v uzamykané části neexistuje instrukce monitorenter, která by mohla představovat potenciální nebezpečí uváznutí. Dokazování korektnosti opravy je prováděno za pomocí Control Flow Graph analýzy nad Java byte-kódem. Prototyp k tomuto účelu využívá statickou analýzu zastoupenou nástrojem FindBugs.

Klíčová slova

automatická oprava chyb, formální analýza software, FindBugs, Java byte-kód, statická analýza, SHADOWS

Abstract

The project SHADOWS has started research which is developing software for automatic bug healing. We work with self-healing software, which looks for concurrent bugs. If the detection software finds a bug, the healing action will be performed. After every healing action, one would like to know whether this action has fixed the detected problem and, perhaps even more importantly, that it has not caused any other, possibly even more serious, problem. Therefore this paper describes a technique which gives the answer for this question after automatical healing. One can fix some concurent bugs by adding healing locks. One does healing assurance by searching monitorenter instruction and uses Control Flow Graph analysis over Java byte-code. The prototype uses static analysis (tool FindBugs) for this purpose.

Keywords

automatic bug healing, FindBugs, Java byte-code, software formal analysis, static analysis, SHADOWS

Citace

Pavel Vyvial: Statická analýza Java programů, bakalářská práce, Brno, FIT VUT v Brně, 2008

Statická analýza Java programů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Bohuslava Křeny, Ph.D. a uvedl jsem všechny prameny ze kterých jsem čerpal.

.....
Pavel Vyvial
14.5.2008

Poděkování

Rád bych poděkoval Ing. Bohuslavu Křenovi, Ph.D., za poskytnutí možnosti pracovat na projektu SHADOWS a Bc. Zdeňkovi Letkovi za jeho čas a trpěloviost, kterou mi věnoval při konzultacích. Tato práce byla podpořena Evropskou unií v rámci FP6-IST projektu SHADOWS (č. smlouvy IST-035157). Za obsah práce odpovídá pouze její autor. Tato práce nevyjadřuje názor Evropské unie a Evropská unie není odpovědná za užití jakékoliv informace v práci uvedené.

Obsah

Kapitola 1

Úvod

Jak ostatně vše v životě nemá pouze dobré či špatné stránky, tak i nástup paralelního programování spolu se zvýšením výkonu přinesl řadu těžko odhalitelných chyb. Z tohoto důvodu nám dnes sice řada aplikací využívajících multiprocesorové jádra běží rychleji, ale návrháři softwaru se tak musí denodenně vypořádávat s problémy synchronizace výpočetního výkonu. Když k tomuto faktu připočítáme, že programátoři jsou také pouze chybující lidé, otevírá se nám tímto další potencionální zdroj chyb, které nejsou ve vytvářeném software vítány.

Jednou z vlastností programátorských chyb je to, že se jejich velká část opakuje. Této skutečnosti se snaží využít projekt *SHADOWS* (A Self-healing Approach to Designing Complex Software Systems), jehož úkolem je vytvoření software, který bude schopen některé z těchto chyb najít a automaticky opravit. Projekt patří do programu EU – IST – Information Society Technologies a jedná se o FP6 – Sixth Framework Programme. [12]

SHADOWS se zabývá odhalováním a opravováním chyb, které postihují tyto tři oblasti: funkčnost, výkonnost a paralelizmus. V této práci se budu zabývat chybami spadajícími pod paralelizmus, jelikož je to oblast, které se věnuje vědecká skupina na FIT VUT v Brně.

Chyby vyplývající z paralelizmu bývají často zapříčiněny špatnou synchronizací. Jednou z takovýchto chyb je např. níže uvedený data race dvou vláken reprezentovaných obrázkem 1.1. Jako *data race* je nazýván stav, kdy v jednom okamžiku nejméně dvě vlákna přistupují k téže proměnné a alespoň jeden z těchto přístupů je zápis. Příklad 1.1 nám představuje 1. vlákno, které chce inkrementovat sdílenou proměnnou x a 2. vlákno, které chce tutéž proměnnou x zvýšit o číslo 2. Před spuštěním obou vláken je proměnná $x = 0$. V případě správné synchronizace 1.2 bude mít proměnná x vždy po provedení obou vláken číslo $0+1+2 = 3$. Jak ovšem můžete vidět na 1.3, špatně synchronizovaný kód při nešťastně přepnutém kontextu může zapříčinit i výsledek $0 + 1 + 2 = 1$, který jistě není v programu žádoucí.

1. vlákno:	**2. vlákno:**
load x	load x
inc	add 2
store x	store x

Obrázek 1.1: Abstraktní byte-kódová reprezentace dvou vláken [12]

1. vlákno:	2. vlákno:	z1	z2	x
load x		0		0
inc		1		0
store x		1		1
	load x		1	1
	add 2		3	1
	store x		3	3

Obrázek 1.2: Při správné synchronizaci je výsledek v proměnné $x = 3$ [12]

1. vlákno:	2. vlákno:	z1	z2	x
load x		0		0
inc		1		0
	load x	1	0	0
	add 2	1	2	1
	store x	1	2	2
store x		1		1

Obrázek 1.3: Chybějící sychronizace může zapříčinit špatný výsledek v proměnné $x = 1$ [12]

Jak výše zmíněný příklad ukazuje, může mít chybějící synchronizace neočekávané výsledky. Z důvodu, že většina programátorů tvoří programy, které by rádi měli bez takovýchto překvapení, je to dobrá motivace pro vytvoření samoopravovacího software. Takovýto software by pak totiž patřičnou synchronizaci, kterou mohl programátor zapomenout napsat do programu, doplnil a tím zaručil bezpečný a správný chod programu.

Každé naše automatické opravování chyb v programu se skládá z těchto čtyř fází: [13]

1. **Detekce problému** – Předtím, než začneme cokoliv léčit, je potřeba zjistit, že něco v analyzovaném systému nefunguje správně.
2. **Lokalizace problému** – Jakmile zjistíme nekorektní chování analyzovaného programu, je třeba najít pravou podstatu problému.
3. **Léčení problému** – Pokud uspějeme v hledání zdroje špatného chování programu, podíváme se do seznamu možných léčících akcí a jednu z nich vybereme.
4. **Ověření korektnosti léčení** – Jakékoliv léčení představuje zásah do systému, který nemusí námi léčený problém vyřešit a může se dokonce stát zdrojem chyby nové. Právě proto je potřeba ověřit korektnost provedeného léčení a dokázat tak, zda léčící akce k odstranění problému pomohla či nikoliv. Řešení tohoto problému je předmětem této bakalářské práce.

Detekce problému je prováděna monitorováním běžícího Java programu pomocí instrumentovaného Java byte-kódu. Hledání chyb při testování konkuretních programů je velice obtížné z důvodu, že existuje obrovské množství způsobů, jak se můžou jednotlivá vlákna při běhu aplikace prokládat. Což také předurčuje nesystematické prohledávání těchto průchodů k malé pravděpodobnosti nalezení všech chyb, které může konkurentní program skrývat. Z tohoto důvodu chyby pouze nehledáme, ale snažíme se je i vyvolávat pomocí tzv. vkládání „šumu". Pro instrumentaci a vkládání „šumu" je využíván nástroj ConTest od IBM se kterým VUT–FIT v této oblasti výzkumu spolupracuje a můžete se o něm více dočíst v [7, 13].

Lokalizace problému je u nás prováděna pomocí data race detectoru nebo deadlock detectoru, které slouží jako příklad tzv. oráklu, který se k tomuto účelu využívá. Druhým využívaným přístupem, který se používá pro získání podstaty chyby, je velký počet testování s rozdílnou instrumentací a statistickým vyhodnocením. Oba tyto přístupy mohou být kvůli redukci hlášení o neexistujících chybách kombinovány s metodami formální verifikace. Přitom je třeba mít na paměti, že použití formálních metod má i svou stinnou stránku v podobě stavové exploze. [13]

Léčení problému může pracovat na základě výběru léčící akce, která provede zásah do kódu běžící aplikace. Druhou možností pak je pouhé upozornění programátora na to, že při psaní programu se nejspíš dopustil chyby. Naším cílem je vyvinutí software, který bude schopen provádět ony léčící úpravy kódu. K tomuto způsobu léčení máme připraveny léčící akce v podobě ovlivnění časovače nebo synchronizačních modifikací. Přitom druhý způsob léčení může mít např. podobu vložení zámků, ovlivňujících sdílenou proměnnou (nad kterou byl zjištěn data race) ve všech vláknech, které k proměnné mají přístup. Výše uvedený příklad by tedy mohl být vyléčen způsobem zobrazeným na 1.4. [13]

1. vlákno:	**2. vlákno:**
monitorenter	monitorenter
load x	load x
inc	add 2
store x	store x
aload	aload
monitorexit	monitorexit

Obrázek 1.4: Abstraktní byte-kódová reprezentace dvou zámků správně opravených vláken

Ověření korektnosti léčení (OKL) je potřeba proto, že jakákoliv léčící akce může sice chybu odstranit, avšak může v opravovaném programu novou chybu i vytvořit. Z tohoto důvodu je nutno po léčení zkontrolovat, zda byla léčená chyba odstraněna a nebyla zanesena chyba nová. K tomu využíváme *formální verifikaci* (model checking a *statickou analýzu*). V této práci se zaměřuji právě na oblast ověřování koretnosti léčení, provedeného prostřednictvím vložení nové synchronizace, pomocí *statické analýzy.* [13]

Při léčení pomocí zámků je potřeba ověřit, zda se v sekcích, které jsme vložením zámků vytvořili, nenachází jiný zámek, který by mohl vést k *uváznutí programu (deadlock).*

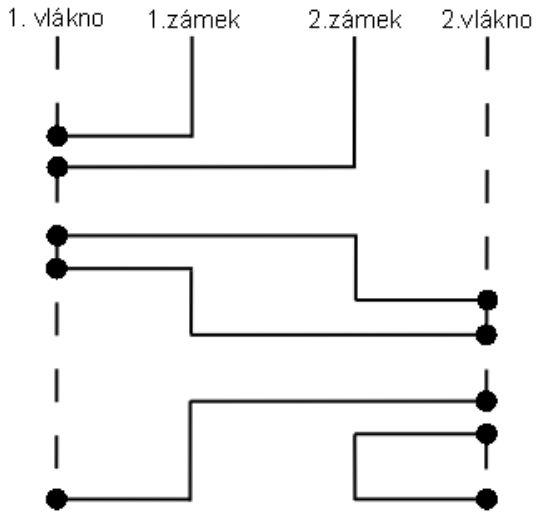
Uváznutí (deadlock) je v programech nechtěný stav zapříčiněný špatnou synchronizací programu. Při uváznutí zpravidla všechny procesy čekají na uvolnění zdroje, který vlastní jiný čekající proces. [18]

Deadlock nastává pouze v případě splnění následujících čtyř podmínek: [20]

1. Prostředek může v jednom okamžiku používat jenom jeden proces.
2. Proces musí vlastnit jeden zdroj a čekat na druhý.
3. Prostředky vrací pouze proces po dokončení jejich využití.
4. Cyklická závislost na sebe čekajících procesů.

Obrázek 1.5 znázorňuje problém uváznutí. Je na něm ukázano, jak si nejdřív 1. vlákno vyžádá oba zámky. Následně jsou oba zámky 1. vláknem uvolněny a získá je 2. vlákno, které si je chvilku ponechá a také je uvolní. Kdyby program fungoval tímto způsobem, bylo by vše v pořádku.

Jak je ovšem na témže obrázku ukázano, tak po uvolnění obou zámků 2. vláknem požádají obě vlákna o přidělení obou zámků. Přitom se ovšem každému vláknu podaří získat pouze po jednom zámku, z důvodu, že se jedná o zámky, které může najednou vlastnit pouze jedno vlákno. Vlákna mohou žádat o více jak jeden zámek najednou. Zámků se vlákna vzdávají pouze poté, co najednou získají oba zámky (vzniká zde kruhová závislost). První vlakno čeká na 2. zámek, který vlastní druhé vlákno, jenž čeká na 1. zámek vlastněný prvním vláknem. Tímto máme splněny všechny nutné podmínky pro vytvoření deadlocku a program uvázne.



Obrázek 1.5: Deadlock v případě použití dvou zámků dvěma vlákny

Abychom během léčení přidáním dalšího zámku nemohli podobný problém způsobit, ověřujeme pomocí statické analýzy zda oblast uzamykaná léčícími zámky neobsahuje jiný zámek v podobě instrukce monitorenter.

Statická analýza využitá při ověřování korektnosti programu v této bakalářské práci mě nutí napsat pár řádků o *formální verifikaci*, do které tento druh analýzy spadá. Přičemž o samotné statické analýze toho bude napsáno více v 2. kapitole.

Formální verifikace je jednou z možností využívaných k ověření korektnosti systému. Vedle inspekce systému, simulace a testování, má formální verifikace výhodu v tom, že nemusí být využívána pouze k hledání chyb. S pomocí formální verifikace je totiž dokonce možné matematicky dokázat, že v daném systému na základě určitých podmínek se již další chyby nevyskytují. Před začátkem takovéhoto dokazování je nutné si určit, jak je chápán termín "korektní systém". [18]

Korektní systém je v tomto článku program, kde nemůže dojít k uváznutí (deadlock).

