VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

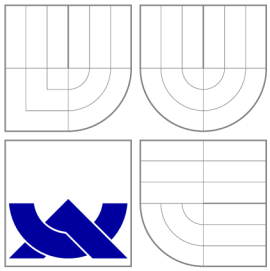
BOUNDED MODEL CHECKING V NÁSTROJI JAVA PATHFINDER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

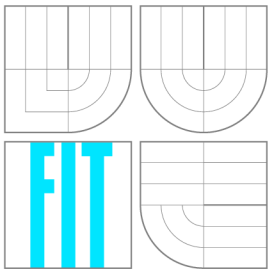
AUTOR PRÁCE
AUTHOR

Bc. VENDULA HRUBÁ

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

BOUNDED MODEL CHECKING V NÁSTROJI JAVA PATHFINDER

BOUNDED MODEL CHECKING USING JAVA PATHFINDER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VENDULA HRUBÁ
AUTHOR

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

BRNO 2008

Abstrakt

Diplomová práce je věnovaná aplikaci formální metody bounded model checking pro automatickou opravu chyb. Oprava se specializuje na chyby spojené se souběžností. Práce je zaměřena na programy napsané v jazyce Java, a proto pro verifikační metodu byl zvolen model checker Java Pathfinder, který je určen pro Java programy. Vlastní verifikační metoda spočívá v aplikaci strategie pro navigaci stavovým prostorem do místa verifikace. Z daného místa je spuštěn bounded model checking pro ověření opravy. Navigace stavovým prostorem je implementována pomocí strategie record&replay trace. Pro aplikaci bounded model checkingu jsou implementovány další parametry a moduly pro verifikaci speciálních vlastností systému, které ověřují koreknost opravy chyby. Bounded model checking se provádí v okolí opravy.

Klíčová slova

Model Checking, Java PathFinder, Bounded model checking, verifikace, Record&Replay trace, automatická oprava, souběžnost, ověřování opravy

Abstract

This thesis deals with the application of bounded model checking method for self-healing assurance of concurrency related problems. The self-healing is currently interested in the Java programming language. Therefore, it concetrate mainly on the model checker Java PathFinder which is built for handling Java programs. The verification method is implemented like the Record&Replay trace strategy for navigation through a state space and performance bounded model checking from reached state through the use of Record&Replay trace strategy. Java PathFinder was extended by new moduls and interfaces in order to perform the bounded model checking for self-healing assurance. Bounded model checking is applied at the neighbourhood of self-healing.

Keywords

Model Checking, Java PathFinder, Bounded model checking, verification, Record&Replay trace, self-healing, concurrency, healing assurance

Citace

Vendula Hrubá: Bounded model checking v nástroji Java PathFinder, diplomová práce, Brno, FIT VUT v Brně, 2008

Bounded model checking v nástroji Java PathFinder

Prohlášení

Prohlašuji, že jsem tuto diplovomou práci vypracovala samostatně pod vedením pana Ing. Bohuslava Křeny, Ph.D.

.....
Vendula Hrubá
19. května 2008

Poděkování

Chtěla bych poděkovat celé své rodině, prátelům a spolužákům za jejich podporu a pomoc při studiu.

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.

Acknowledgment

This work is supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD – project SHADOWS contract IST-035157. The authors are solely responsible for the content of this thesis. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

Obsah

Kapitola 1

Úvod

V dnešní době se setkáváme s počítači nebo počítačovými programy téměř na každém kroku. Ať už se jedná o mobilní telefony, bankovní účty, průmyslové stroje nebo řídící střediska. Vždy chceme, aby nám naše přístroje pracovaly správně, což ale již z principu nelze. O žádném rozsáhlejším programu nemůžeme prohlásit, že neobsahuje chyby, lze pouze říci, jaké chyby neobsahuje. Jsou případy, kdy nám nevadí, že program „zatuhne“ nebo provede neplatnou akci, např. při posílání emailu se nepodaří komunikace se serverem, email se neodešle a jeho obsah je ztracen. V takovém případě nás chyba zamrzí, musíme email napsat znova. Může ovšem nastat situace, kdy se jedná o důležité události, které mohou způsobit závažnější problémy – nepřičtení správné částky na bankovní účet, porucha na lékařských přístrojích, atd. V takovém případě rozhodně nechceme, aby program zatuhl nebo vykonal neočekávanou akci. Z toho důvodu se vynakládá spousta peněz na testování a ověřování korektnosti programů, aby programy vykonávaly pouze požadované události.

I přes veškeré snahy v programech zůstávají chyby, které se dostanou až k uživateli. Dává tedy smysl se zabývat opravou těchto chyb, které se neodhalí při testování a snažit se je opravit za běhu aplikace. Právě o to se snaží projekt SHADOWS, jehož součástí je i tato diplomová práce. SHADOWS – A **S**elf-**h**ealing **A**pproach to **D**esigning C**o**mplex Soft**w**are **S**ystems je evropský projekt, který se zabývá procesem automatické opravy (self-healing approach). Princip procesu a jeho cíl se dají specifikovat pomocí následujících čtyř základních kroků:

1. **zjištění problému (problem detection)** – předtím než se začne s opravou (léčením) je potřeba určit, zda je v systému nějaký problém a čeho se týká,

2. **nalezení příčiny problému (problem localization)** – druhým krokem po určení možného problému v systému je třeba tento problém lokalizovat resp. určit místo v programu (sledovaném systému), kde se problém vyskytuje,

3. **oprava problému (problem healing)** – dalším krokem je výběr akce, ze seznamu nabízených léčících akcí, které je možné pro opravu detekovaného problému použít,

4. **ověření opravy (problem assurance)** – posledním krokem je ověření, zda provedení opravné akce nezpůsobilo v systému jiný problém. Léčící akce změní chování systému, předpokládáme, že se nyní systém bude chovat korektně vzhledem k určeným požadavkům. Nicméně opravná akce může změnit chování systému takovým způsobem, že se objeví jiný problém. Z toho důvodu je potřeba provést ověření opravy.

Projekt SHADOWS se zabývá splněním funkčních požadavků, výkonem a v neposlední řadě správným použitím souběžného provádění v programu. Fakulta informačních technologií se v projektu věnuje části zaměřené na souběžné provádění (concurrency). Přesněji se věnuje léčení chyb, které vznikají pomocí paralelismu resp. souběžnosti v systému. Jedná se o chyby typu deadlocks, data races, lost notification, atd. Léčení se zaměřuje na opravy chyb v Java programech, ve kterých se souběžnost implementuje velice snadno pomocí vláken.

Tato diplomová práce je věnovaná poslednímu z uvedených kroků opravného procesu – ověření léčící akce. K tomu, aby bylo možné navrhnout vhodnou metodu pro ověření léčící akce, je zapotřebí znát, co léčící akce provádí a jakým způsobem funguje. Pokud známe princip opravné akce, je možné navrhnout a posléze implementovat metody pro ověření korektnosti opravy.

Další text je dělen do následujících kapitol. V druhé kapitole jsou podrobněji rozepsány jednotlivé kroky opravného procesu pro lepší porozumění celého procesu opravy. Dále jsou zde stručně popsány formální metody, ze kterých byla pro ověření opravy zvolena metoda model checking. O model checkingu, jeho základech a principu pojednává třetí kapitola. Zde jsou také popsány jeho modifikace, problém stavové exploze a jeho možná řešení. Ve čtvrté kapitole jsou nejprve uvedeny různé model checkery, ze kterých byl pro verifikaci zvolen model checker – Java PathFinder. Jeho popis a nastavení jeho vlastností jsou v této kapitole také uvedeny. Další kapitola popisuje hlouběji zvolenou metodu pro verifikaci systému z hlediska implementace v Java PathFinderu. Jsou zde uvedeny příklady a dosažené výsledky. Poslední kapitolou je závěr, který obsahuje shrnutí výsledků a nástin další práce na projektu.

Kapitola 2

Rozbor problematiky

2.1 Proces automatické opravy

V následujících odstavcích budou podrobněji rozepsány jednotlivé kroky procesu automatické opravy. Kroky procesu popisují princip léčení chyb, které vznikají souběžností v Java programech [12].

Zjištění problému. V prvním kroku se monitoruje vykonávání programu. Monitorování probíhá pomocí instrumentace programu nad byte-codem. Pokud by se chyba hledala na úrovni byte-codu pomocí ladění (debugging), je velká pravděpodobnost, že chyba nebude nalezena. Program na úrovni byte-codu umožňuje velké množství možností prokládání instrukcí a tím se snižuje pravděpodobnost nalezení chyby během testování. Pomocí instrumentace se do programu zavádí také šum (další instrukce), pomocí kterých se zvyšuje pravděpodobnost odhalení chyb vzniklých nesprávným prokládáním instrukcí (data race). Instrumentace kódu se provádí pomocí nástroje ConTest, který pracuje nad Java byte-codem a jehož bližší popis a specifikace jsou uvedeny v [5]. ConTest je nástroj vyvíjený výzkumnou laboratoří IBM pro testování Java programů, které obsahují více vláken [6].