Mezi tři základní metody formální verifikaci patří: [18]

- dokazování vět (theorem proving)
- model checking
- statická analýza (static analysis)

Dokazování vět je přístup podobný matematickému dokazování. Jednou z jeho nevýhod je, že nástroje zabývající se dokazováním teorému zcela neautomatizují dokazování korektnosti systému. Z tohoto důvodu je při využití dokazování teorému zapotřebí expert, který má důkaz vést. [17]

Model checking je založen na systematickém, explicitním či symbolickém průchodu všemi stavy systému nebo jeho pokud možno automaticky vytvořené abstrakce. Tento způsob ověření korektnosti systému sebou ovšem nese dvě nevýhody (časovou a paměťovou náročnost). [17]

Statická analýza se snaží obejít stavovou explozi získáním co nejvíce informací ze zdrojového kódu, aniž by byl program vůbec spuštěn. Nevýhodou tohoto ověřování korektnosti systému je hlášení chyb, které se v programu nemusí vůbec vyskytovat. Jednou z oblastí, kterým se statická analýza věnuje, je např. hledání chybových vzorů v programu, který je využit v této práci. [17]

Z výše uvedeného úvodu nám vyplývá, že tato bakalářská práce je zaměřena na provádění automatického ověřování korektnosti léčení, které bylo provedeno za pomocí přidání zámků do zdrojového kódu. Ověřování korektnosti léčení má chránit před uváznutím, které mohou automaticky přidané léčící zámky potencionálně způsobit. K tomuto ověřovaní je přitom využíváno statické analýzy, která spadá do formální verifikace.

V práci naleznete kapitolu věnující se statické analýze. Tato kapitola obsahuje charakteristiku statické analýzy, výčet nástrojů, které statickou analýzu využívají a podkapitolu věnující se ověřování korektnosti léčení pomocí statické analýzy. Další kapitola se věnuje nástroji FindBugs. Součástí této kapitoly je charakteristika nástroje FindBugs, Java class formátu a Control Flow Graph analýzy. Čtvrtá kapitola podrobně popisuje princip ověřování korektnosti léčení, který byl implementován. Patří sem definice vstupu a výstupu programu, popis tvorby grafu volání a algoritmus hledání zámků v jednotlivých atomických sekcích. Pátá kapitola obsahuje test, který ověřuje správnou funkcionalitu programu pro ověřování korektnosti léčení. Poslední kapitola je věnována závěru. Patří sem zhodnocení práce a nástin práce budoucí. V textové příloze lze najít návod ke spuštění analýzy.

Kapitola 2

Statická analýza

Statická analýza je činnost sloužící k ověřování software patřící do oblasti formální verifikace. Je založena na zjišťování informací o ověřovaném systému bez jeho spouštění (analýza, která získáva informace ze spuštěné aplikace, bývá nazývána *dynamickou analýzou*). Ve většině případů statická analýza pracuje nad zdrojovým kódem nebo nějakou abstrakcí ověřované aplikace. Z toho taky plyne její široké zaměření od jednoduchého syntaktického ověřování až po iterativní fixpoint výpočty nad abstrakcí zkoumaného systému. Bývá často využívána při tvorbě nástrojů pro automatickou analýzu zdrojových kódů.

Mezi základní typy statické analýzy jsou obvykle řazeny: [11]

- typová analýza (type analysis)
- hledání chybových vzorů (bug pattern searching)
- dataflow analýza (dataflow analysis) – pod kterou si můžeme představit např. zjišťování dosažitelnosti definic nebo živosti proměnných, atd.
- abstraktní interpretace (abstract interpretation)

Ve skutečnosti nejsou přesně určeny hranice, co do statické analýzy patří a co už nikoliv. V případě, že statickou analýzu budeme pokládat jako opak k analýze dynamické a vyzdvihneme tedy její vlastnost, že přímo nespouští verifikovaný kód, dal by se za speciální druh považovat i model checking. Statická analýza přitom nebývá využívana výhradně ke kontrole korektnosti systému, ale používá se také v optimalizaci, při vytváření kódu, atd... [17]

Výhody statické analýzy: [17] [11]

- umí zacházet s velmi velkými systémy
- nepotřebuje model prostředí (vstupy/výstupy, knihovny, jiné moduly) ⇒ možnost ověřování malé části velkého systému
- pomocí statické analýzy lze zjišťovat dosažitelnost definic nebo živost proměnných, atd.
- vysoký stupeň automatizace (pokud není zapotřebí vytvoření vlastní analýzy)

Nevýhody statické analýzy: [17]

- může produkovat velké množství hlášení o chybách, které neexistují, tzv. *„false alarms“*
  - v případě odstraňování tohoto problému a vytváření preciznějších nástrojů se začne narážet na podobné problémy jako při model checkingu
- analýzy jsou často vhodné pro řešení specifických úloh

Na současném softwarovém trhu je k nalezení velké množství nástrojů, které statickou analýzu využívají. Z důvodu, že se v této práci zabývám statickou analýzou Java programů, rozhodl jsem se v tabulce 2.1 uvést přehled několika nástrojů, které práci s Javou podporují.

Praktické zastoupení statické analýzy pak můžeme najít např. v těchto příkladech: [17]

- Microsoft – Windows Vista,
- Linux kernel (Sparse),
- Airbus flight control (Astrée, AbsInt),
- a systémy mnoha jiných zákazníků využívajících developerské nástroje statické analýzy

CheckStyle [2]
– Open source – Integruje se všemi zmiňovanými IDE – Custom bug patterny v Javě – Zaměřen na problémy stylu – Java and EJB 2.x – Detekce duplicit v kódu – Okolo 100 patternů – Nemá quick-fixy – Pouze pro Java kód
FindBugs [15]
– Open source – Integruje do Eclipse a NetBeans – Propracovaná správa reportů a historie chyb – Custom bug patterny v Javě – Analyzuje byte-code – Přes 300 hledaných chybových vzorů – Nemá quick-fixy – Pouze pro Java kód

IntelliJ IDEA [8]
– Analýza integrovaná do IDE – Přes 700 hledaných chybových vzorů – Custom bug patterny – Profile management – Suppresion pomocí anotací – Analýza závislostí, Dependency Structure Matrix (DSM) – Detekce duplicit v kódu
PMD [6]
– Open source – Integruje se všemi zmiňovanými IDE – Custom bug patterny v Javě a Xpath – Suppression pomocí anotací – Okolo 200 patternů v Javě, JSP, JSF – Detekce duplicit v kódu – Nemá quick-fixy
TPTP [4]
– Open source – Integrován do Eclipse – Více než 100 patternů (včtně JDT) – Custom bug patterny – Pouze pro Java kód

Tabulka 2.1: Přehled nástrojů využívajících statické analýzy k práci s Java programy

Hlavními důvody, proč byla statická analýza vybrána jako prostředek k ověřování korektnosti léčení, byly tyto: [16]

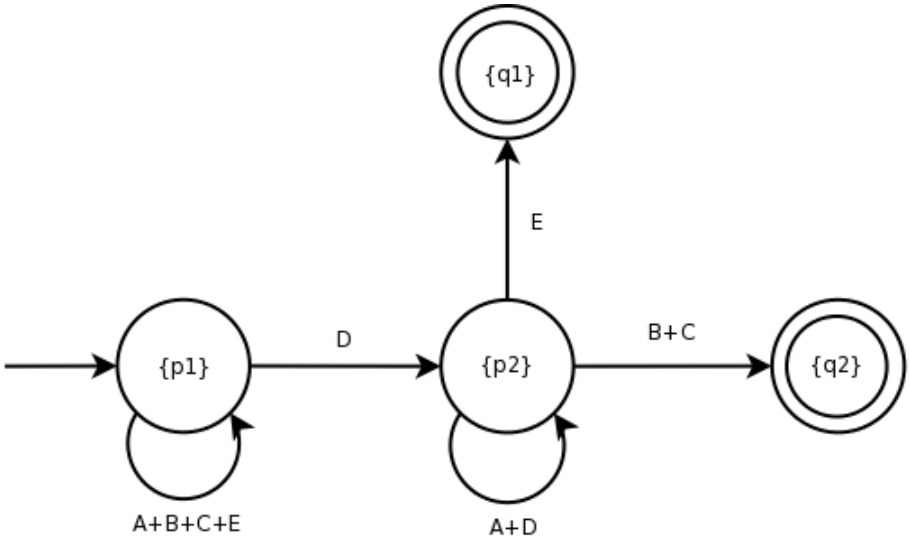
- Statickou analýzu je možné použít k ověření, že analyzovaná část kódu je korektní.
- Rychlost analýzy.
- Dovoluje vytvoření plně automatického detektoru.
- Může být prováděna nad malou částí obrovského systému.
- Při hledání chyb může dobře doplňovat testování.
- Na rozdíl od dynamické analýzy není potřeba spouštět analyzovaný kód a proto může být analýza použita už při vývoji software. Jestliže ale uvažujeme použití pro ověření léčení, máme spustitelný kód stejně k dispozici.

2.1 Ověřování korektnosti léčení pomocí statické analýzy

Jak bylo v úvodu zmíněno, hlavním úkolem práce je vytvoření detektoru, který bude schopen vyloučit případné uváznutí, který může po léčení pomocí zámků v programu potenciálně vzniknout. Budeme vycházet z faktu, že systém může uváznout jen v případě splnění všech podmínek uváznutí, které byly výše uvedeny. Stačí tedy zajistit, aby alespoň jedna z těchto podmínek nebyla nikdy splněna a k uváznutí nemůže dojít. Jeden z možných přístupů, jak se dá tedy nechtěnému deadlocku předcházet, je zakaz uzamykání místa, které už nějaký jiný zámek obsahuje. Tímto způsobem totiž nebude moci nikdy dojít ke splnění druhé nutné podmínky: „Proces musí vlastnit jeden zdroj a čekat na druhý.“ V této práci se řeší pouze detekce zamykání za pomocí monitorenter instrukcí. Ostatní druhy synchronizace jsou předmětem budoucí práce na projektu SHADOWS.

K detekci takovéto sychronizace je využívána *na vzorech založená statická analýza* (pattern based static analysis). Jako *vzor* je přitom chápána sekvence po sobě jdoucích příkazů. Obvykle pak bývají vzory popisovány pomocí regulárních výrazů, konečných automatů, gramatik, systémů různých omezení, atd. Na vzorech založená analýza pak v praxi funguje na principu detekce výskytu vzoru, nebo porušení podmínek, které hledaný vzor definují. [11]

Vzor vyhledávaný v této práci by se přitom dal vyjádřit např. tímto deteminirstickým konečným automatem:



Obrázek 2.1: Vzor vyjádřený deterministickým konečným automatem

Deterministický konečný automat z obrázku 2.1 ukazuje, že analýza kódu bude probíhat následujícím způsobem. Začne se prohledávat kód(stav $p1$). Dokud není nenalezen začátek sekce (D), ve které mají být zjištěny nechtěné synchronizační instrukce, jsou všechny ostatní instrukce ignorovány. Po detekci místa, kde by v budoucnu měl být umístěn začátek léčícího zámku, přechází automat do stavu $p2$, ve kterém začíná hledat jinou synchronizaci. V případě nalezení synchronizační nebo invoke instrukce, která odkazuje na metodu, jež obsahuje nějaký zámek, přechází automat do stavu $q2$. Tento stav přitom značí zrušení léčení

v zadaných místech. Na druhou stranu, pokud je ve stavu $p2$ nalezena instrukce, která značí ukončení sekce, ve které se mají nalézt všechny něchtěné zámky, přechází automat do stavu $q1$. Tento stav pak značí potvrzení léčení v zadaných místech. Všechny ostatní instrukce jsou konečným automatem v tomto stavu ($p2$) ignorovány.