Nalezení příčiny problému. Nalezení příčiny problému je obtížný úkol i pro člověka, tím těžším úkolem se stává navržení mechanizmu pro automatickou detekci příčiny chyby. V projektu SHADOWS byly navrženy metody pro automatickou detekci. První metodou je správný odhad a výběr opravné akce ze speciálního seznamu, který byl navržen pro výběr správné opravy chyb vzniklých souběžností. Jedná se o následující detektory: dataRace detektor, atomRace detektor nebo deadlock detektor. Jinou možností je provádění velkého množství testů s různými body instrumentace a následné statistické vyhodnocení dosažených výsledků. Pomocí získaných statistických dat a korektnosti chování programu v daném testu se dá určit problém. Oba výše popsané přístupy lze kombinovat s formálními metodami (např. model checkingem, statickou analýzou) a pomocí nich snížit počet false alarmů. False alarmy vznikají v případě, pokud je nadetekován problém, který ve skutečnosti v programu není. Z principu funkce formálních metod, by se tyto metody daly aplikovat na samotnou lokalizaci problému v systému. Nicméně jejich aplikace na reálný systém je problematická z důvodu stavové exploze. Ta je kritickým problémem formálních metod, který brání jejich nasazení na reálné systémy a provedení jejich celkové verifikaci.

Oprava problému. Nástroj (tool) pro opravu problému může být založen na výběru opravné akce, které jsou vyjmenovány ve speciálním seznamu. Zvolená oprava detekovaného problému může být nabídnuta vývojáři jako možné řešení během vývoje systému. Vývojář se pak sám rozhodne, zda nabízenou akci provede či nikoliv. Cílem projektu je ovšem

léčit chyby, které se projeví u uživatele. Záměrem je tedy léčit chyby, které se již jednou v systému vyskytly a na ně aplikovat opravu, a tím zamezit jejich opětovnému výskytu v aplikaci u uživatele. Prozatím se léčící akce zaměřují na problémy typu data races. Data races vznikají v důsledku paralelního přístupu ke sdílené proměnné ve stejný časový okamžik z více míst, jehož důsledkem je konfliktní zápis do sdílené proměnné. Možným řešením odstranění problému data race je přidání zámků (locks). Přesnější popis jednotlivých oprav i samotného problému data race je blíže uveden v [1, 12].

Ověření opravy. Posledním krokem procesu je ověření opravné akce. Bylo by chybné předpokládat, že lze navrhnout univerzální opravu určitého problému, která by fungovala za všech okolností. Z toho důvodu jsou navrženy speciální léčící akce pro konkrétní případy výskytu určitého problému. Proto je také potřeba ověřit, zda zvolená oprava chyby byla úspěšná a bezpečná (např. ověření zda přidání zámků pro odstranění problému data race, nezpůsobilo jiný problém – deadlock, apod.). Ke kontrole opravy je vhodné použít formální verifikaci, jedná se efektivní metody pro ověřování správnosti systému. Mezi formální metody patří model checking, statická analýza nebo theorem proving. V případě jejich omezení resp. modifikace je lze aplikovat na reálný systém a zverifikovat ho.

Jak již bylo uvedeno v úvodu, k tomu aby bylo možné navrhnout správnou metodu pro ověření opravy, je potřeba znát její podstatu a způsob aplikace na problém resp. chybu.

2.1.1 Princip opravy problému

Předpokládejme, že v monitorovaném systému byl lokalizován problém. Ten se nachází ve speciálním seznamu problémů, pro které jsou navrženy léčící akce. Posléze se zvolená léčící akce aplikuje na detekovaný problém, a ta problém v programu opraví [1]. Existují dvě možnosti jak opravu na problém aplikovat. První možností je pouze navržení opravy vývojáři, který se sám rozhodne, zda ji na opravu použije či nikoliv. Druhou vhodnější metodou je použití arichitektury listenerů. Listenery umožnují modifikovat chování systému za jeho běhu bez nutnosti explicitního zásahu do zdrojového kódu. Architektura listenerů je součástí již zmiňovaného nástroje ConTest [5, 16].

Oprava pomocí plánování. Díky monitorování (sledování) systému za jeho běhu vzniká v běhu programu šum, resp. v programu se vykonává více instrukcí než v běžném běhu programu bez monitorování. Pomocí tohoto mechanizmu je snadnější odhalit chyby vzniklé souběžností, které se projevují pouze při určitém prokládání instrukcí. K nesprávnému prokládání instrukcí vedoucí k chybě dochází například při vytížení procesoru jinými programy, které nejsou běžně při testování programu spuštěny. Proto monitorování, které také potřebuje porvádět vlastní instrukce umožnuje odhalení těchto chyb. Pro opravu podobných chyb v Java programech je někdy postačující zavolání metody `yield()`. Metoda se zavolá před kritickou seckí, kde dochází k chybě. Princip metody `yield()` spočívá v přepnutí kontextu na jiné vlákno (aktuální vlákno se vzdává procesoru), při dalším přidělení procesoru je již dostatek času na vykonání kritické sekce vcelku (bez přepnutí kontextu). Uvedenou techniku lze kombinovat s přidělováním různých priorit vláknům. V takovém případě do kritické sekce může vstoupit nebo ji přerušit pouze vlákno se stejnou nebo vyšší prioritou.

Oprava pomocí synchronizace. Zmiňovaná část projektu SHADOWS je zaměřena na chyby, které jsou způsobeny nevhodným použitím synchronizace, nebo když synchronizace zcela chybí. V případě chybného použití synchronizace obsahuje oprava vhodnou změnu v synchronizaci. Pokud se jedná o problém chybějící synchronizaci nad sdílenou proměnnou, je třeba věnovat dostatečnou pozornost přidání zámku. Zámek je potřeba vložit takovým

způsobem, aby jeho přidání nezpůsobilo jiný problém v systému, typickým problémem je vznik deadlocku. Je tedy nutno ověřit, zda přidání zámků skutečně vyřešilo problém (data race) a dále zverifikovat, že zmíněná oprava nezpůsobila v systému jiný problém.

2.1.2 Princip ověření opravy

Pro ověření opravy je možné použít techniky formální verifikace jako model checking, statickou analýzu atd. K tomu, aby bylo možné formální metody aplikovat na reálný systém, je třeba udělat jejich omezení nebo určitou modifikaci. V tomto případě – ověřování opravy, není cílem zverifikovat celý systém, ale pouze okolí léčené chyby. Jde tedy o aplikaci formální metody na konkrétní část programu a na kompletní reálný systém. Aplikace formální metody může vypadat následovně.

Použití statické analýzy nad byte-codem: v případě použití opravné akce – přidání zámků (lock/unlock) je třeba ověřit, zda nové zámky nejsou v kolizi s již naimplementovanou synchronizací. Případná kolize synchronizace by mohla způsobit deadlock. Problém data races se často vyskytuje nad jednoduchými příkazy (malá část kódu). Provést statickou analýzu nad malou částí kódu není problém a tudíž odhalení daného problému je možné.

Pro použití model checkingu se je nejprve potřeba dostat do požadovaného stavu resp. do jeho okolí a zde posléze provést bounded model checking. Bounded model checking se provede do předem určené hloubky stavového prostoru. K prohledávání stavového prostoru existují různé heurisitiky, dále je potřeba pro prohledávání stavového prostoru zadat různé specifikace, které mají být v systému splněny (verifikovány).

Pokud opravná akce způsobí jiný problém a pomocí ověřování opravy se daný problém zjistí, je potřeba upravit léčící akci tak, aby k nové chybě nedocházelo nebo zvolit jinou léčící akci. Po změně opravy je třeba provést nové ověření běhu programu, zda se již systém chová korektně. Stále zde ale přetrvává problém možného výskytu chyb. Verifikace systému nebo části systému se provádí vždy vzhledem k nějakým konkrétním vlastnostem a specifikacím. Stále tedy neřeší problém, kdy o žádném systému nelze říci, že je bezchybný. O programu lze pouze říci, které chyby neobsahuje. Pokud je znám princip opravy, je možné navrhnout a posléze implementovat takovou metodu pro ověření korektnosti opravy, která ověří ty specifikace, u kterých během léčení došlo ke změně.

2.2 Formální metody

Formální metody jsou založeny na matematických metodách, umí dokázat/vyvrátit určité specifikace systému. Formálními metodami rozumíme verifikační nebo analytické metody.

Verifikace ověřuje zadané vlastnosti v systému. Například v paralelních programech může verifikace odpovědět na otázku, zda se v programu vyskytuje deadlock. Verifikace tedy vrací odpověď ano/ne podle toho, jestli je verifikovaná vlastnost v systému splněna nebo ne. Existují ale také specifikace, které ověřit nelze.

Analýza poskytuje odpovědi na obecnější otázky o systému, neodpovídá tedy ano/ne, zda je vlastnost v systému splněna, ale podává komplexnější informace o chování systému jako optimalizace, syntéza, apod.