Deterministický konečný automat 2.1 se dá také matematicky popsat jako pětice $A = (Q, T, \sigma, s, F)$ kde: [14]

$$\begin{aligned} &Q = \{q1, q2, p1, p2\} \\ &T = \{A, B, C, D, E\} \\ &\sigma(p1, A) = \sigma(p1, B) = \sigma(p1, C) = \sigma(p1, E) = \{p1\}, \sigma(p1, D) = \{p2\}, \\ &\sigma(p2, A) = \sigma(p2, D) = \{p2\}, \sigma(p2, B) = \sigma(p2, C) = \{q2\}, \sigma(p2, E) = \{q1\} \\ &s = p1 \\ &F = \{q1, q2\} \end{aligned} \tag{2.1}$$

Jednotlivé prvky, které se v tomto zápisu 2.1 vyskytují mají následující význam:

- Q – abeceda stavů ve kterých se může daný nedeterministický automat nacházet
  - q1 – stav označující úspěšné léčení ⇒ zámek nenalezen
  - q2 – stav označující neúspěšné léčení ⇒ zámek nalezen
  - p1 – stav hledání začátku místa, které se má prohledat
  - p2 – stav hledání synchronizace
- T – abeceda vstupních symbolů
  - A – všechny instrukce kromě synchronizačních instrukcí, invoke instrukcí, které odkazují na metodu se synchronizací a instrucí značící začátek a konec místa, které se má prohledávat
  - B – invoke instrukce odkazující na metodu se synchronizací
  - C – synchronizační instrukce
  - D – instrukce značící začátek místa, které se má prohledávat
  - E – instrukce značící konec místa, které se má prohledávat
- σ – přechodová funkce
- s – počáteční stav
- F – množina koncových stavů

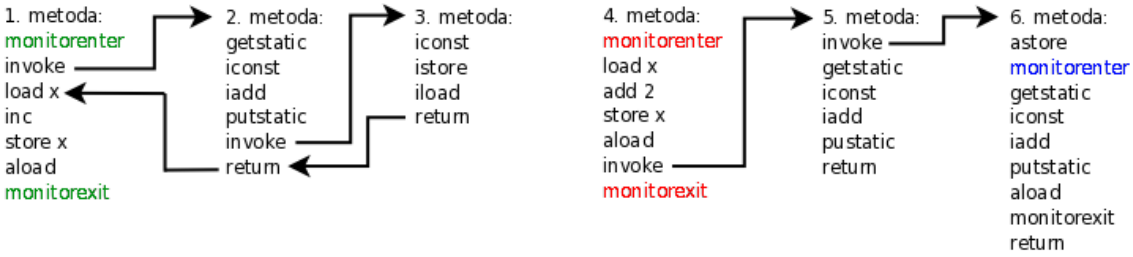
Jak může vypadat příklad s povoleným a zakázaným zamykáním, je vidět na obrázku 2.2. První vlákno na tomto obrázku má léčení znázorněné zelenými instrukcemi monitorenter a monitorexit povoleno. Je to z důvodu, že se mezi vkládanými zelenými instrukcemi nenachází žádná jiná synchronizační instrukce, která by v budoucnu mohla vyvolat deadlock. Léčení druhého vlákna je zakázáno. Je tak učiněno z důvodu, že se mezi léčícími instrukcemi monitorenter a monitorexit nachází blokující instrukce monitorenter, která by v případě léčení mohla zavinit deadlock. Tento příklad je přitom zobrazen s abstraktním byte-kódem, který zanedbává některé pro tento příklad nepodstatné instrukce. Konkrétní

1. vlákno:	2. vlákno:
	monitorenter
monitorenter	load x
load x	add 2
inc	monitorenter
store x	store x
aload	monitorexit
monitorexit	aload
	monitorexit

Obrázek 2.2: 1. vlákno lze bezpečně opravit. U 2. vlákna hrozí riziko uváznutí.

řešení ověřování korektnosti léčení za pomocí nástroje FindBugs přitom bude probráno v kapitole 4.

Ve skutečnosti situace nemusí být vždy tak jednoduchá, jak zobrazuje obázek 2.2. Léčící zámek může postihovat větší oblast, než jen pár byte-kódových instrukcí. V uzymakaném prostředí můžeme také nalézt „invoke instrukce“ reprezentující volání jiných metod. Tyto metody mohou patřit do jiných než zdrojových tříd a musí být také testovány na přítomnost synchronizačních a „invoke“ instrukcí, atd. Pro lepší představivost je přiložen obrázek 2.3, který skutečnost invoke instrukcí v uzamknutém kódu zobrazuje.



Obrázek 2.3: Povolené a zakázané léčení

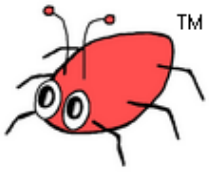
Na obrázku 2.3 můžete vidět pokus o léčení ve dvou na sobě nezávislých metodách (1. a 4. metodě).

Léčení 1. metody je z důvodu, že se mezi zelenými léčícími instrukcemi monitorenter a monitorexit nenechází žádný jiný zámek, povoleno. Při kontrole 1. metody přitom musely být na umístění zámku zkontrolovány i 2. a 3. metoda. Tato kontrola je z důvodu, že se při vykonávání kódu, který chceme v 1. metodě zamknout můžeme do těchto dvou metod dostat.

Léčení 4. metody je z důvodu, že se mezi červenými léčícími instrukcemi monitorenter a monitorexit nachází synchronizační instrukce monitorenter zakázáno. Tento zámek byl přitom nalezen až v 6. metodě do které se můžeme z uzamykaného kódu 4. metody přes 5. metodu dostat.

Kapitola 3

FindBugs



Obrázek 3.1: FindBugs logo

V předchozí kapitole bylo napsáno pár slov o statické analýze a teoretickém řešení ověřování korektnosti léčení. Nyní bych rád uvedl pár informací o nástroji FindBugs, který byl využit při praktickém řešení výše zmiňovaného vyhledávání zámků.

FindBugs je program, který za pomocí statické analýzy hledá chyby v Java kódu. Jedná se o Open source software distribuovaný pod „Lesser GNU Public License“. Jméno $FindBugs^{TM}$ a logo 3.1 je zákonem chráněným vlastnictvím od „The University of Maryland“. FindBugs je sponzorovaný Fortify Software.

Mezi další vlastnosti FindBugs patří, že pracuje nad *byte-kódem* (zkompilovanými *.class soubory). Můžeme ho integrovat do Eclipse a NetBeans. Nástroj obsahuje dobrou správu reportů a historii chyb. Při hledání chyb využívá tzv. *hledání chybových vzorů*, přičemž nám poskytuje přes 300 předdefinovaných chybových vzorů a také možnost si svůj vzor a potažmo detektor vytvořit. Možnost vytvoření *vlastního detektoru* ve formě *pluginu* byla ostatně využita i při tvorbě programu pro ověření korektnosti léčení. Jednou z nevýhod tohoto nástroje je, že se občas na výstupu objeví chybové hlášení, které upozorňuje na chybu, jež neexistuje. Je napsán v Javě a ke svému provozu potřebuje *virtuální stroj* kompatibilní se Sun's JDK 1.4. FindBugs využívá k analýze byte-kódu *BCEL* (Byte Code Engineering Library) [5]. Od verze 1.1 FindBugs podporuje detektory napsané v *ASM*[1] byte-kód frameworku. K práci s XML využívá *dom4j* [3]. Informace byly získány z [15].

FindBugs dále umí produkovat xml výstup v podobě [11]:

- Filtrování chyb – hlášení pouze závažných chyb
- Historie chyb – stopování vývoje chyb mezi vícenásobnými kontrolami
- Dolování chyb (bug mining) – využití techniky dolovaní dat při reportech

Za zmínku pak určitě stojí i uživatelské rozhraní, které je schopno zvýrazňovat chyby v kódu. Další vlastností tohoto rozhraní je možnost vkládání informací z analýzy nebo od programátora. Tohoto pak může být dobře využito při testování software v týmových projektech.

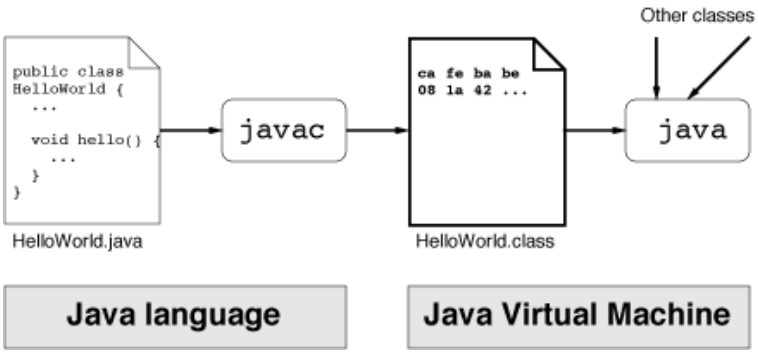
Tento nástroj byl úspěšně využit v řadě aplikací např.: Sun JDK 1.7.0–b12, eclipse–SDK–3.3M7–solaris–gtk, netbeans–6_0–m8, glassfish–v2–b43, jboss–4.0.5. V každé z těchto aplikací tak byly nalezeny chyby, které mohly být následně opraveny. [15]

FindBugs je schopen hledat tyto typy chyb [11] [15]:

- Špatný postup (bad practice) – porovnávaní stringů, hashCode() a equals(), obecné zásady jmen.
- Korektnost (correctness) – jedná se o části kódu, které programátor původně asi nezamýšlel – nekonečné smyčky, rekurzivně nekonečné smyčky, neinicializované ukazatele, matoucí jména metod.
- Záludná poruchovost kódu (malicious code vulnerability)
- Multivláknová korektnost (multithreaded correctness) – sdílení statického modelu, zamykání a odemykání ve všech cestách, nepodmíněné čekání.
- Výkonnostní – new String(String), nepoužité pole (field), použití URL ve množině nebo mapě (zpomalení hashCode a equals).
- Prohnané (dodge) – redundantní porovnávání (null), zápis do statických polí (field) z instance metody.

3.1 Formát třídového souboru (Java class format)

Výše bylo zmíněno, že FindBugs pracuje nad Java byte-code. Abych ucelil představu o tom, co se skrývá pod slovním spojením „Java byte-code“, dovoluji si uvést následující podkapitolu převzanou ze stránek BCEL [5] .



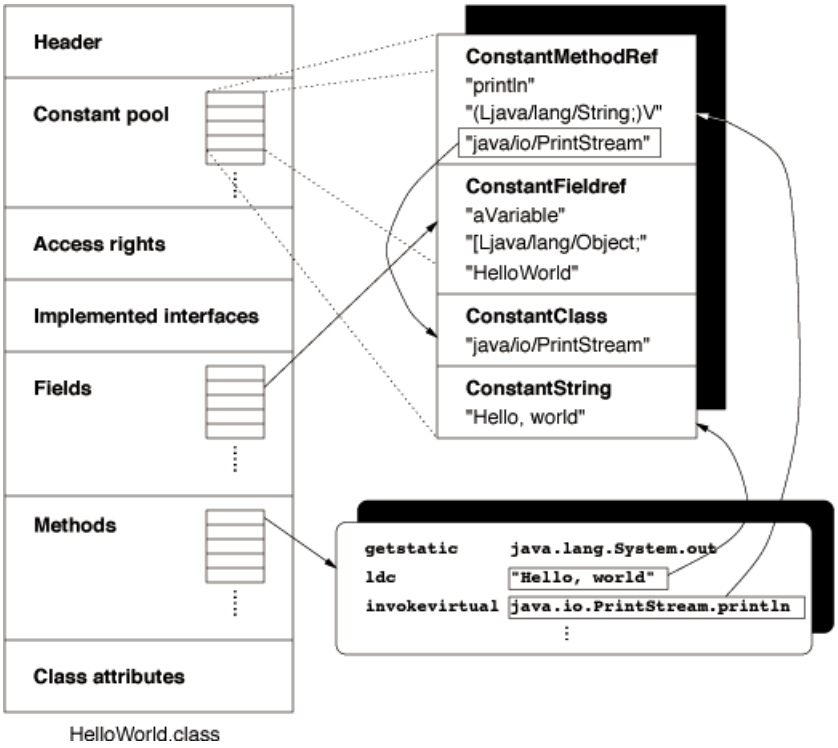
Obrázek 3.2: Kompilace a spuštění Java tříd

Obrázek 3.2 nám ukazuje proceduru kompilace a spuštění Java třídy: Zdrojový soubor (HelloWold.java) je překompilován do souboru Java třídy (HelloWold.class), načten jako

byte-kód interpreterem a spuštěn. V případě, že např. vývojáři chtějí přidat nějaké specifické vlastnosti třídě, musí transformovat třídový soubor (nakreslený silnými linkami) předtím, než je spuštěn. Toho využívá i instrumentace ConTestu [7], se kterou pracuje nástroj pro ověření korektnosti léčení.

Obrázek 3.3 ukazuje jednoduchý příklad obsahu souboru Java třídy. Začíná s hlavičkou obsahující „magickou konstantu“ a číslo verze, následuje *constant pool*, který může být hrubě přirovnán k proveditelnému textovému segmetu. Dále zde máme přístupová práva tříd zakódovaných do bitové masky, seznam rozhraní implementovaných třídou, seznam obsahující pole a metody třídy a nakonec atributy třídy, např. atribut *SourceFile* říká jméno zdrojového souboru. Atributy jsou přitom cestou pro vložení uživatelsky definovaných informací do struktury class souboru.

Proto, že všechny informace potřebují být dynamické, přijalo se usnesení, že symbolické reference tříd, polí a metod jsou za běhu zakódovány řetězcovými konstantami. Constant pool obykle zaujímá asi 60% class souboru. Tato skutečnost pak constant pool tvoří lehce zaměřitelný pro manipulace s kódem. Byte–kódové instrukce tvoří pouhých 12% z celkové velikosti class souboru.



Obrázek 3.3: Java class file formát

3.2 FindBugs CFG analýza

Mírně upravená verze níže uvedené analýzy založené na CFG (Control Flow Graph) byla využita při implementaci této práce, takže by bylo vhodné o ní napsat pár řádků. Nejdřív zavedu teminologii, kterou následně použiji v obecném tvaru algoritmu tohoto druhu analýzy.

Terminologie: [10]

- ClassContext – pokrývá všechna známá fakta o třídě (zjištěná na začátku pomocí BCEL).
- Detector – algoritmem popsaný hledaný vzor.
- Analysis objects – konečný produkt nějaké analýzy, která byla v minulosti ukončena
- Dataflow fact – stav nějaké dataflow analýzy v nějaké lokaci

Detektory založené na CFG používají CFG reprezentaci Java metod k vykonávání poněkud více sofistikovaných analýz než detektory založené na „inspekci“. FindBugs implementuje mnoho druhů dataflow analýz pro použití na CFG založených detektorů.

Níže uvedený pseudo-kód je od metody *visitClassContext()* patřící do detektorů založených na CFG. Základní idea je navštívit každou metodu analyzované třídy v kole dotazování nějaké informace z analysis objects.

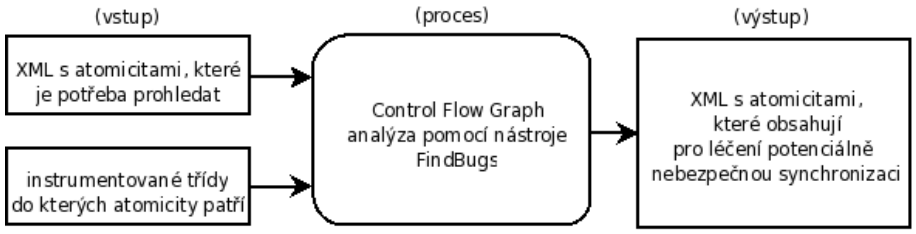
```
for each method in the class do
    request a CFG for the method from the ClassContext
    request one or more analysis objects on the method from the ClassContext
    for each location in the method do
        get the the dataflow facts at the location
        inspect the dataflow facts
        if a dataflow fact indicates an error then
            report a warning
        end if
    end for
end for
```

Obrázek 3.4: Algoritmus CFG analýzy.[10]

Kapitola 4

Ověřování korektnosti léčení pomocí FindBugs

V této chvíli by čtenář měl mít ucelený pohled na problém, který se zde bude řešit na základě návrhu výše zmiňovaného deterministeckého konečného automatu 2.1 ve spojení s nástrojem FindBugs. Na obrázku 4.1 je uveden popis fungování ověřování korektnosti léčení jako celku za kterým následují podrobněji rozebrány podstatné části programu.



Obrázek 4.1: Základní idea programu

Popis ověřování korektnosti léčení: Program začne s *vytvářením grafu volání (CallGraph)* pro jednotlivé instrumentované třídy, které byly poskytnuty k analýze prostřednictvím argumentu při spuštění FindBugs. Po vytvoření a uložení všech těchto grafů jsou ze vstupního souboru *načteny atomicity* (části kódu, které se mají prohledat – určující začátek a konec léčící synchronizace). Následně je pro každou načtenou *atomicitu spuštěna analýza*, která má mezi jejím začátkem a všemi konci nalézt nežádoucí synchronizační instrukce. V případě, že je v některé atomické sekci synchronizační instrukce nalezena, je tato atomicita poznamenána do kolekce, která je nakonec zapsána do výstupního souboru.

Vstup a výstup programu pro ověřování korektnosti léčení se dělí na tyto části:

- **Vstupní XML soubor** – definuje atomicity (místa), které se mají prohledat.
- **Vstupní instrumentované třídy** – jsou Java třídy (soubory s příponou .class) do kterých patří nadefinované atomicity, které se budou prohledávat.
- **Výstupní XML soubor** – ze vstupu definuje atomicity (místa) ve kterých byly nalezeny synchronizační instrukce

Vstupní XML soubor je dán názvem *CGOCVariables.ATOMICITY_INPUT_FILE* a umístěním *CGOCVariables.ATOMICITY_INPUT_FILE_LOCATION*.

Vstupní atomicity jsou přitom tvořeny RaceDetectorem, který je vytváří za pomocí data-flow analýzy FindBugsem nad instrumentovaným kódem (instrumentovaný pomocí nástroje ConTest).

Vstupní instrumentované třídy, které jsou použity jako jeden ze vstupů programu, musí být umístěny v místě, odkud FindBugs bere třídy, nad kterými provádí analýzy. Implicitně je to kořenový adresář FindBugs. Vstupní třídy programu FindBugs musí být předány prostřednictvím argumentu při spuštění. Více v návodu ke spuštění.

Výstupní XML soubor má stejnou XML strukturu jako vstupní soubor. Je naplněn atomicitami, ze vstupního souboru, které obsahují synchronizační instrukce a je identifikován názvem *CGOCVariables.ATOMICITY_OUTPUT_FILE* a uživatelem definovaným umístěním *CGOCVariables.ATOMICITY_INPUT_OUTPUT_LOCATION*.

Struktura vstupního a výstupního xml je definována pomocí atomicity.dtd 4.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Atomicity by Zdenek -->
<!ELEMENT ATOMICITY (SIMPLEATOM | DOUBLEATOM | TRIPLEATOM | MULTIATOM)*>
<!ELEMENT SIMPLEATOM (BEGIN, END)>
<!ELEMENT DOUBLEATOM (BEGIN, END, END)>
<!ELEMENT TRIPLEATOM (BEGIN, END, END, END)>
<!ELEMENT MULTIATOM (BEGIN, END+)>
<!ELEMENT BEGIN EMPTY>
<!ELEMENT END EMPTY>
<!ATTLIST ATOMICITY correctRun CDATA "true">
<!ATTLIST SIMPLEATOM violatedCount CDATA "0">
<!ATTLIST DOUBLEATOM violatedCount CDATA "0">
<!ATTLIST TRIPLEATOM violatedCount CDATA "0">
<!ATTLIST MULTIATOM violatedCount CDATA "0">
<!ATTLIST BEGIN loc CDATA "">
<!ATTLIST BEGIN mode (read | write) "read">
<!ATTLIST END loc CDATA "">
<!ATTLIST END mode (read | write) "read">
```

Obrázek 4.2: Obsah atomicity.dtd.

Pro představu jak vypadají atomicity v praxi uvádím příklad na obrázku 4.3. O atomicitě z obrázku 4.3 se dá říct, že má začátek definovaný pomocí `BEGIN loc="..."`. Z toho vyplývá, že atomicita patří do zdrojového textu třídy *TestSwitch.java* metody metoda1() a začíná na 35.řádku 3. pro ConTest významnou byte-kódovou instrukcí, která je typu „zápis“. XML zápis nám dále říká, že atomicita má tři konce: první z nich patří do stejného zdrojového textu a metody jako začátek. Tento konec je typu „výstup“ a „-2 1“ nám říká, že ho nenalezneme ve zdrojovém textu (může jít např. o neoštřenou vyjímku). Druhý konec nám říká, že patří do stejného zdrojového textu a metody jako začátek. Tento konec je typu „čtení“ a „40 1“ nám říká, že jde o 1. pro ConTest významnou byte-kódovou instrukci

40. řádku zdrojového textu. Ve FindBugs tyto významné instrukce přitom poznáme díky instrumentaci, kterou si ConTest program označkoval. Třetí (poslední) konec atomicity je obdoba druhého s rozdílným typem (výstupní) a místem (43 1).