Rozdíl mezi formálními metodami a testováním spočívá v odpovědi na otázku, zda systém obsahuje danou chybu. Pokud se během testování objeví chyba, je zřejmé, že program danou chybu obsahuje, ale pokud se během testování chyba neprojeví, neznamená to, že v programu zadaná chyba není. Narozdíl od formální verifikace, jestliže se během

verifikace chyba neprojeví a byla provedena verifikace celého systému, uvedená chyba se v systému opravdu nevyskytuje.

Ideální formální verifikace splňuje následující vlastnosti. Ideální formální verifikace je plně automatická, spolehlivá (pokud dojde k závěru, musí být správný), úplná (pokud najde chybu, jedná se o skutečnou chybu – nejede o false alarm), konečná (verifikace vždy skončí s určitým závěrem). Bohužel v praxi ideální verifikace neexistuje, kritickým problémem verifikace je stavová exploze. V případě systému s konečným stavovým prostorem a možností využití dostatečně výkonného počítače – verifikace vždy dojde k závěru. V praxi se ale často setkáváme s neomezeným stavovým prostorem nebo nadměrnými požadavky na výkon počítače (důvodem je práce s daty – jeden třicetidvou-bitový integer může nabývat 2^{32} hodnot, paralelismus, nedeterminismus, ...).

Reálné verifikace mají tedy následující vlastnosti: produkují false alarmy, nekončí vždy korektně, nejsou plně automatické a nejsou stoprocetně spolehlivé.

K tomu, aby bylo možné provádět verifikace, je nejprve zapotřebí nadefinovat systém, který bude verifikován a jeho vlastnosti, které se mají zkontrolovat. Vstupní systém může být popsán pomocí modelu (petri net, promela, SMV, atd.) nebo jako reálný systém (programy v různých programovacích jazycích, Verilog, hardware pospaný ve VHDL, atd.). Vlastnosti, které se mají verifikovat, mohou být např. živost proměnných, invarianty, ukončení systému pouze v požadovaném místě. Vlastnosti, které se verifikují, lze rozdělit do dvou skupin: bezpečnost a živost. *Bezpečnost* nám říká, že v systému nikdy nenastane nic špatného, protipříklad takové vlastnosti je konečný – někdy nastane něco špatného. Oproti tomu *živost* systému říká, že v systému někdy nastane něco dobrého a protipříklad je nekonečný – nikdy nenastane nic špatného. Lze tedy uvést, že bezpečnost se verifikuje snadněni než živost. Pro popis vlastností je možné použít například formule temporálních logik,μ-calcul, labels, atd.

Další důležitou charakteristikou vlastností pro verifikaci je čas, který má dvě základní rozdělení na čas logický/fyzický a lineární/větvící se. *Logický čas* ukazuje časový sled jednotlivých událostí podle běhu, kdežto *fyzický čas* slouží k určení kolik času na jakou událost bylo potřeba vzhledem ke zvolenému měřítku. *V lineárním čase* je pouze jedna možnost času, je možný pouze lineární běh programu. Oproti tomu *větvící se čas* umožňuje společný náhled na dva běhy, které se rozcházejí až posléze.

Nejpoužívanějšími metodami pro formální analýzu a verifikaci jsou *model checking*, *statická analýza* a *theorem proving*.

Model checking. Model checking je algoritmická technika, která ověřuje, zda systém splňuje požadované vlastnosti [4]. Verifikovaný systém může být popsán jak modelem, tak reálným systémem. Model checking umožňuje verifikovat systémy s konečným stavovým prostorem. Kritickým problémem model checkingu je exploze stavového prostoru, kde velikost stavového prostoru roste exponenciálně s velikostí verifikovaného systému [21]. Vlastnosti jsou typicky specifikovány pomocí temporálních logik.

Výhodami model checkingu jsou: vysoký stupeň automatizace, snadné použití (pokud to umožňuje systém), poměrně všeobecné použití pro verifikaci různých vlastností systému. Oproti tomu hlavní nevýhodou je již zmiňovaná exploze stavového prostoru, který se dá řešit různými metodami jako redukce, abstrakce, kompozice systému z částí, atd. Další nevýhodou je nutnost modelovat okolí verifikovaného systému jako jsou vstupy.

Statická analýza. Statická analýza se neprovádí nad skutečným během systému, ale pouze nad zdrojovým kódem. Tím umožňuje zjistit informace o programu ze zdrojového kódu, aniž by bylo nutné program spouštět. Statické analýzy jsou různé druhy (typová

analýza, vyhledávání chybových vzorů – bug pattern, dataflow analýza, apod.). Výhodami statické analýzy je možnost verifikovat obrovské systémy, zároveň není nutné modelovat okolí systému a nabízí velký stupeň automatizace. Nevýhodou je velká specializace jednotlivých statických analýz na konkrétní problémy. Pokud je cílem zverifikovat nově zadaný problém, musí se nejprve navrhnout nová statická analýza pro jeho ověření. Další nevýhodou je vznik false alarmů. Díky tomu, že se statická analýza provádí nad zdrojovým kódem, může nadetekovat chybu, která při skutečném běhu nikdy nenastane (false positive).

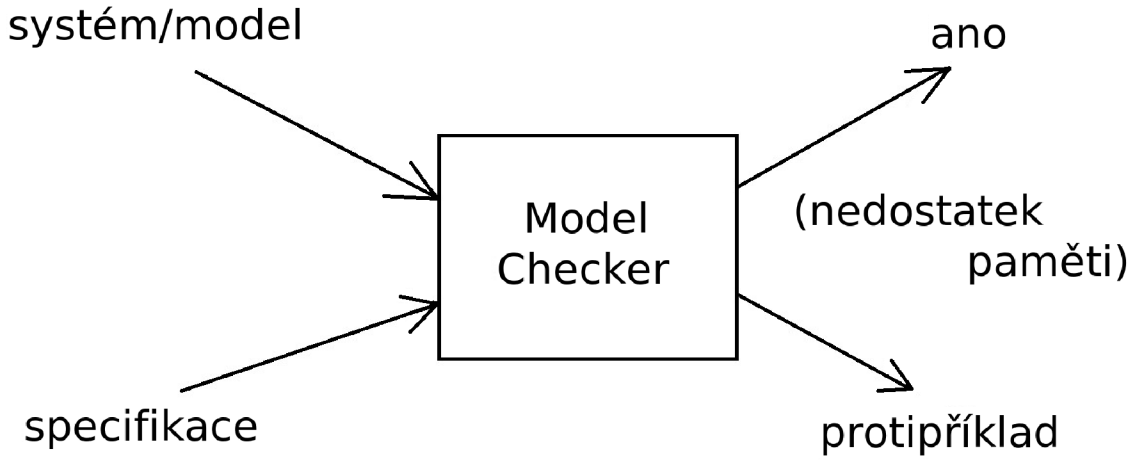
Theorem proving. Jedná se o deduktivní verifikaci, která vychází z axiomů a pomocí použití různých pravidel produkuje vlastnosti systému. Užití jednotlivých pravidel není automatické, je potřeba zásahu uživatele „experta“, který vede důkaz. Jedná se tedy o částečně automatickou metodu, která je velmi obecná, což je její výhodou. Nevýhodou je problém s diagnostickou informací, která může, ale také nemusí být k dispozici.

Pro ověření opravy byla zvolena fromální metoda – model checking. Ta umožňuje verifikovat systém dynamicky, a tedy hledat pouze chyby, které mohou v systému reálně nastat oproti statické analýze, která produkuje hodně false alarmů. Další výhodou je široký výběr z exitujících nástrojů (model checkerů) pro verifikaci pomocí model checkingu.

Kapitola 3

Model Checking

Model checking je algoritmická cesta k ověření, zda zadaný systém splňuje požadovanou vlastnost [4]. Základní schéma verifikačního procesu je znázorněné na obr. 3.1. Vstupem pro verifikace je verifikovaný systém, který je typicky popsán pomocí formálního modelu systému. Druhým vstupem je formální specifikace vlastností, které se mají ověřit. Model systému a specifikované vlastnosti vstupují do model checkeru, který ověřuje resp. verifikuje zadané vlastnosti. V ideálním případě nám model checker vrátí odpověď ano/ne, podle toho jestli je vlastnost splněna. V praxi ovšem může nastat i situace, kdy nám model checker není schopen poskytnout odpověď, např. z důvodu nedostatku paměti.