```
<TRIPLEATOM>
<BEGIN loc="UNPROVIDED TestSwitch.java metoda1(int) 35 3" mode="WRITE"/>
<END loc="UNPROVIDED TestSwitch.java metoda1(int) -2 1" mode="EXIT"/>
<END loc="UNPROVIDED TestSwitch.java metoda1(int) 40 1" mode="READ"/>
<END loc="UNPROVIDED TestSwitch.java metoda1(int) 43 1" mode="EXIT"/>
</TRIPLEATOM>
```

Obrázek 4.3: Příklad atomicity se třemi konci.

Podstatné informace, které tyto xml zápisy atomicit obsahují pro CFG analýzu jsou:

- `BEGIN loc="..."` – určuje začátek v CFG, odkud se má začít hledat synchronizace
- `END loc="..."` – určuje jeden z možných konců atomicity v CFG
- počet konců – určuje kolik cest v CFG vede od BEGIN k END

Po tomto představení vstupů a výstupu programu je čas se podívat na vytváření grafu volání, který je využíván u CFG analýzy. Call graph, jak je jinak grafu volání přezdíváno, je tvořen postupně pro každou třídu, nad kterou se má ověřování korektnosti léčení uskutečnit. Tyto třídy přitom musí být programu předány jako parametr při spouštění FindBugs.

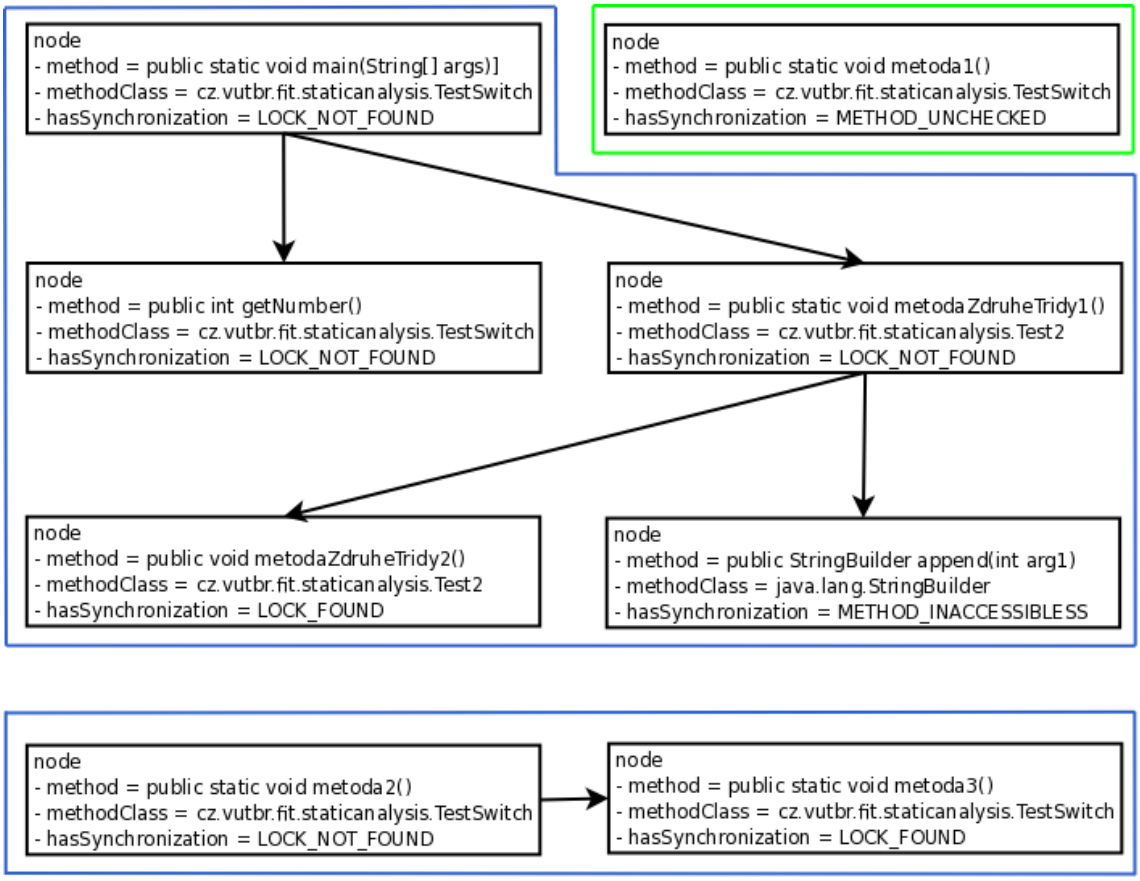
4.1 Tvorba grafu volání

Graf volání je reprezentován uzly, což jsou metody vstupních instrumentovaných tříd a hranami představující jednotlivé „invoke“ instrukce v metodách. Hrany spojují vždy dva uzly. Jeden uzel hrany je považován za zdrojovou metodu. Druhý uzel hrany je považován za cílovou metodu. Zdrojový uzel je určen podle metody ve které byla „invoke“ instrukce nalezena. Uzel cílový je získán z parametru „invoke“ instrukce a představuje metodu, která je touto instrukcí volána. Uzly kromě zdrojové/cílové metody ještě obsahují název třídy do které patří a *stav metody. Stav metody (hasSynchronization)* přitom určuje zda je metoda pro kterou byl uzel vytvořen bezpečná. Tato informace o bezpečnosti metody přitom může nabývat čtyř hodnot:

- CGOCVariables.MethodState.LOCK_FOUND – metoda obsahuje sychronizační instrukci ⇒ je nebezpečná.
- CGOCVariables.MethodState.LOCK_NOT_FOUND – metoda neobsahuje synchronizační instrukci ⇒ je bezpečná.
- CGOCVariables.MethodState.METHOD_UNCHECKED – metoda nebyla testována = inicializační hodnota ⇒ je nebezpečná.
- CGOCVariables.MethodState.METHOD_INACCESSIBLESS – metoda je nedostupná = metoda patří do třídy, která nebyla zadána jako jedna ze vstupních instrumentovaných tříd a program ji tedy nemůže zkontrolovat ⇒ je nebezpečná.

Pro lepší představu o tom, jak takový graf volání může vypadat, je zde přiložen obrázek 4.4. Na obrázku je zobrazen graf v průběhu jeho vytváření. Vytváření grafu volání je zrovna ve fázi, kdy byly prohledány všechny metody z třídy *cz.vutbr.fit.staticanalysis.Test2* a v současné době se prohledávájí metody od třídy *cz.vutbr.fit.staticanalysis.TestSwitch*. Zatím z této třídy byly prohledány metody: *main()*, *getNumber()*, *metoda2()* a *metoda3()* ⇒ hodnota *stavu metody* u uzlů ke který tyto metody patří je různá od CGOCVariables.MethodState.METHOD_UNCHECKED. Tuto hodnotu má v příkladu 4.4 pouze zeleně ohraničená metoda1, která ještě nebyla prohledána ⇒ nebyl pro ni ještě vytvořen graf volání. Nutno podotknout, že v tomto grafu volání přitom nebyla prohledána ještě jedna metoda. Tato druhá neprohledaná metoda je *append()* z třídy *java.lang.StringBuilder*. Stav metody tohoto uzlu je CGOCVariables.MethodState.METHOD_INACCESSIBLESS a je to z důvodu, že metoda nepatří ani do jedné ze vstupních instrumentovaných tříd.

Graf volání má v současné době tři nezávislé časti, které jsou v obrázku 4.4 ohraničeny modrou a zelenou barvou. Jako relativně bezpečné metody jsou považovány metody, jejichž stav metody je roven CGOCVariables.MethodState.LOCK_NOT_FOUND. Při zjišťování skutečné bezpečnosti metody je přitom potřeba vzít v potaz i stavy metod na které se, z uzlu pro který chci skutečnou bezpečnost zjistit, mohu dostat (ať už přímo nebo přes jiné uzly). Za skutečně bezpečnou je tedy v obrázku 4.4 považována jen metoda getNumber() z třídy *cz.vutbr.fit.staticanalysis.TestSwitch*.



Obrázek 4.4: Příklad jak může vypadat graf volání v průběhu jeho vytváření

Samotné vytváření grafu volání vypadá tak, že metodě, která graf volání vytváří jsou postupně jako parametr předávány ClassContexty (všechny známé fakty o třídě získané při inicializaci pomocí BCEL) všech vstupních instrumentovaných tříd. Tato metoda nad každou takto získanou třídou provede následující postup:

Vytvoří uzly pro všechny metody, které jsou v dané třídě obsaženy a které ještě v grafu volání nemají uzel vytvořen. Vytvoření uzlu se skládá z vytvoření nového objektu, kterému je přiřazena metoda kterou má uzel reprezentovat, třída do které metoda patří a *stav metody* nastaví na hodnotu METHOD_UNCHECKED = netestováno. Pak každou metodu prohlídne. Prohlížení představuje vytvoření ControlFlowGraph (CFG = BCEL abstrakce metody). Přičemž CFG je složen z tzv. BasicBlocků (BCEL abstrakce pro seskupení několika byte-kódových instrukcí), které začne sekvenčně všechny prohledávat. Před prohlížením nastaví *stav metody* na CGOCVariables.MethodState.LOCK_NOT_FOUND. Při prohlížení se vždy prohlédnou všechny instrukce daného BasicBlocku a přistoupí se k dalšímu. Po prohledání všech BasicBlocků jedné metody se přejde k další, až jsou prohledány všechny z třídy a může se přistoupit k třídě další.

Veškerá pozornost při prohledávání je zaměřena na dva typy byte-kódových instrukcí:

- Synchronizační instrukce (MONITORENTER) – při nalezení alespoň jedné z nich je *stav metody* nastaven na CGOCVariables.MethodState.LOCK_FOUND v opačném případě (když v celé metodě není nalezena ani jedná z nich zůstane *stav metody* nastaven na CGOCVariables.MethodState.LOCK_NOT_FOUND
- Invoke instrukce (InvokeInstruction) – určuje, že metoda volá jinou metodu, což může vést k vytvoření nové hrany nebo uzlu.

Detekce invoke instrukce ovlivňuje to, zda se bude v grafu volání vytvářet nový uzel nebo hrana. Tvorba nového uzlu přitom závisí na několika věcech:

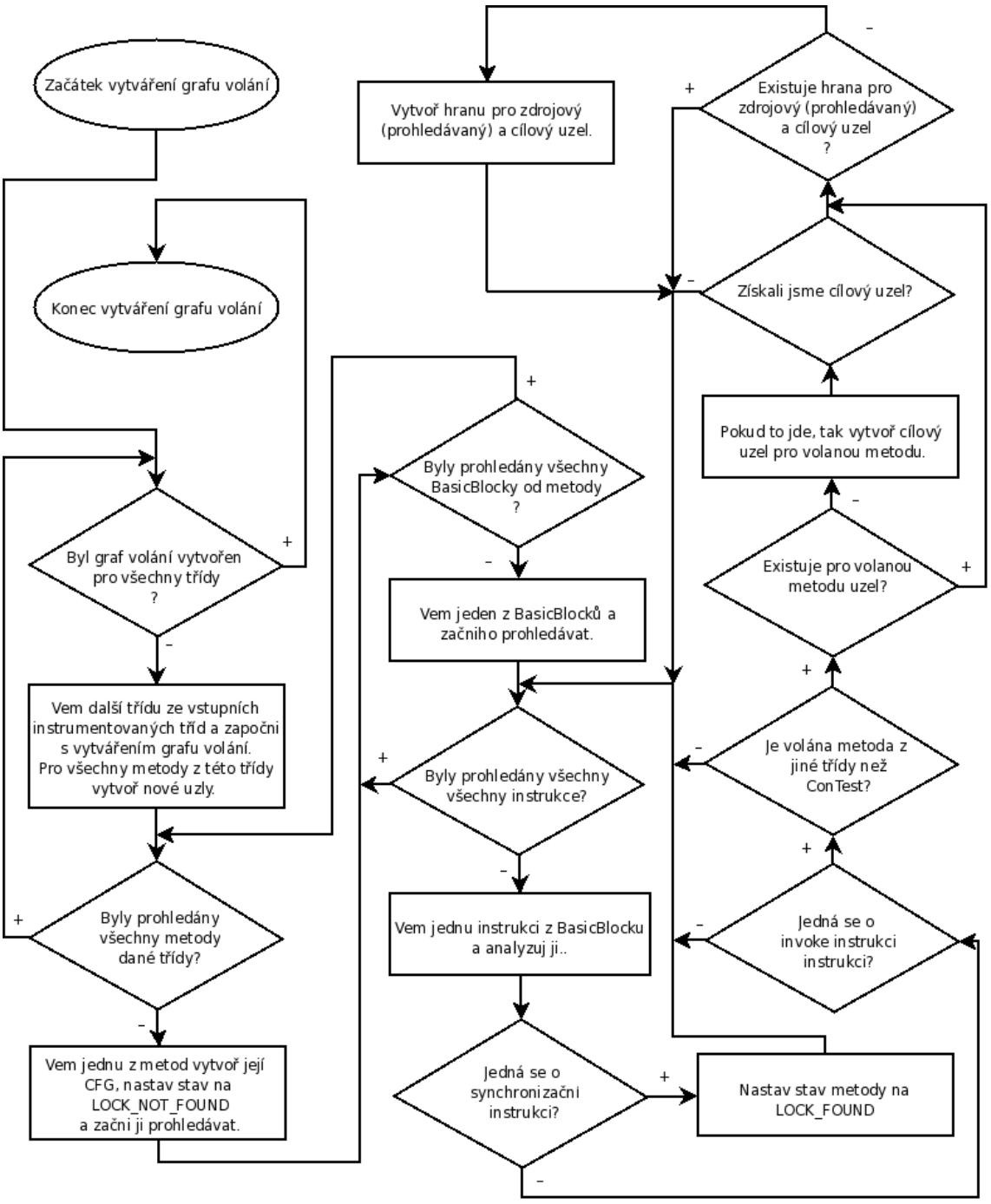
- Cílová (volaná) metoda nesmí být od ConTest (com.ibm.contest.runtimecore.Manager) z důvodu, že tyto metody jsou dílem instrumentace, které v originálním kódu neexistují a my je tedy nechceme testovat.
- Pro cílovou (volanou) metodu ještě neexistuje žádný uzel v grafu volání.
- Pro cílovou metodu (volanou) musí jít získat třída do které patří.
- Ve třídě do které údajně patří musí skutečně existovat.

Při vytváření uzlu se přitom přihlíží na to, zda je cílová (volaná) metoda z jedné ze vstupních instrumentovaných tříd. Pokud tomu tak je, tak je uzel založen se stavem metody = CGOCVariables.MethodState.METHOD_UNCHECKED v opačném případě, se jedná o metodu, kterou nemůžeme prohledat a stav metody v nově vznikajícím uzlu je nastaven na CGOCVariables.MethodState.METHOD_INACCESSIBLESS.

Vytváření hrany závisí na:

- Pro volanou metodu byl vytvořen nový cílový uzel nebo je znám již existující cílový uzel ⇒ nejedá se o ConTest metodu.
- V grafu volání ještě neexistuje hrana pro zdojový (uzel od metody ve které byla nalezená invoke instrukce) a cílový uzel.

Výše popsaný postup tvorby grafu volání je pro lepší pochopení ještě zobrazen ve vývojovém diagramu z obrázku 4.5.



Obrázek 4.5: Vývojový diagram tvorby grafu volání

Po průchodu vývojovým diagramem z obrázku 4.5 má program k dispozici graf, který zobrazuje vztahy mezi jednotlivými metodami. Uzly metod přitom vlastní informaci, zda obsahují nějakou synchronizační instrukci, čehož je pak využito v prohledávání atomických sekcí.

Na závěr této podkapitoly bych ještě dodal, že jsem si při tvorbě grafu volání vybíral ze dvou možných přístupů k problematice. První z nich je založen na tvorbě malých grafů volání až při analýze, když se narazí na nějakou invoke instrukci. Druhý (implementovaný) je založen na vytvoření grafu volání pro všechny metody ze vstupních instrumentovaných tříd již před analýzou. Nevýhoda výše popsaného a naimplementovaného postupu je v tom, že generuje graf volání nad všemi metodami, které patří do vstupních instrumentovaných tříd. ⇒ Větší paměťová náročnost než první zmiňovaný postup. Takto vytvořený graf volání taky může obsahovat části, které nejsou v analýze využity. Na druhou stranu jeho výhoda oproti prvně zmiňovanému postupu tkví v tom, že samotná analýza, pokud bude prováděna nad velkým počtem vstupních atomicit, bude rychlejší. Při prvně zmiňovaném postupu je totiž pravděpodobné, že by se určité části grafu volání, generovaly víckrát než jednou, což by brzdilo provádění analýzy.

4.2 Hledání zámků v jednotlivých atomických sekcích

Hledání zámku v jednotlivých atomických sekcích je analýza založená na hledání chybových vzorů při procházení Control Flow Grafu (CFG). Jedná se o jádro celého programu pro ověřování korektnosti léčení. Na obrázku 4.6 je vidět, jak tato níže popisovaná analýza funguje.

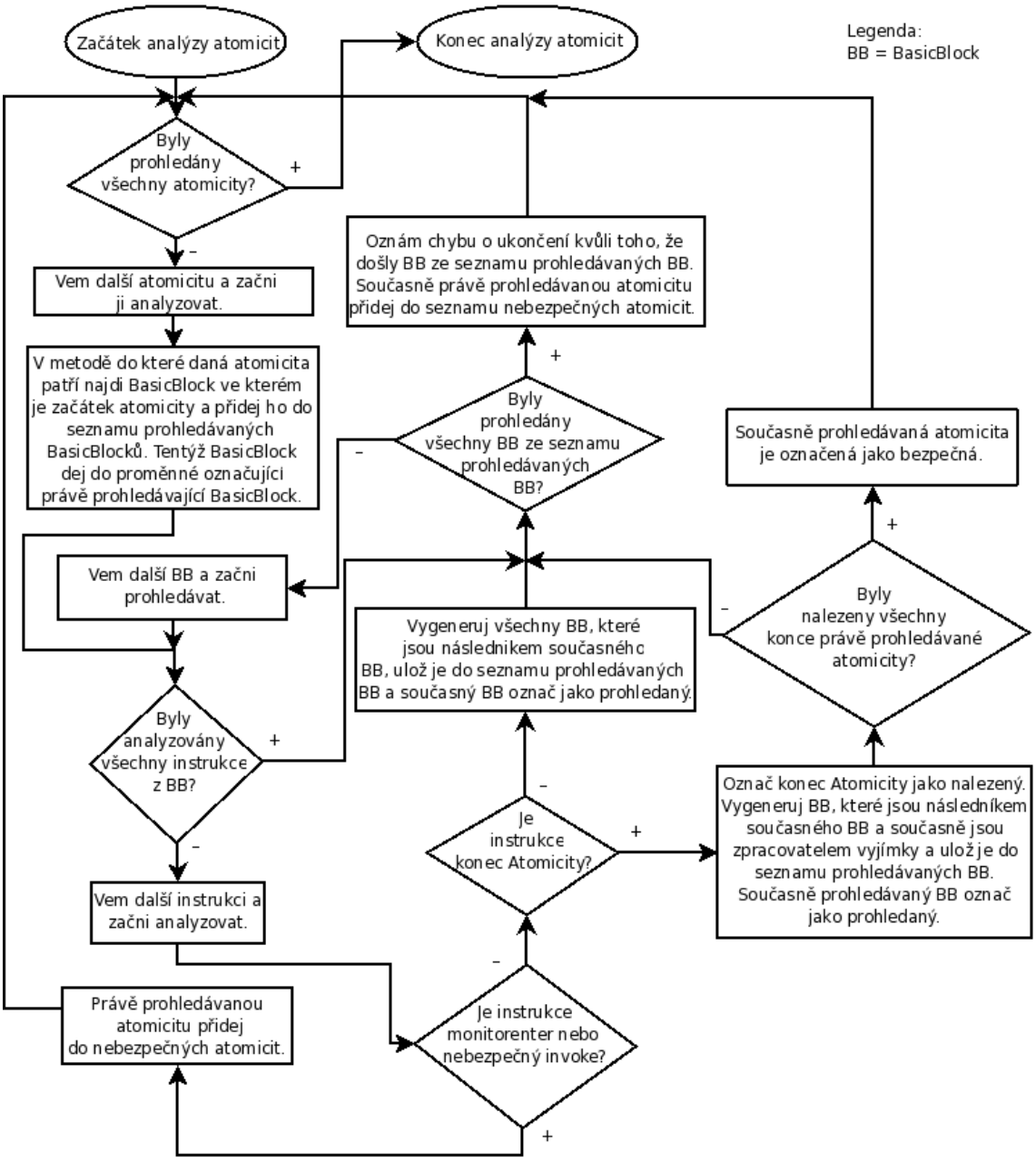
Program si načte atomicity ze vstupního souboru do kolekce. Z této kolekce pak bere jednu po druhé a provádí jejich analýzu následujícím způsobem. U každé atomicity je zjištěn její počátek, všechny konce a třída společně s metodou do které patří. Následně je v metodě a třídě do které zvolená atomicita patří spuštěno vyhledávání synchronizace.

Pro metodu ve které se bude synchronizace hledat je vytvořen CFG. V takto vytvořeném CFG se začnou sekvenčně prohledávat BasicBlocky za účelem nalezení BasicBlocku ve kterém začíná atomicita, která je právě předmětem ověřování. Každý z BasicBlocků je přitom prohledáván stylem, kdy je testována každá instrukce BasicBlcoku na to, zda je instrukcí *ldc*, která má jako svůj parametr identifikaci začátku atomicity (část z atomicity mezi uvozkami u buňky BEGIN loc="...").

Po takto nalezeném BasicBlocku si analýza zjistí další BasicBlocky do kterých se může program z právě prohledaného BasicBlocku dostat a uloží si je do seznamu v budoucnu prohledávaných BasicBlocků. V tomto seznamu jsou postupně uloženy všechny vygenerované BasicBlocky, kam se program z počátečního BasicBlocku přes rozgenerované BasicBlocky může dostat. Každý BasicBlock je přitom do seznamu přiřazen vždy s vlastností, která určuje, že ještě nebyl prohledán. Tato vlastnost se mění na stav „prohledáno“ až poté, kdy je BasicBlock analyzován a jsou z něj rozgenerovány další BasicBlocky.

Analýza BasicBlocku je založena na prohledání všech instrukcí prohledávaného BasicBlocku. Když je při prohledávání BasicBlocku instrukce rozpoznána jako monitorenter, je právě ověřovaná atomicita prohlášena za nebezpečnou. Takováto nebezpečná atomicita je uložena do kolekce nebezpečných atomicit, které jsou na konci programu zapsány do výstupního souboru.

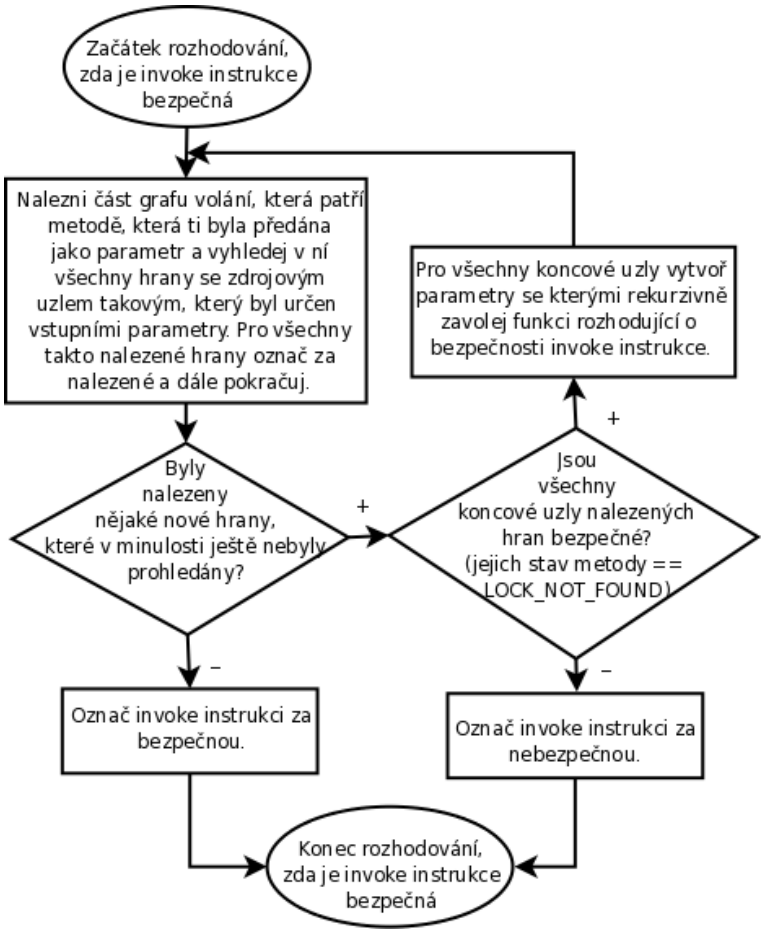
V případě, že je instrukce určena jako invoke instrukce, která volá metodu z jiné třídy než *com.ibm.contest.runtimecore.Manager* (ConTest), provede se kontrola nad dříve vytvořeným grafem volání. Kontrola spočívá v rozhodnutí o tom, zda část grafu volání, kam se může program pomocí detekované invoke instrukce dostat je bezbečná (neobsahuje nějakou nebezpečnou metodu = metoda s monitorenter instrukcí, neprohledaná metoda nebo metoda nedosažitelná) či nikoli.



Obrázek 4.6: Vývojový diagram analýzy atomicit

Metodě, která toto zjišťování provádí je předán název metody a třída ve které byla invoke instrukce nalezena. V patřičném podgrafu se najdou všechny hrany se zdrojovým uzlem určeným získaným názvem a třídou metody. Pro tyto hrany se zkontroluje zda je cílová metoda bezpečná. Tato informace (stav metody) jde vyčíst přímo z uzlů obsažených v grafu volání. Pokud je zjištěno, že některá z cílových metod je nebezpečná, je postup podobný jako při detekci monitorenter přímo v atomicitě. Prohledávání se ukončí s označením atomicity za nebezpečnou. V opačném případě, se vezme cílový uzel a též metodě, která má na starosti prohledávání grafu volání se dá jako zdroj třída a název cílové metody.

Tímto způsobem se rekurzivně rozgeneruje malý podgraf ve kterém nesmí být nalezen ani jeden uzel s nebezpečnou metodou, aby se mohlo prohlásit, že pomocí původně nalezené invoke instrukce se může program dostat pouze do bezpečných částí kódu. Problematiku rozhodování o bezpečnosti invoke instrukce pro upřesnění ještě zobrazuje obrázek 4.7



Obrázek 4.7: Rozhodnutí o bezpečnosti nalezené invoke instrukce

Jako poslední druh instrukcí, které mají pro analýzu nějaký význam a nesmí být tím pádem ignorovány jsou instrukce LDC a LDC_W. Při detekci obou těchto instrukcí se musí zjistit zda parametr této instrukce není shodný s některým z konců právě prohledávané atomicity (END loc="..."). Pokud se o konec nejedná, instrukce je ignorována a pokračuje se v analýze. V opačném případě je analýza konkrétního BasicBlocku zastavena a při rozgenerování nových BasicBlocků z toho ve kterém byl nalezen konec atomicity se rozgenerují jen ty, které spadají do ošetřování podmínek. Konec, který byl přitom nalezen se označí jako nalezený.

Celá tato analýza může končit dvěma způsoby:

- Při analýze jsou nalezeny všechny konce atomicity. Mezi začátkem a žádným z konců přitom není nalezen ani jeden nebezpečný invoke nebo monitorenter instrukce. ⇒ Atomicita je prohlášena za bezpečnou a neobjeví se ve výstupním souboru.

- Při analýze je mezi začátkem a jedním z konců atomicity nalezena nebezpečná invoke instrukce nebo instrukce monitorenter. ⇒ Atomicita je prohlášena za nebezpečnou a objeví se ve výstupním souboru.

Po analýze všech atomicit ze vstupního souboru jsou atomicity ve kterých byla nalezena synchronizační instrukce nebo nebezpečný invoke uloženy do výstupního souboru a program končí.

V současné chvíli by měl mít čtenář ucelený pohled na řešenou problematiku společně s obrazem jak je ověřování korektnosti léčení pomocí nástroje FindBugs v této práci implementováno. V následující kapitole 5 se může uživatel na uvedeném testu přesvědčit o funkčnosti analýzy. Pokud má čtenář potřebu si vyzkoušet analýzu vlastních tříd, pomůže mu k tomu obsah CD společně s návodem na spuštění analýzy, který se nachází v textové příloze práce.

Kapitola 5

Testy

V této kapitole naleznete pár testů, které ukazují funkčnost uvedených postupů a vytvořeného detektoru. Na začátku teto kapitoly jsou uvedeny zdrojové texty tříd které byly k testování využity. Jsou zde proto, aby si čtenář mohl dát jednotlivé XML výpisy do souvislostí s jejich zdrojem. Následuje výpis z konzole při instrumentaci tříd. Další podsekce je věnována vytvoření souboru s atomicitami (atomicity.xml) společně s výpisem atomicit nad kterými bude analýza provedena. Pak je zde ukázána úprava souboru CGOCVariables.java. V závěru kapitoly je příklad spuštění detektoru společně s odkazem na výstupní atomicity.

5.1 Testovací třídy

Třídy *(TestHA, TestHA2 a TestHA3)* zobrazeny na obrázcích 5.2, 5.1 a 5.3 byly použity k testování. Třídy *TestHA a TestHA3* byly vytvořeny tak aby, obsahovaly sdílené proměnné *(sidelnaPromenna)*, ke kterým je z metod *zamknutiSMonitorem(), getPromenna(), metoda1() a main()* přistupováno. Metody *zamknutiSMonitorem()*, které jsou obsaženy v třídách *TestHA a TestHA3* obsahují synchronizaci, kterou se bude program snažit najít. Metoda *metoda2()* z třídy *TestHA* je zde společně s metodou *testHA2CalltestHA3()* z třídy *TestHA2* proto, aby byla při analýze ukázána i funkčnost prohledávání grafu volání. Návrh testovacích tříd byl proveden tímto způsobem z důvodu, aby vytvoření vstupních atomicit, které se zaměřují na sdílené proměnné obsahovaly jak bezpečné, tak i nebezpečné atomicity. Testovací třídy při analýze postihují využití CFG analýzy i prohledávání Call grafu.