Obrázek 3.1: Základní schéma procesu verifikace pomocí model checkeru

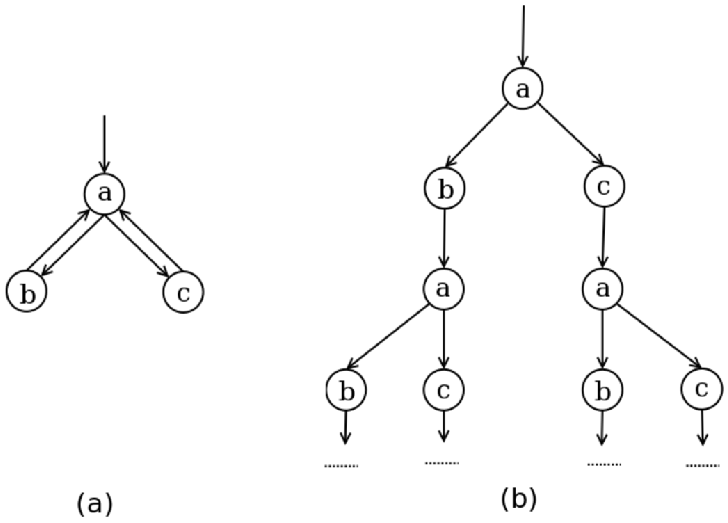
Principem model checkingu je systematické prohledávání stavového prostoru. *Stav (state)* je snímek systému v čase, zachycuje hodnoty proměnných v daném časovém okamžiku. Zároveň chceme znát, jak se stavy mění – popis změny mezi jednotlivými stavy (stav před událostí a stav po události) se nazývá *přechod (transition)*. *Výpočet (computation)* je sekvence stavů, kde se následující stav získá z předchozího stavu pomocí přechodu.

3.1 Model systému

Reálné systémy jsou většinou zadány pomocí textu programu (zdrojový kód) nebo diagramem cest. Je tedy potřeba mechanizmus, který je schopen popsat všechny typy systémů tak, aby mohli být verifikovány. Jednou možností popisu chování systému je *Kripkeho struktura (Kripke structure)* obr. 3.2(a) . Jedná se o graf, který popisuje stavy systému a jeho přechody:
Nechť AP je množina atomických výroků, pak Kripkeho struktura M nad AP je čtveřice $M = (S, S_0, R, L)$, kde

- S je množina stavů,
- $S_0 \subseteq S$ je množina počátečních stavů,
- $R \subseteq S \times S$ je přechodová relace mezi stavy taková, že $\forall s \in S : \exists s' \in S : R(s, s')$
- $L : S \to 2^{AP}$ je funkce přiřazující každému stavu množinu všech atomických tvrzení, která v daném stavu platí, $AP = \{v = d \mid v \in V \wedge d \in D\}$, kde V je množina proměnných nad doménou D.

Pro popis sekvenčního chování Kripkeho struktury M jsou definovány cesty (paths). *Cesta (path)* v Kripkeho struktuře M ze stavu s je nekonečná sekvence stavů $\pi = s_0 s_1 s_2 ...$, která je řazená s ohledem na přechodovou relaci v M. Přechodová relace $R(s_i, s_{i+1})$ platí pro všechny $0 \leq i < |\pi| - 1$. Jestliže $I(s_0)$ t.j. s_0 je počáteční stav, potom se cesta nazývá počáteční (initialized). Délka cesty (length) $|\pi|$ může být jak konečná, tak nekonečná. Pro $i < |\pi|$ je definovaný i-tý stav s_i v sekvenci jako $\pi(i)$. Suffixem π je $\pi_i = (s_i, s_{i+1}, \dots)$, který začíná stavem s_i. Všechny možné počáteční cesty systému, lze z Kripkeho struktury zapsat pomocí *výpočetního stromu (computation tree)* obr. 3.2(b). Výpočetní strom tedy vznikne rozepsáním Kripkeho struktury z počátečního stavu.



Obrázek 3.2: Ukázka Kripkeho struktury (a) a výpočetního stromu (b)

Granularita přechodů. Jedním ze zásadních nastavení verifikace je určení granularity přechodů systému. Tento fakt je podstatný pro určení atomických přechodů, tyto

přechody jsou dále nedělitelný. Pokud přechod obsahuje více operací, provádějí se jako jedna jediná atomická operace. Běžnou chybou je nastavení hrubé granularity, kdy se za atomické přechody považují i takové přechody, které v reálném systému atomické nejsou. Tím může dojít k chybě při verifikaci, model checking neodhalí chyby (errors), které v systému jsou. Oproti tomu nastavení velmi jemné granularity může způsobit detekování chyb, které v reálném běhu systému nemohou nikdy nastat.
Příklad z [4]: Systém má 2 proměnné x, y a dva souběžně proveditelné přechody α, β.

$\alpha : x := x + y$

$\beta : y := y + x$

s počátečním stavem $x = 1, y = 2$. Oba přechody mají jemnou granularitu, implementace přechodů je pomocí instrukcí assembleru: *load*, *add*, *store*, které slouží pro práci s pamětí a registry:

$\alpha_0 :$ *load* R_1, x $\quad$ $\beta_0 :$ *load* R_2, y

$\alpha_0 :$ *add* R_1, y $\quad$ $\beta_0 :$ *add* R_2, x

$\alpha_0 :$ *store* R_1, x $\quad$ $\beta_0 :$ *store* R_2, y

Výpočet přechodu α a pak β dává výsledek $x = 3 \wedge y = 5$. Pokud je přechod β vykonán před α obdržíme výsledek $x = 4 \wedge y = 3$. Na druhou stranu podle granularity implementace může nastat i situace $\alpha_0\beta_0\alpha_1\beta_1\alpha_2\beta_2$ a výsledek $x = 3 \wedge y = 3$.

Předpokládejme, že $x = 3 \ \wedge \ y = 3$ porušuje vlastnosti systému, dále předpokládejme, že systém je implementován pro přechod α a β. V tom případě je nemožné získat výsledek $x = 3 \wedge y = 3$. Potom s granularitou $\alpha_0\alpha_1\alpha_2\beta_0\beta_1\beta_2$ můžeme dostat stavy, které v systému nikdy nenastanou. Na druhou stranu máme implementaci $\alpha_0\alpha_1\alpha_2\beta_0\beta_1\beta_2$, ale systém modelujeme s α a β, potom nikdy nenalezneme chybu v systému.

Systémy se souběžností. Takové systémy se skládají z množiny komponent, které běží současně. Jednotlivé komponenty si mezi sebou předávají různá data – komunikují. Způsob komunikace je v jednotlivých systémech různý. Komunikace může probíhat *asynchronně* – v časovém okamžiku pracuje pouze jedna komponenta, nebo *synchronně* – všechny komponenty dělají krok ve stejném čase. Komunikace probíhá za pomoci *sdílených proměnných (shared variables)* nebo *zasílání zpráv (exchanging messages)*. Pokud program obsahuje sdílené proměnné resp. proměnné, ke kterým má přístup více než jeden proces, je třeba zajistit, aby ke sdílené proměnné měl v daný časový okamžik přístup pouze jeden proces. V opačném případě může dojít k nekonzistenci dat. Pro tyto účely slouží synchronizační příkazy, které jsou atomické a mohou mít podobu příznaku čekání (wait) nebo zámků (lock/unlock).

3.2 Specifikace systému

Po formálním nadefinování systému je třeba také formálně nadefinovat vlastnosti, které se mají verifikovat. Typickou formou specifikace vlastností pro model checking jsou následující temporální logiky:***L**inear-time **T**emporal **L**ogic (LTL)*, ***C**omputation **T**ree **L**ogic (CTL)* a *CTL**, která spojuje vyjadřovací možnosti LTL a CTL.

Temporální logiky jsou speciálním typem modálních logik, pomocí nichž se dají nadefinovat formule, které vyajdřují určitou vlastnost systému. V temporální logice není čas uveden explicitně, naopak, formule umožňují určit, zda je nějaký stav proveditelný nebo tento stav nikdy nenastane (např. chybový stav), jedná se o logický čas. Vlastnosti typu *někdy (sometimes)* nebo *nikdy (never)* jsou popsány speciálními temporálními operátory. Temporální operátory mohou být kombinovány libovolně s booleovskými spojkami nebo

vzájemně vnořeny. Jednotlivé temporální logiky se liší v operátorech, které poskytují a také v jejich sémantice.

Logika CTL*. Tato logika zahrnuje v sobě logiky CTL a LTL. CTL^* formule popisují vlastnosti výpočetních stromů. CTL^* formule obsahuje: atomické výroky (AP), booleovské spojky ($\wedge, \vee, \neg$), kvantifikátory cesty, temporální operátory.
Kvantifikátory cesty jsou použity pro popis větvení ve výpočetním stromu:

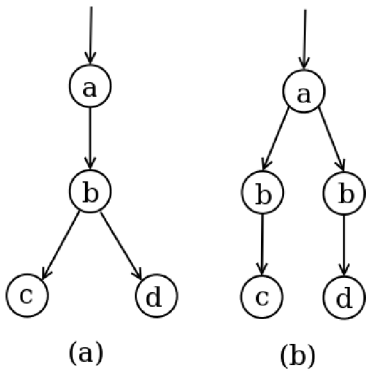
- **A** pro všechny cesty
- **E** existuje cesta

Temporální operátory popisují vlastnosti cesty (cesty přes strom). Existuje pět základních temporálních operátorů:

- **Xp** *(next time)* vlastnost p bude splněna v následujícím stavu dané cesty,
- **Fp** *(eventually)* vlastnost p bude splněna někdy na dané cestě,
- **Gp** *(globally)* vlastnost p je splněna ve všech stavech dané cesty,
- **pUq** *(until)* vlastnost p platí na cestě dokud neplatí q,
- **pRq** *(release)* vlastnost q byla splněna ve všech stavech cesty až po první stav včetně, ve kterém platí formule p.

V CTL^* logice existují dva typy formulí *stavové formule (state formulas)* a *formule cesty (path formulas)*.

Logika CTL a LTL. Jsou podlogiky CTL^*, rozdíl mezi nimi je v upořádání větvení ve výpočetním stromu. CTL (Computation Tree Logic) je logika s větvícím se časem a temporální operátory určují cesty, které jsou možné z daného stavu obr. 3.3(b). Oproti tomu LTL (Linear Temporal Logic) je logika s lineárním časem a operátory určují události, které mohou nastat po cestě ve výpočetním stromu obr. 3.3(a).



Obrázek 3.3: KS, VS, CTL, LTL

3.3 Výstup verifikace

Výsledkem verifikace je tedy odpověď na otázku model checkingu: Nechť, je dána Kripkeho struktura $M = (S, S_0, R, L)$, pomocí které je popsán verifikovaný systém a temporální formule f, reprezentující požadovanou vlastnost, kterou má systém splňovat. Odpovědí model checkingu je, zda zadaná formule f je splněna v Kripkeho struktuře M:

$$M \models \varphi \ \ ?,$$

respektive určit množinu stavů systému S, ve kterých je formule f splněna:

$$\{s \in S \mid M, s \models \varphi\}.$$

Postup určení, zda je formule f plněna spočívá v ověření, že:

- pro každou podformuli ψ formule φ je vypočtena množina stavů:

  $$S_\psi = \{s \in S \mid M, s \models \psi\},$$

- výpočet množiny stavů S_ψ začíná u nejvnitřenjších (dále nedělitelných) podformulí a postupuje se směrem ven k získání množiny stavů původní formule φ.

Typicky se provádí model checking z počátečního stavu (stavů) systému, díky tomu se ověří, zda je specifikace splněna v celém systému ($S_0 \subseteq S_\varphi$) nebo jeho části – formule je splněna v požadovaných stavech.

Existují různé algoritmy, pomocí nichž se určují množiny stavů, ve kterých je požadovaná formule splněna. Nicméně pro výpočet množiny u různých formulí, které obsahují jiné temporální operátory je třeba aplikovat různé algoritmy, jednotlivé algoritmy lze najít například v [4].

Algoritmy pro ověření formule jsou založeny na procházení stavovým prostorem a jejich hlavním problém se tedy stává exploze stavového prostoru.

3.4 Problém stavové exploze

Exploze stavového prostoru je kritickým problémem model checkingu (state space explosion problem), kde verifikace spočívá v generování stavového prostoru. Velikost stavového prostoru roste exponenciálně s velikostí verifikovaného modelu resp. systému [21]. Z toho důvodu je prakticky model checking bez redukce stavového prostoru v praxi nepoužitelný. Exploze stavového prostoru spočívá v nedeterminismu – ve velkém množství možností stavů, kterými může systém pokračovat z aktuálního stavu. Pro řešení problému stavové exploze existují různé přístupy [7, 15, 21, 23]. Jedním z možných dělení těchto přístupů je následující:

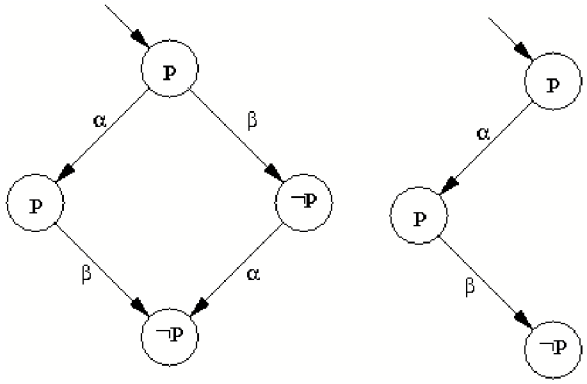
- **Předzpracování (preprocessing).** Ještě před začátkem provádění model checkingu se vykoná předzpracování verifikovaného systému. Do této skupiny patří: *slicing a jemu podobné techniky založené na statické analýze*, jejich podstatou je „oříznutí“ verifikovaného systému pouze na tu část, která může ovlivnit verifikovanou vlastnost. Vlastní model checking se provádí pouze nad „oříznutou“ částí systému. Další metodou je *nad aproximace*, pomocí ní je možné zjistit, co ovlivňuje zkoumanou oblast a dále pracovat se zvolenou částí systému. Dále je možné použít *explicitní určení atomické sekce*. Uživatel sám zadá oblasti, které mají být vykonány atomicky a nemá se rozgenerovávat jejich stavový prostor. V tom případě sám uživatel zaručuje, že jím zadaná atomická sekce nezpůsobí nenalezení chyby. Existují i metody, které dokáží určit atomické sekce automaticky. Tyto sekce jsou ovšem většinou velice jednoduché a

neefektivní vzhledem k redukci stavového prostoru. Další možností je *abstrakce (abstraction)*. Abstrakce se může vytvořit buď ručně (theorem proving, uživatel) nebo automaticky (predikátová abstrakce, petriho sítě).

- **Efektivní uložení Kripkeho struktury v paměti.** Další možností jak se vypořádat se stavovou explozí je komprimace stavového prostoru. *BDD (binary decision diagram)* je jednou z možností jak komprimovat stavový prostor, jednotlivé stavy jsou reprezentovány pomocí formulí. Další možností je *efektivní uložení stavu v paměti*, neukládají se celé nově generované stavy, ale pouze změny mezi již uloženým stavem a nově generovaným. *Abstrakce stavů* je metoda, která redukuje informace o procesech. Neuchovává se informace o přesném počtu procesů, ale pouze informace zda na daném řádku kódu je 0, 1 nebo ∞ procesů, jedná se o nadaproximaci.

- **Redukce stavového prostoru.** Redukce stavového prostoru neznamená pouze efektivní procházení stavového prostoru, ale také redukce počtu stavů. Redukce stavů může probíhat buď předem nebo za běhu programu – některé generované stavy nejsou z hlediska verifikované vlastnosti zajímavé. Metody pro redukci stavového prostoru jsou: *symmetry reduction*, *Petri-net unifolding* (rozvinutí petriho sítě do lineárního stromu).

  Zajímavé redukce pro programy se souběžností jsou „Partial order reduction“ (POR) a „on-the-fly model checking“. POR je založená na redukování generování stavového prostoru. V programech existují nezávislé přechody u kterých nezáleží na pořadí jejich vykonání, po jejich provedení se verifikace dostává do shodného stavu. Takové přechody (operace) se typicky vyskytují u více vláknových aplikací, kdy různá válkna provádí na sobě nezávislé operace a nezáleží tedy na pořadí jejich provádění. Ukázka části vygenerovaného stavového prostoru, kde je možné tuto generovanou část redukovat pouze na provedení jednoho průchodu je na obr. 3.4.



Obrázek 3.4: Ukázka redukce stavového prostoru pomocí POR

Metoda on-the-fly je založena na současném generování stavového prostoru a ověřování vlastnosti systému najednou. Při generování stavového prostoru se současně verifikuje, zda je vlastnost v daném generovaném stavu splněna. Pokud dojde k nalezení protipříkladu verifikované vlastnosti, generování stavového prostoru končí. Verifikace dospěla k závěru bez nutnosti rozgenerovávávat celý stavový prostor a tím k jeho redukci.

- **Kompoziční model checking (Compositional of MC).** Jedná se o rozdělení systému na komponenty, kde u každé komponenty sledujeme určité vlastnosti. Nakonec z těchto dílčích vlastností komponent odvodí vlastnosti celého systému.

Existují i další metody pro redukci stavového prostoru, které zde nejsou uvedeny. Také je možné jednotlivé metody vhodným způsobem kombinovat, a tím docílit vyšší efektivnosti při redukci stavového prostoru.

3.5 Bounded Model Checking

Bounded model checking resp. ohraničený model checking se používá pro verifikaci konečně stavových systémů. Bounded model checking může účinně redukovat problém splnitelnosti booleovské formule – SAT. Tento problém byl jedním z důvodů pro vytvoření této metody [3].

Základní myšlenkou bounded model checkingu je ověření určité vlastnosti systému pouze za pomoci konečného prefixu cesty v systému. Nalezení důkazu o splnění nebo vyvrácení určité vlastnosti systému se provádí v konečném počtu kroků. Prohledávání stavového systému je omezeno na předem danou délku k. Pomocí omezení prohledávání stavového systému se omezí i verifikace požadované vlastnosti. Verifikace je omezena na verifikaci prefixu cesty, která má délku k. V praxi se většinou omezená délka cesty prodlužuje na takovou dobu, dokud se nezíská důkaz o splnění/vyvrácení verifikované vlastnosti.