```
01: package cz.vutbr.fit.staticanalysis;
02: public class TestHA2 {
03:   public TestHA2() {
04:   }
05:   public static void testHA2CalltestHA3(){
06:     TestHA3.zamknutiSMonitorem();
07:   }
08: }
```

Obrázek 5.1: Zdrojový kód TestHA2.java.

```
01: package cz.vutbr.fit.staticanalysis;
02: public class TestHA {
03:   static int sdilenaPromenna = 0;
04:   public TestHA() {
05:   }
06:   public static void zamknutiSMonitorem(){
07:     Object zamek3 = new Object();
08:     synchronized(zamek3){
09:       sdilenaPromenna++;
10:     }
11:   }
12:   public static int getPromenna(){
13:     return sdilenaPromenna;
14:   }
15:   public static void metoda2(){
16:     TestHA2.testHA2CalltestHA3();
17:   }
18:   public static void metoda1(int prepinac){
19:     sdilenaPromenna++;
20:     if (prepinac == 0){
21:       System.out.println("sdilenaPromenna = " + getPromenna());
22:     }else{
23:       metoda2();
24:       sdilenaPromenna++;
25:       prepinac++;
26:     }
27:   }
28:   public static void main(String[] args) {
29:     System.out.println("sdilenaPromenna = " + sdilenaPromenna);
30:     zamknutiSMonitorem();
31:     System.out.println("sdilenaPromenna = " + sdilenaPromenna);
32:   }
33: }
```

Obrázek 5.2: Zdrojový kód TestHA.java.

5.2 Vytvoření instrumentovaných tříd

Na obrázku 5.4 je znázorněn test instrumentace pomocí ConTest. První červený obdelník zhora upozorňuje na pozici, kde jsem se při instrumentaci nacházel. Druhý červený obdelník upozorňuje na obsah aktuálního adresáře před začátkem instrumentace. Adresář obsahoval třídy, které jsem chtěl instrumentovat (TestHA.class, TestHA2.class, TestHA3.class). Třetí obdelník ukazuje příkaz, který byl použit k vytvoření instrumentovaných tříd. Čtvrtá zvýrazněná oblast ukazuje, že byly úspěšně instrumentovány tři třídy. Poslední červený obdelník upozorňuje na obsah adresáře po instrumentaci. Tento adresář obsahuje instrumentované třídy (TestHA.class, TestHA2.class, TestHA3.class) a zálohu neinstrumentovaných tříd (TestHA.class_backup, TestHA2.class_backup, TestHA3.class_backup).