Přestože prefix cesty je konečný – obsahuje konečný počet stavů, může reprezentovat nekonečnou cestu. Pokud cesta obsahuje zpětnou smyčku (back loop) z posledního stavu prefixu do některého z předchozích stavů (obr. 3.5(b)) jedná se o reprezentaci nekonečné cesty pomocí konečného počtu stavů. V případě, že prefix cesty neobsahuje zpětnou smyčku (obr. 3.5(a)), nelze říci nic o nekonečném chování systému. Nelze zverifikovat vlastnosti, které požadují, aby určitá vlastnost byla platná nekončně dlouho. Pokud prefix neobsahuje zpětnou smyčku není známé chování systému za stavem s_k.

Obrázek 3.5: Dvě možnosti omezené cesty

Definice pro cestu, která obsahuje zpětnou smyčku je následující:

Pro $l \leq k$ nazýváme cestu π: (k,l)-*loop*, pokud $T(\pi(k), \pi(l))$ a $\pi = u.v^{\omega}$, kde $u = (\pi(0), \ldots, \pi(l-1))$ a $v = (\pi(l), \ldots, \pi(k))$. Pokud existuje $k \geq l \geq 0$, pro které je cesta π: (k,l)-*loop*, pak nazýváme cestu π: k-*loop*.

Pokud na cestě délky k existuje přechod z posledního stavu cesty s_k do některého z předešlých stavů cesty s_l, pak je možné danou část cesty neomezeně krát opakovat k-*loop*.

Sémantika model checkingu na limitovaných cestách se nazývá *bounded semantics*. Využívá se prvních $k+1$ stavů cesty $(s_0, \ldots, s_k)$, jedná se o konečný prefix cesty. Pomocí tohoto prefixu se určuje splnitelnost formule na cestě. Pokud cesta obsahuje smyčku k-*loop*, může se pro splnitelnost formule použít originální sémantika model checkingu. Tato sémantika lze využít vzhledem k faktu, že pokud cesta obsahuje zpětnou smyčku jsou nekonečné vlastnosti

cesty obsaženy v jejím konečném prefixu. Formálně lze toto tvrzení „Bounded Semantics for a Loop“ zapsat následovně:

Nechť $k \geq 0$ a π je *k-loop*. Potom LTL formule f je splněna na cestě π délky k (symbolický zápis $\pi \models_k f$ iff $\pi \models f$).

Druhá možnost nastává v případě, kdy cesta π neobsahuje *k-loop*. Potom formule $f = \mathbf{F}p$ je splněna na cestě π v originální (*neomezené*) sémantice, jestliže existuje index $i \geq 0$, takový, kde p je splněna na sufixu (π_i) cesty π. Pokud je použita *omezená* sémantika, potom k+1-ní stav $\pi(k)$ nemá následovníka a není tedy možné tuto sémantiku definovat rekurzivně pomocí sufixů cesty (π_i). Z toho důvodu se zavádí značení $\pi \models_k^i f$, kde i je aktuální pozice v prefixu na cestě π a suffix π_i cesty π splňuje formuli f: $\pi \models_k^i f$ implikuje $\pi_i \models f$. Formální definice tohoto tvrzení „Bounded Semantics without a Loop“ lze zapsat následovně:

Nechť $k \geq 0$, cesta π není *k-loop*. Potom LTL formule f je splněna na cestě π délky k resp. $\pi \models_k f$ iff $\pi \models_k^0 f$, kde

$\pi \models_k^i p$	iff	$p \in L(\pi(i))$
$\pi \models_k^i \neg p$	iff	$p \notin L(\pi(i))$
$\pi \models_k^i f \wedge g$	iff	$\pi \models_k^i f$ and $\pi \models_k^i g$
$\pi \models_k^i f \vee g$	iff	$\pi \models_k^i f$ or $\pi \models_k^i g$
$\pi \models_k^i \mathbf{G}f$	není splněna nikdy	
$\pi \models_k^i \mathbf{F}f$	iff	$\exists j, i \leq j \leq k.\ \pi \models_k^j f$
$\pi \models_k^i \mathbf{X}f$	iff	$i < k$ and $\pi \models_k^{i+1} f$
$\pi \models_k^i f\mathbf{U}g$	iff	$\exists j, i \leq j \leq k.\ \pi \models_k^j g$ and $\forall n, i \leq n < j.\ \pi \models_k^n f$
$\pi \models_k^i f\mathbf{R}g$	iff	$\exists j, i \leq j \leq k.\ \pi \models_k^j f$ and $\forall n, i \leq n < j.\ \pi \models_k^n g$

Na závěr této kapitoly o bounded model checkingu zle uvést následující dvě věty:

Nechť f je LTL formule a π je cesta, potom $\pi \models_k f \Rightarrow \pi \models f$.

Nechť f je LTL formule a M je Kripkeho struktura. Jestliže $M \models \mathbf{E}f$, potom existuje $k \geq 0$, kde $M \models_k \mathbf{E}f$.

Z těchto dvou vět lze odvodit následující theorém o bounded model checkingu:

Nechť f je LTL formule a M je Kripkeho struktura, potom $M \models \mathbf{E}f$ iff existuje $k \geq 0$, resp. $M \models_k \mathbf{E}f$.

Theorém říká, pokud zle získat takovou délku k, na které je formule splněna, potom omezená a neomezená sémantika jsou ekvivalentní.

3.6 Navigace stavovým prostorem

V projektu SHADOWS se bounded model checking využívá pro zverifikování okolí léčené chyby, není tedy cílem provést bounded model checking z počátečního stavu (s_0), ale z nějakého konkrétního stavu ve stavovém prostoru systému. Aby bylo možné z toho stavu provést bounded model checking, je potřeba nejprve daného stavu dosáhnout, projít stavový prostor do konkrétního stavu. K tomuto účelu existují různé metody [14, 20], které slouží k navigaci stavovým prostorem do požadovaného stavu. Tyto metody mohou být následující.

- **Record&Replay trace.** Tato strategie pro navigaci stavovým prostorem je založena na zaznamenání běhu programu (trace) a posléze přehrání této cesty ve zvoleném

model checkeru. Pomocí záznamu běhu programu se je možné navádět stavovým prostorem až do místa opravy chyby, ze kterého je možné spustit bounded model checking. Výhodou této metody je zaznamenání celé cesty provádění programu. Díky tomu je možné provést bounded model checking ne pouze z chybového stavu programu, ale z kteréhokoliv stavu, který mu na cestě předchází. Zásadní nevýhodou této metody je nutnost ukládání celého běhu programu, a tím zpomalení chodu aplikace. Další nevýhodou je paměťová náročnost, pro záznam cesty programu je zapotřebí velké množství dat. Čím delší dobu program běží, tím je potřeba více paměti pro záznam cesty. Minimálně stejně dlouhá doba jako pro záznam cesty je také potřeba pro přehrání zaznamenané cesty. Proto se tato metoda nehodí pro navigaci stavovým prostorem u systémů, které jsou dlouhodobě v chodu.

Aby bylo možné přehrát zaznamenanou cestu běhu programu, je třeba ukládat relevantní informace jako informace o aktuálním vlákně, vykonané byte-code instrukci atd.

- **Store&Restore state.** Další strategie pro získání požadovaného stavu ze stavového prostoru systému je založena na uložení a opětovném obnovení stavu. Neprve se uloží aktuální stav běžícího programu a následně se uložený stav obnoví ve zvoleném model checkeru. Po obnovení uloženého stavu je z něj možné provést bounded model checking. Nevýhodou této metody je nemožnost začít bounded model checking z jiného než z pouze uloženého stavu. Není možné jako u předešlé metody zahájit bounded model checking z nějakého z předcházejících stavů běhu programu. Další nevýhodou je slabost v ukládání stavu. Pokud dojde ke změně v uloženém stavu je třeba provést příslušnou změnu i v ukládaném resp. obnoveném stavu. Např. u Java programů může dojít ke změně verze Java virtálního stroje (JVM) nebo změna verze model checkeru může způsobit také nutnost změny v ukládání a obnovení stavu. Nevýhoda možnosti verifikace pouze z uloženého stavu se dá částečně odstranit opětovným ukládáním stavů systému např. po určitém časovém intervalu. Nicméně stále tu zůstává nutnost uložení veškerých potřebných informací pro obnovení stavu systému v model checkeru. Těchto informací je značná spousta a jak již bylo zmíněno, s každou změnou verze může dojít k nutnosti změně ukládaných informací. Naopak výhodou je fakt, že nezpůsobuje trvalé zpomalení běhu programu, ke zpomalení dochází pouze v době ukládání stavu.

- **Další Strategie.** Výše uvedené strategie se dají různě kombinovat, např. po určitém časovém intervalu může docházek k ukládání stavu systému a zaznamenávání cesty z tohoto stavu. Po určité době dojde k přemazání uložené cesty a stavu novými informacemi.

  Další možností je modifikace uvedených strategií, jako například Record&Replay trace lze kombinovat s „ořezáváním (slicing)“. Jedná se o metodu, pomocí které se neukládají všechny informace o běhu programu, ale pouze ty informace, které jsou relevantní k přehrání běhu programu do požadovaného stavu.

Kapitola 4

Java PathFinder

Projekt SHADOWS se věnuje léčení programů napsaných v jazyce Java, a proto bylo vybíráno z model checkerů vhodných pro verifikaci Java programů. Byly zkoumány vlastnosti těchto tří známých a zajímavých model checkerů.

- **Bogor** [10] je softwarový model checking framework, který poskytuje vizualizaci, grafické uživatelské rozhraní, a také různé algoritmy po model checking (pro redukci stavového prostoru, vyhledávací heuristiky, abstraktní definice atd.). Bogor lze použít i jako plugin do Eclipse. (Eclipse je integrované vývojové prostředí pro programování Java programů, jeho návrh umožnuje rozšíření prostředí pomocí pluginů [2].) Jedním z možných využití Bogoru, je pro studijní účely. Je možné ho využít pro výuku základních algoritmů model checkingu a jeho podstaty. Zároveň lze Bogor využít pro klasický model checking.

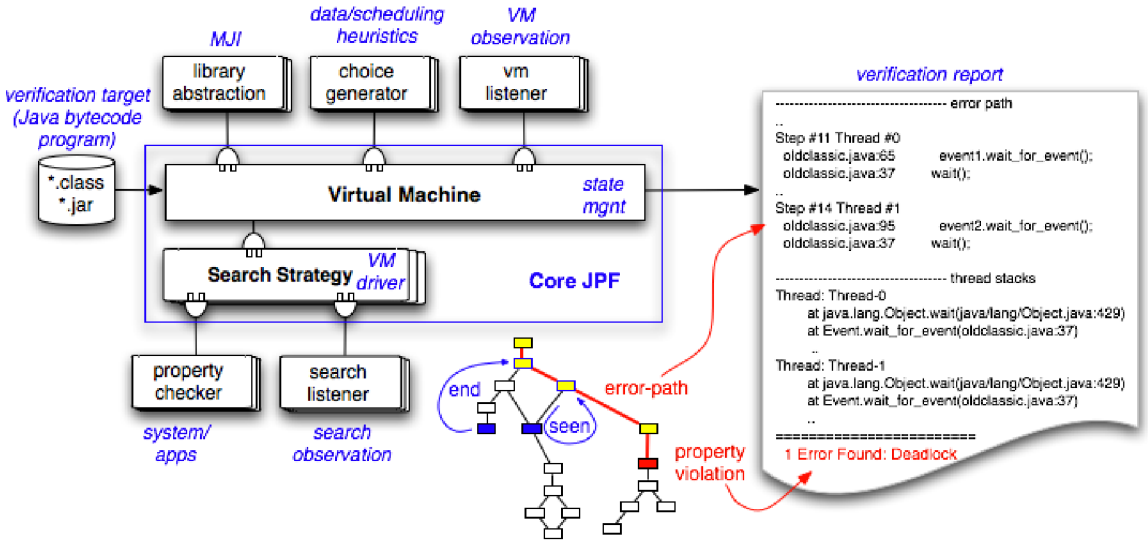
- **Bandera** [8] je model checker pro programy napsané v Javě, které obsahují paralelismus. Bandera překládá zdrojový kód v Jave do vstupního jazyka nějakého jiného existujícího model checkeru jako například SPIN, SMV, SAL, atd. Tyto jiné model checkery zabezpečují vlastní verifikaci systému. Po zverifikování systému Bandera umožňuje namapovat výstup verifikace ze zvoleného model checkeru na původní Java kód.

- **Java PathFinder(JPF)** [9] je model checker, který provádí verifikaci nad Java byte-codem. JPF je implementován jako speciální Java virtuální stroj (JVM), který v sobě přehrává programy určené k verifikaci. Během přehrávání programu kontroluje požadované vlastnosti nebo specifikace systému. JPF je implicitně nastaven na detekci *deadlocks*, *unhandled exception*, *violations of assertions*. Zároveň JPF může detekovat i další vlastnosti systému, které se dají zadat pomocí parametrů. Nebo je možné naimplementovat další vlastní speciální součásti JPF pro verifikaci požadovaných vlastností systému.

Pro vlastní implementaci bounded model checkingu byl zvolen model checker Java PathFinder pro jeho snadnou rozšiřitelnost o další moduly a funkce. Java PathFinder obsahuje řadu *vyhledávacích strategií (search strategies)*, redukce stavového prostoru, různé heuristiky pro prohledávání stavového prostoru atd. Model checker Bogor je také možné rozšířit o vlastní moduly, nicméně výhodou JPF je jeho nasazení již na reálné systémy. Hlavní nevýhodu model checkeru Bandera je nutnost transformovat vstupní zdrojový kódu programu do jiného jazyka. Tím je způsobena nemožnost selodávání průběhu verifikace nad původním Java kódem.

4.1 Základní charakterisitika

Java PathFinder je explicitní stavový model checker pro programy napsané v jazyce Java [9, 13, 17, 19, 22]. Verifikace se provádí na úrovni Java byte-codu. JPF představuje speciální virtuální stroj, ve které se spouští verifikovaný systém. Z důvodu běhu aplikace přímo v JPF, není nutné spouštět program vícekrát nebo ho nějakým způsobem upravovat. JPF neprovádí jednoduchý běh programu, ale vykonává rovnou verifikaci za běhu (runtime). Prohledávání stavového prostoru probíhá pomocí různých prohledávacích strategií, které budou popsány dále. Samotný Java PathFinder je také napsán v jazyce Java, jeho architektura je rozdělena do modulů, které umožňují další rozšiřitelnost. Je možné rozšiřovat již existující moduly o další funkce nebo implementovat nové moduly. Pokud JPF během verifikace nalezne chybu (error), standardně vypíše cestu (trace), která k chybě vedla, a také vypíše relevantní informace o běhu (aktuální vlákno, jednotlivé řádky zdrojového kódu, apod), tyto informace mohou pomoci opravit nalezenou chybu. Základní architektura Java PathFinderu je vyobrazena na obr. 4.1.



Obrázek 4.1: Architektura Java PathFinderu [9]

Vstupem JPF je Java byte-code program, který je určen k verifikaci. Na běh JPF jsou přilinkovány různé moduly, které definují jakým způsobem bude verifikace probíhat. Pomocí nastavení JPF se určí, jaká vyhledávací strategie (*search strategy*) bude použita pro prohledávání stavového prostoru, které listenery (*search listener*) budou s touto strategií použity (pro získání požadovaných informací). Dále lze nastavit generování možností (*choice generator*) nedeterminismu, jedná se o hodnoty vstupních dat, prokládání vláken atd. Další nastavení umožnuje přilinkovat listener založený na sledování jednotlivých kroků virtuálního stroje. JPF tedy systematicky, podle zvolené strategie, prohledává stavový prostor vstupního programu, pokud dojde k detekci chyby. JPF uživateli poskytne zprávu o verifikaci, která obsahuje cestu, která vedla k chybě, typ detekovaného problému a další informace, které byly nastaveny uživatelem.

JPF umožňuje verifikovat programy, které mají více vláken, nicméně neumožňuje verifikaci souběžného běhu vláken (na dvou procesorech). JPF pouze simuluje souběžnost vláken. Zároveň neumožňuje zverifikovat programy, které obsahují nativní metody (metody psané v jiném jazyce než Java). JPF neobsahuje podporu všech knihoven Java, pokud program obsahuje určité knihovny, není ho možné verifikovat [9].

Nicméně umožňuje vytvoření nativních metod, které jsou v rámci verifikace brány jako atomické části kódu a neprovádí se nad nimi verifikace. Metoda je vykonána bez prokládání jinými instrukcemi kódu. Díky tomuto mechanizmu je možné verifikovat programy obsahující knihovny nebo metody, které nechceme brát v úvahu do verifikace. Pro verifikaci vstupních dat obsahuje JPF specializované API, pomocí kterého lze určit, jakých hodnot mohou tyto data nabývat. Jinou možností je volba pro náhodný výběr hodnot vstupních dat. Nastíněné vlastnosti JPF budou dále rozepsány. JPF může simulovat nedeterminismus, pro generování nedeterminismu obsahuje JPF dva mechanismy:

- **Backtracking** znamená, že se JPF může vrátit k vykonanému stavu a nahradit zvolenou možnost (hodnotu proměnné), z které vznikl nedeterminismus, jinou možností a tím vygenerovat nový stav a následně generovat cestu. Tento způsob se aplikuje na postupné rozgenerování všech možných plánovacích sekvencí (možných hodnot).
- **State matching** je založen na mechanismu vyhnutí se generování jednoho stavu systému dvakrát, každý nový stav je uložen na heap. Pokud je daný stav uložen, neukládá se již vygenerovaný stav, ale JPF se vrací k prvnímu nerozgenerovanému stavu a zde se pokračuje s verifikací.