```
01: package cz.vutbr.fit.staticanalysis;
02: public class TestHA3 {
03:   static int sdilenaPromenna = 0;
04:   public TestHA3() {
05:   }
06:   public static void zamknutiSMonitorem(){
07:     Object zamek3 = new Object();
08:     synchronized(zamek3){
09:       sdilenaPromenna++;
10:     }
11:   }
12: }
```

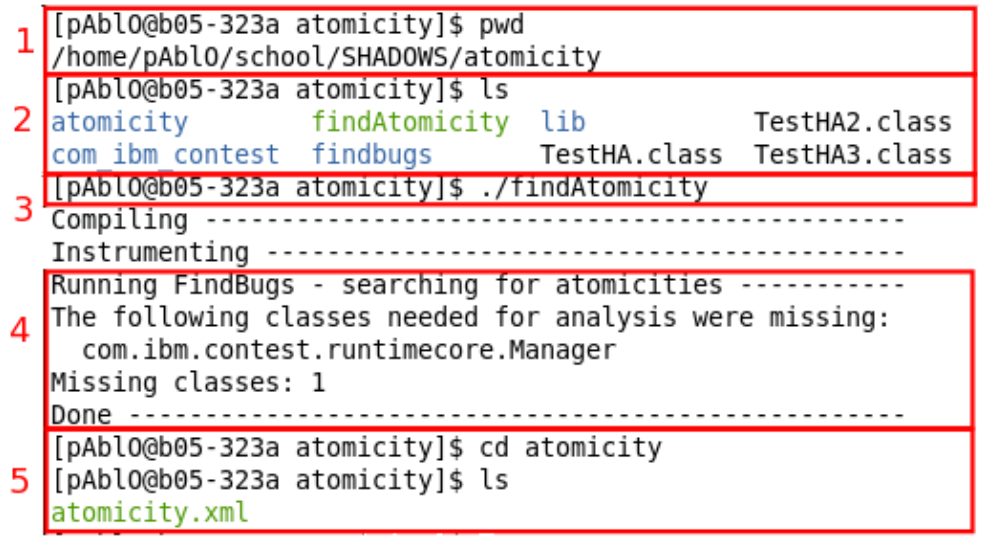
Obrázek 5.3: Zdrojový kód TestHA3.java.

```
1 [pAblO@b05-323a instrumentedTestHA]$ pwd
  /home/pAblO/school/SHADOWS/findbugs/findbugs/instrumentedTestHA
2 [pAblO@b05-323a instrumentedTestHA]$ ls
  TestHA.class  TestHA2.class  TestHA3.class
3 [pAblO@b05-323a instrumentedTestHA]$ java -classpath /home/pAblO/scho
  ol/SHADOWS/JavaConTest/Lib/ConTest.jar com.ibm.contest.instrumentatio
  n.Instrument /home/pAblO/school/SHADOWS/findbugs/findbugs/instrumente
  dTestHA
  >>> ConTest: ConTest for Java, version: 2.6.5.3
  >>> ConTest:    build: Thu Dec 27 09:17:10 CET 2007
  >>> ConTest: (c) Copyright IBM Corporation (1999, 2007), ALL RIGHTS R
  ESERVED.
  >>> ConTest: properties file not specified as JVM property, looking a
  t current working directory
  >>> ConTest: properties file not found in current working directory,
  looking in the classpath
  >>> ConTest: Found properties file: /home/pAblO/school/SHADOWS/JavaCo
  nTest/Lib/KingProperties
  >>> ConTest: Scanned source dir ., found 0 source files
  *** ConTest warning: no source file found in specified source directo
  ries
  >>> ConTest: created ConTest output directory: [/home/pAblO/school/SH
  ADOWS/findbugs/findbugs/instrumentedTestHA/com_ibm_contest]
  >>> ConTest: not writing any coverage tasks
  >>> ConTest: target classes not specified in properties file, instrum
  enting everything
4 >>> ConTest: instrumenting all locations in specified classes
  /home/pAblO/school/SHADOWS/findbugs/findbugs/instrumentedTestHA: 3 cl
  asses successfully instrumented
5 [pAblO@b05-323a instrumentedTestHA]$ ls
  com_ibm_contest      TestHA2.class         TestHA3.class_backup
  TestHA.class         TestHA2.class_backup
  TestHA.class_backup  TestHA3.class
```

Obrázek 5.4: Test Instrumentace

5.3 Vytvoření vstupního souboru s atomicitami

Obrázek 5.5 zobrazuje vytvoření vstupního XML souboru s atomimicitami. Atomicity jsou přitom tvořeny pomocí části patřící RaceDetectoru. K tvorbě byl využit skript s názvem findAtomicity. První červený obdelník ukazuje na pozici, kde byl test vytváření atomicit proveden. Druhý červený obdelních upozorňuje na obsah adresáře atomicity. Adresář obsahuje jiný adresář atomicity, kde se po spuštění skriptu nacházel vstupní XML soubor s atomicitami (atomicity.xml). Dále tentýž obdelník upozorňuje na skript findAtomicity a instrumentované třídy (TestHA.class, TestHA2.class, TestHA3.class) ze kterých se budou atomicity vytvářet. Tředí obdelník ukazuje spuštění skriptu. Čtvrtý obdelník upozorňuje na to, že v analýze chybí třída od ConTest, ale i přesto byl soubor s atomicitami vytvořen. Varováním o chybějící ConTest třídě se uživatelé zabývat nemusí, jelikož atomicity nad ConTest zpravidla vytvářet není předmětem analýzy. Z důvodu, že program ovšem při vytvoření atomicit na tuto třídu v instrumentovaných třídach narazil, považuje to za podezřelé. Pátý obdelník ukazuje na výsledný produkt vytváření atomicit (atomicity.xml).



Obrázek 5.5: Test vytvoření vstupního xml souboru s atomicitami

Obsah vygenerovaného atomicity.xml v tomto testu vypadá takto:

```
<ATOMICITY>
<DOUBLEATOM>
<BEGIN loc="UNPROVIDED TestHA.java metoda1(int) 24 1" mode="READ"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) -2 1" mode="EXIT"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) 24 2" mode="WRITE"/>
</DOUBLEATOM>

<DOUBLEATOM>
<BEGIN loc="UNPROVIDED TestHA.java metoda1(int) 19 2" mode="READ"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) -2 1" mode="EXIT"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) 19 3" mode="WRITE"/>
</DOUBLEATOM>
```

```
<DOUBLEATOM>
<BEGIN loc="UNPROVIDED TestHA.java zamknutiSMonitorem() 9 1" mode="READ"/>
<END loc="UNPROVIDED TestHA.java zamknutiSMonitorem() 9 2" mode="WRITE"/>
<END loc="UNPROVIDED TestHA.java zamknutiSMonitorem() -2 1" mode="EXIT"/>
</DOUBLEATOM>


<DOUBLEATOM>
<BEGIN loc="UNPROVIDED TestHA.java main(java.lang.String[]) 29 4" mode="READ"/>
<END loc="UNPROVIDED TestHA.java main(java.lang.String[]) -2 1" mode="EXIT"/>
<END loc="UNPROVIDED TestHA.java main(java.lang.String[]) 31 2" mode="READ"/>
</DOUBLEATOM>

<TRIPLEATOM>
<BEGIN loc="UNPROVIDED TestHA.java metoda1(int) 19 3" mode="WRITE"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) 24 1" mode="READ"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) -2 1" mode="EXIT"/>
<END loc="UNPROVIDED TestHA.java metoda1(int) 27 1" mode="EXIT"/>
</TRIPLEATOM>


<DOUBLEATOM>
<BEGIN loc="UNPROVIDED TestHA3.java zamknutiSMonitorem() 9 1" mode="READ"/>
<END loc="UNPROVIDED TestHA3.java zamknutiSMonitorem() 9 2" mode="WRITE"/>
<END loc="UNPROVIDED TestHA3.java zamknutiSMonitorem() -2 1" mode="EXIT"/>
</DOUBLEATOM>
</ATOMICITY>
```

5.4 Úprava CGOCVariables.java

Před spuštěním analýzy je potřeba si nastavit ve zdrojových textech jisté konstanty. Úprava musí být provedena v souboru CGOCVariables.java a v testu byla provedena takto 5.6.

```
// This constant determine input file position.
public final static String ATOMICITY_INPUT_FILE_LOCATION =
    "/home/pAbl0/school/SHADOWS/atomicity/atomicity";

// This constant determine input file name.
public final static String ATOMICITY_INPUT_FILE =
    "atomicity.xml";

// This constant determine input file position.
public final static String ATOMICITY_OUTPUT_FILE_LOCATION =
    "/home/pAbl0/school/SHADOWS/atomicity/atomicity";

// This constant determine output file name.
public final static String ATOMICITY_OUTPUT_FILE =
    "atomicityWithLock.xml";
```

Obrázek 5.6: Nastavení patřičných cest v CGOCVariables.java

5.5 Export upraveného balíčku Healing_assurance

Po úpravě zdrojového souboru CGOCVariables.java je třeba změny uložit a balíček vyexportovat na patřičné místo, aby ho FindBugs při spuštění našel. Při testování byl balíček vyexportován na místo označené prvním červeným obdelníkem z obrázku 5.7. Druhý obdelník z tohoto obrázku zobrazuje obsah adresáře /plugin po exportu balíčku.

```
1 [pAblO@b05-323a plugin]$ pwd
  /home/pAblO/school/SHADOWS/findbugs/findbugs/plugin
2 [pAblO@b05-323a plugin]$ ls
  coreplugin.jar  CVS  Healing_assurance.jar
```

Obrázek 5.7: Export balíčku Healing_assurance

5.6 Spuštění detektoru

Po všech těchto přípravách bylo možné program pro ověřování korektnosti léčení spustit. Tato aplikace, která provádí analýzu nad atomickými sekcemi z výše uvedených tříd byla spuštěna způsobem zobrazeným na obrázku 5.8. První a druhý červený obdelník z obrázku ukazuje umístění souboru atomicity.dtd. Třetí část obrázku upozorňuje na místo odkud se bude detektor spouštět. Obsah čtvrtého obdelníku představuje spuštění analýzy nad instrumentovanými třídami, které jsou obsaženy v archívu instrumentedTestHA.jar. Pátý obdelník hlásí, že byl úspěšně načten vstup a zapsán výstup. Poslední část hlásí, místo zápisu výstupního souboru a pro test nepodstatné varování o chybějící třídě ConTestu.

```
1 [pAblO@b05-323a lib]$ pwd
  /home/pAblO/school/SHADOWS/findbugs/findbugs/bin/lib
2 [pAblO@b05-323a lib]$ ls
  atomicity.dtd
  [pAblO@b05-323a lib]$ cd ..
3 [pAblO@b05-323a bin]$ pwd
  /home/pAblO/school/SHADOWS/findbugs/findbugs/bin
4 [pAblO@b05-323a bin]$ ./findbugs -textui -visitors HealingAssuranceBy
  SearchingMonitors /home/pAblO/school/SHADOWS/findbugs/findbugs/instru
  mentedTestHA.jar
5 Restoring atomicity from file...
  Saving atomicity to the file...
  Atomic sections which inculde locks were note to the output file:/hom
  e/pAblO/school/SHADOWS/atomicity/atomicity/atomicityWithLock.xml
6 It's all over.
  The following classes needed for analysis were missing:
    com.ibm.contest.runtimecore.Manager
  Missing classes: 1
```

Obrázek 5.8: Test spuštění detektoru

Ve výsledném souboru atomicityWithLock.xml byly po analýze uvedeny dvě atomicity (4. a 5.) ze vstupního souboru atomicit.xml, což potvrzuje správný běh programu.

Kapitola 6

Závěr

V práci jsme se seznámili s řešením problému ověřování korektnosti léčení v Java programech pomocí nástroje FindBugs, stručnou charakteristikou statické analýzy a přehledem nástrojů, které statickou analýzu využívají. Práci jsem začal studií problematiky, kterou se zabývá projekt SHADOWS. Následovalo vytyčení cílů v podobě vytvoření programu pro ověřování korektnosti léčicí akce zjištěním, zda uzamykaná atomicita obsahuje jinou uzamykací instrukci (monitorenter). Pak jsem se dal do studia statické analýzy v podobě návštěvy kurzu Formální analýza a verifikace. Návštěvu kurzu jsem přitom doplňoval studiem níže uvedených literárních zdrojů. Na tuto teoretickou přípravu jsem přitom začat plynule navazovat zkoumáním programu pro statickou analýzu FindBugs. Posléze jsem se dal do práce na detektoru, kterou jsem v hojné míře konzultoval s Bc. Zdeňkem Letkou, kterému tímto ještě jednou děkuji za pomoc.

Odevzdaná verze programu je schopna nalézt instrukci monitorenter v byte-kódu, který je určen vstupující atomicitou a instrumentovanou třídou, do které tato atomicita patří. Na základě tohoto nálezu je pak program schopen prohlásit zda je daná atomicita potenciálně bezpečná či nikoliv. Skutečnou bezpečnost atomicity tento nástroj odhalit ještě nedokáže. Program umí pouze zjišťovat přítomnost byte-kódové instrukce monitorenter v části CFG patřící atomicitě a v případných jiných metodách do kterých se z této atomicity může program pomocí instrukce invoke dostat. To, zda ovšem v metodě může uváznutí nastat, závisí i na jiných blokujících přístupech jako je např. volání metody wait() nad objektem. V detektoru pak taky není zcela ošetřena dědičnost. Vytvoření detektoru, který bude schopen odhalit ostatní přístupy, jež by mohly vést k uváznutí a uměl současně řešit problém dědičnosti považuji za svou budoucí práci na projektu SHADOWS.

Jako přínos vidím to, že jsem se pomocí této bakalářské práce naučil základům programování v Javě. Své výsledky jsem prezentoval na konferenci EEICT [19]. Dále jsem se seznámil se základy metod formální verifikace, nejvíce se statickou analýzou a pronikl jsem do problematiky tvorby detektorů pro nástroj FindBugs. Velice si také vážím zkušenosti, s prací ve fungujícím týmu vědecky pracujících lidí na mezinárodním projektu.

Literatura

[1] Eric Bruneton, Eugene Kuleshov, and Andrei Loskutov. Asm – java bytecode manipulation and analysis framework. [Online], Verze 3.1 (2007), [rev. 2007-10-29], [cit. 2008-05-08]. Dostupné na URL: `http://asm.objectweb.org/`.

[2] Oliver Burn. Checkstyle – checkstyle 4.4. [Online], Verze 4.4 (2007), [rev. 2007-12-19], [cit. 2008-05-08]. Dostupné na URL: `http://checkstyle.sourceforge.net/`.

[3] Maarten Coene. dom4j – the flexible xml framework for java. [Online], Verze 1.6.1 (2005), [rev. 2005-05-16], [cit. 2008-05-08]. Dostupné na URL: `http://dom4j.org/`.

[4] Oliver Cole, Chris Elford, and Harm Sluiman. Tptp. [Online], Verze 4.4.1 (2008), [rev. 2008-02-25], [cit. 2008-05-08].
Dostupné na URL: `http://www.eclipse.org/tptp/`.

[5] Markus Dahm. Bcel – byte code engineering library. [Online], Verze 5.2 (2006), [rev. 2006-06-03], [cit. 2008-05-08].
Dostupné na URL: `http://jakarta.apache.org/bcel/`.

[6] David Dixon-Peugh, Tom Copeland, and Xavier Le Vourch. Pmd. [Online], Verze 4.2.1 (2008), [rev. 2008-04-11], [cit. 2008-05-08].
Dostupné na URL: `http://pmd.sourceforge.net/`.

[7] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *j-CCPE*, 15(3–5):485–499, 2003.

[8] Neal Ford. Intellij idea – the most intelligent java ide. [Online], Verze 7.0.3 (2008), [rev. 2008-03-14], [cit. 2008-05-08].
Dostupné na URL: `http://www.jetbrains.com/idea/`.

[9] Ray Gans. Developer resources for java technology. [Online], Verze 6u10 (2008), [rev. 2008-05-01], [cit. 2008-05-08]. Dostupné na URL: `http://java.sun.com/`.

[10] David Hovemeyer. The architecture of findbugs. [Online], [rev. 2008-01-09], [cit. 2008-05-08].
Dostupné na URL: `http://www.cs.vassar.edu/ hovemeye/architecture.pdf/`.

[11] Ota Jirak and Zdeněk Letko. Fav - pattern based static analysis. slajdy z kurzu Formální analýza a verifikace. 2008.

[12] Bohuslav Křena. Shadows – a self-healing approach to designing complex software systems. slajdy z Pojďte dělat vědu na FIT 2008. 2008.

[13] Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *PADTAD '07*, pages 54–64. ACM, 2007.

[14] Alexander Meduna and Roman Lukáš. *Formální jazyky a překladače – Studijní opora*. Vysoké učení technické v Brně, 2006.

[15] Bill Pugh, David Hovemeyer, and Ben Langmead. Findbugs – find bugs in java programs. [Online], Verze 1.3.4 (2008), [rev. 2008-05-07], [cit. 2008-05-08]. Dostupné na URL: `http://findbugs.sourceforge.net/`.

[16] Michael I. Schwartzbach. *Lecture Notes on Static Analysis*. BRICS, Department of Computer Science, 2006.

[17] Tomáš Vojnar. Formal analysis and verification – fav 2007/2008 – 1. lecture. slajdy z kurzu Formální analýza a verifikace. 2008.

[18] Tomáš Vojnar. Synchronizace procesu. slajdy z kurzu Operační systémy. 2008.

[19] Pavel Vyvial. Healing assurance in java programs. In *Proceedings of the 14th conference Student EEICT 2008*, volume 1, pages 204–206. Vysoké učení technicke v Brně, Fakulta elektrotechniky a komunikačních technologií a Fakulta informačních technologií, 2008.

[20] Wikipedia. Mnohojazyčná webová encyklopedie – deadlock. [Online], [rev. 2008-03-30], [cit. 2008-05-08].
Dostupné na URL: `http://cs.wikipedia.org/wiki/Deadlock`.

Seznam příloh

- Návod na spuštění analýzy
- CD-R obsahující:
  - zdrojové texty práce – /Healing_assurance/src/cz/vutbr/fit/staticanalysis
  - elektronickou podobu technické zprávy – /technickaZprava.pdf
  - programovou dokumentaci – /Healing_assurance/doc
  - nástroj pro instrumentaci tříd (ConTest) – /JavaConTest/Lib/ConTest.jar
  - nástroj pro tvorbu vstupních atomicit (RaceDetector) – /atomicity/racedetector.jar
  - skript pro tvorbu vstupních atomicit – /atomicity/findAtomicity

Návod na spuštění analýzy

Před spuštěním programu pro ověřování korektního léčení je třeba mít k dispozici:

- virtuální stroj kompatibilní se Sun's JDK 1.4
- nástroj kompatibilní s FindBugs 1.3.2
- balíček Healing_assurance (balíček, který obsahuje zdrojové a binární soubory od této práce + další potřebné zdrojové soubory od RaceDetectoru a binární knihovnu od ConTest)
- výše zmiňovaný atomicity.dtd
- vstupní soubor v XML formátu definovaný podle výše zmíňovaného atomicity.dtd
- instrumentované třídy (.class soubory) do kterých patří všechny atomicity ze vstupního xml souboru

Virtuální stroj, který může poskytnout JDK 1.4 může uživatel najít např. na stránkách společnosti Sun [9]. Projekt byl přitom vyvíjen a testován pod: Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b03).

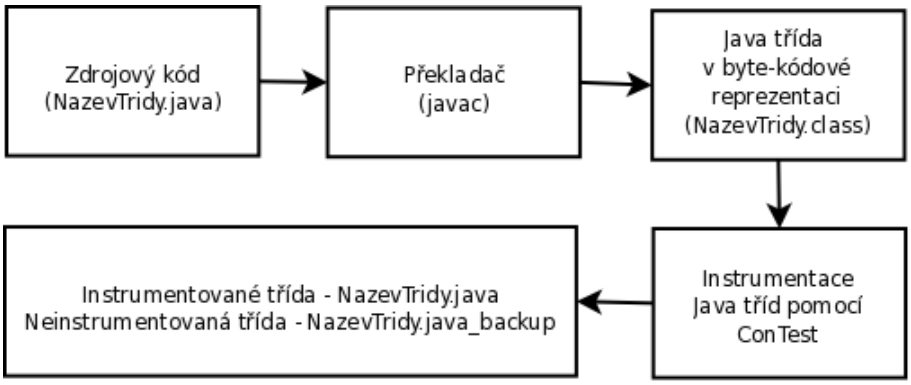
FindBugs lze získat z domovských stránek tohoto nástroje [15]. Osobně bych pro získání nástroje doporučil použít CVS. Při vývoji detektoru bylo pro získání funkčního nástroje využito následující nastavení CVN 1 v Eclipse.

Connection type:	pserver
User:	anonymous
Host	findbugs.cvs.sourceforge.net
Port:	Default
Repository path	/cvsroot/findbugs
Module:	findbugs
Tag:	HEAD

Obrázek 1: Nastavení CVN, které bylo využito při vývoji detektoru.

Balíček *Healing_assurance* je součástí CD přílohy. Přitom pouze zdrojové soubory obsažené v *Healing_assurance/src/cz/vutbr/fit/staticanalysis* jsou výsledkem této práce. Zbytek balíčku jsou k analýze potřebné knihovny a nástroje, které mají odlišné autory.

Soubor *atomicity.dtd* je součástí CD přílohy. Uživatel ho může najít v adresáři *atomicity/lib*. Tento soubor je přitom potřeba překopírovat do adresáře */bin/lib* od nástroje FindBugs. Pokud uživatel spouští FindBugs přes Eclipse, je třeba tento soubor nakopírovat přímo do adresáře */lib* v kořenovém adresáři tohoto nástroje.



Obrázek 2: Postup při instrumentaci

Instrumentace tříd zobrazena na obrázku 2 se provádí pomocí ConTest *(com.ibm.contest.instrumentation.Instrument)*, který může uživatel najít na CD příloze v adresáři *JavaConTest/Lib/ConTest.jar*. Je několik možností, jak k instrumentaci přistupovat. Pokuď má uživatel zájem se o této problematice více dozvědět, doporučuji prostudovat README.html z adresáře JavaConTest. V případě, že má uživatel zájem postupovat při instrumentaci obdobně jak tomu bylo při vývoji tohoto detektoru, musí se řídit těmito pokyny: Nejdřív musí uživatel vstoupit do adresáře, kde má připravené třídy (soubory .class) pro instrumentaci. Následně zadat tento příkaz: *java -classpath /home/pAblO/JavaConTest/Lib/ConTest.jar com.ibm.contest.instrumentation.Instrument /home/pAblO/instrumentedCode*. Přitom samozřejmě musí mít nastaveny správně patřičné cesty. První cesta je ke knihovně od ConTest a druhá k adresáři s třídami, které chce uživatel instrumentovat. Správnou instrumentaci uživatel pozná tak, že se v adresáři prvotní třídy přepíší instrumentovanými a neinstrumentované se přejmenují na staryNazev.class_backup.

Pokud si chce uživatel nechat automaticky vygenerovat soubor s atomicitami musí v adresáři *atomicity*, který se nachází na CD příloze použít skript *findAtomicity*. XML soubor s atomicitami se vytvoří pro všechny .class soubory (které musí být již instrumentované) obsažené v tomtéž adresáři. Takto vytvořený vstup je uložen do podadresáře */atomicity* jako *atomicity.xml*. Tento druh vytváření souboru s atomickými sekcemi přitom využívá k torbě část programu RaceDetector. Druhá možnost je, že si uživatel atomicity sám napíše podle patřičného dtd souboru, který byl výše zmiňován.

V případě, že má uživatel vše z výše zmiňovaného seznamu připraveno, je řada na nastavení patřičných cest ve zdrojových souborech programu pro ověřování korektního léčení. Konkrétně musí uživatel otevřít zdrojový soubor *CGOCVariables.java* a nastavit si tyto čtyři proměnné:

- ATOMICITY_INPUT_FILE_LOCATION – musí být nastavena tak, aby odkazovala na místo v souborovém systému, kde je uložen vstupní soubor.
- ATOMICITY_INPUT_FILE – musí být nastavena, tak aby odrážela název vstupního XML souboru.
- ATOMICITY_OUTPUT_FILE_LOCATION – musí být nastavena, tak aby odkazovala na místo v souborovém systému, kde si přeje uživatel uložit výstup programu v podobě XML souboru
- ATOMICITY_OUTPUT_FILE – uživatel si touto „proměnnou“ nastaví jméno výstupního souboru do kterého bude výstup zapsán

Po tomto nastavení si uživatel musí z balíčku *Healing_assurance* (balíček od ověřování korektního léčení) vyexportovat .jar soubor, který musí nahrát do adresáře */plugin*, jež se nalézá v kořenové složce nástroje FindBugs.

Nakonec už stačí program pouze spustit. Jedna z možností jak to udělat pod operačním systémem Linux je tato: Uživatel vstoupí do adresáře */bin* v nástroji FindBugs a zadá příkaz: „./findbugs -textui -visitors HealingAssuranceBySearchingMonitors /home/pAblO/-findbugs/testClasses.jar“.

Tímto příkazem se provede analýza instrumentovaných tříd obsažených v *testClasses.jar* přičemž umístění a název balíku musí odpovídat uživatelově vstupu. Jednotlivé argumenty při spouštění nástroje FindBugs mají následující význam: „-textui“ – spouští konzolové rozhraní, „-visitors“ – určuje, že nad vstupní třídou/třídami se provede analýza jen určitým detektorem, „HealingAssuranceBySearchingMonitors“ – určuje detektor pro ověřování korektnosti léčení, který byl předmětem této práce, „/home/pAblO/findbugs/testClasses.jar“ – určuje cestu společně s názvem balíku instrumentovaných tříd. Instrumentované třídy přitom mohou být programu předány buď jednotlivě výčtem nebo za pomocí podobného archivu.

V případě, že bude uživatel postupovat dle popsaného návodu, neměl by narazit na problémy se zprovozněním vytvořeného nástroje. Podrobný postup jak při spouštění postupovat lze také vidět na testu v kapitole 5.