4.2 Specifikace

JPF umožňuje verifikovat různé vlastnosti podle zadaných požadavků. V JPF existují tři základní mechanizmy pro nastavení vlastností: *ordinary assertions*, *gov.nasa.jpf.Property* a *listenery* (`gov.nasa.jpf.SearchListner` nebo `gov.nasa.jpf.VMListener`).

Java assertions se zadávají přímo do zdrojového kódu programu a slouží k získání informací závislých přímo na datech aplikace. Jedná se o účinné získání informací a chování systému. Nevýhodou této metody je nutnost zásahu do zdrojového kódu programu. Zároveň může dojít k nárůstu stavového prostoru, pokud se mají zverifikovat i přidané assertions.

Gov.nasa.jpf.Property je mechanismem zapouzdřujícím kontrolu vlastností (properties). Verifikace těchto vlastností může být nastavena staticky pomocí `search.properties` nebo dynamicky pomocí `jpf.getSerach().addProperty()`. Potom je možné tyto vlastnosti kontrolovat za běhu programu při každé změně pomocí *search objektu*. Základní specifikace, které jsou implicitně v JPF pomocí tohoto mechanizmu kontrolovány jsou následující: deadlocks, assertion violation, uncaught exceptions. Kontrola těchto specifikací je již v JPF naiplementována.

Listenery – `gov.nasa.jpf.SearchListener` a `gov.nasa.jpf.VMListener` jsou dalším mechanismem, který lze využít pro ověření komplexnějších informací. Jedná se o dvě třídy, pomocí kterých jsou naimplementovány různé listnery. Ty slouží k získávání různých informací o běhu programu. V JPF je již řada těchto listenerů naimplementována. Jednotlivé listenery dávají například následující inforamce: `SearchMonitor` – výpis statistický informací o běhu programu (počet vygenerovaných stavů, velikost využité paměti, atd.),

`HeapTracker` – výpis využití haldy (heap) na všech cestách verifikace, `StateSpaceDot` – vytvoří graf vygenerovaného stavového prostoru během verifikace, `MethodTracker` – výpis všech metod, které byly během verifikace volány, atd.

Mechanizmus listenerů umožnuje získávát informace o programu na třech úrovních. *Obecné listenery* poskytují informace o programu získané pomocí rozhraní, nacházejí se mimo program. Jedná se o *SearchListnery* a *VMListenery*. Druhým typem listenerů jsou *specializované vyhledávací listenery*, ty se opět nachází mimo program a jsou k němu pouze linkovány. Nicméně jsou navrženy pro získání specifických informací o programu, nebo jsou navrženy ke konkrétnímu účelu, např. *gov.nasa.jpf.search.heuristic.BFSHeuristic* slouží pro nastavení prohledávání stavového prostoru pomocí BFS strategie. Třetím typem listenerů jsou *vnitřní listenery*, které se nachází v konkrétním balíku uvnitř implementace JPF a umožnují získávat inforamce z daného konkrétního balíčku. Např. balík `gov.nasa.jpf.jvm.bytecode` obsahuje jednotlivé instrukce byte-codu a tedy listener v tomto balíku umožnuje získat nebo měnit primární informace o jednotlivých instrukcích byte-codu, které jsou ostatním listenerům skryty. Nejširší využití nabízejí obecné listenery, které umožnují získat nebo měnit informace v programu „bezpečným“ způsobem.

SearchListenery lze použít pro monitorování prohledávacího procesu stavovým prostorem. Umožnují zaznamenávat například informace o jednotlivých stavech systému nebo vytvářet graf stavového prostoru, který obsahuje inforamce o postupu stavovým prostorem i základní informace o jednotlivých stavech.

VMListenery mohou zaznamenávat nebo měnit jednotlivé kroky provádění programu na úrovni virtuálního stroje. VMListener zle například použít pro monitorování vykonání byte-code instrukcí MONITORENTER a MONITOREXIT, které slouží k synchronizaci v programu. Monitorování těchto instrukcí umožnuje odhalit chybějící synchronizaci nebo naopak detekovat místo vzniku deadlocku.

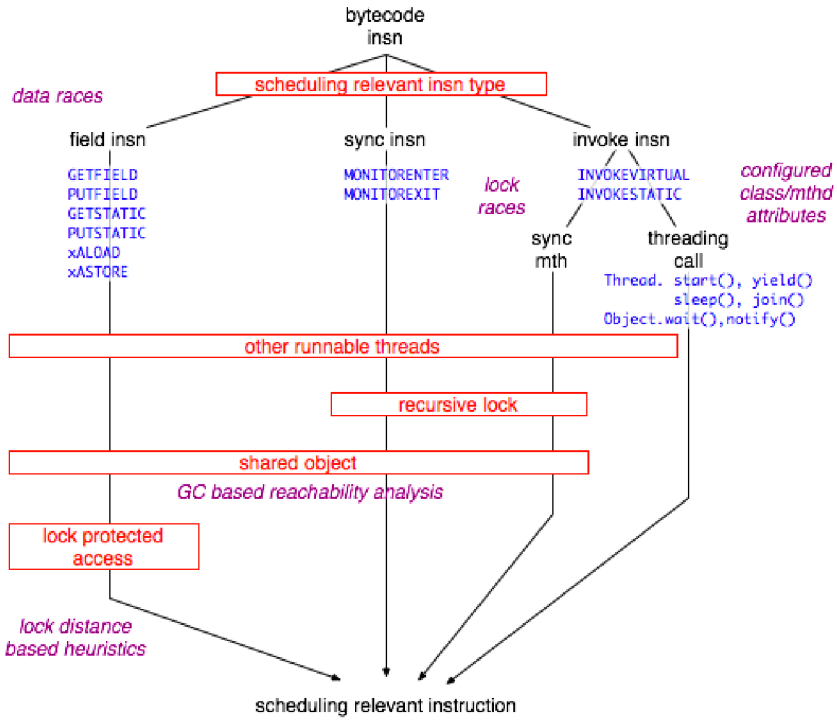
Přilinkování listenerů k běhu JPF lze provést dvěma způsoby. Prvním způsobem je statické nastavení vlastností JPF před jeho spuštěním. Tento mechanizmus umožnuje zvolit jednotlivé listnery, které mají být přilinkovány a také nastavit parametry přidávaných listenerů. Druhou možností je dynamické přidání listenerů do běhu JPF, lisntener je přidán přímo do zdrojového kódu aplikace a vyvolán až za běhu.

4.3 Prohledávání stavového prostoru

JPF obsahuje různé nastavitelné vyhledávací strategie (Search strategies) pro prohledávání stavového prostoru verifikovaného systému. Naimplementovány jsou základní vyhledávací strategie jako *DFS* (Depth First Search – prohledávání do hloubky), *BFS* (Breadth First Search – prohledávání do šírky), A^*, *Best-First* or *BeamSearch*. Zároveň je možné nastavit různé parametry u jednotlivých strategií jako hloubku prohledávání, prioritu stavů, apod. JPF obsahuje i strategie pro náhodný běh programem např.*RandomSearch* nebo *PathSearch*. Vhodným výběrem prohledávací strategie a jejího nastavení lze řídit generování stavového prostoru a tím docílit zverifikování zadané specifikace systému dříve, nemusí dojít k explozi stavového prostoru.

Redukce stavového prostoru. Java PathFinder obsahuje různé mechanizmy, které mají za cíl omezit explozi stavového prostoru. V kapitole 3, která pojednává o Model checkingu jsou vypsány různé přístupy jak docílit redukce stavového prostoru. Mechanizmy redukce v JPF jsou založeny na uvedených principech.

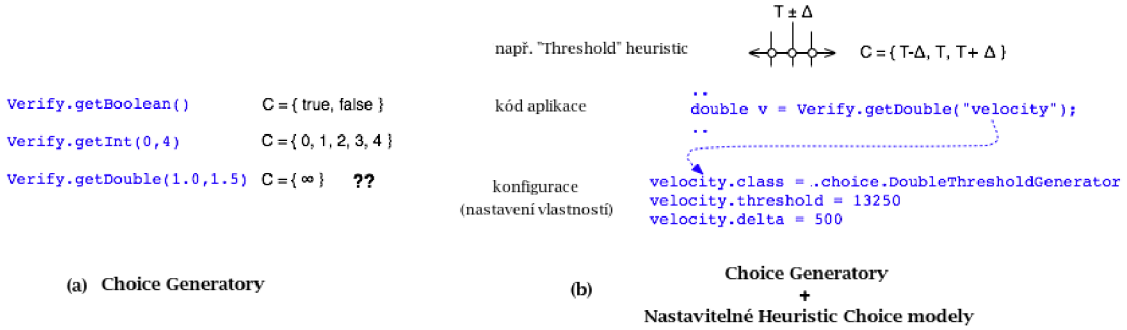
- **Partial Order Reduction (POR)** je metoda, která účinně redukuje počet generovaných stavů. Počet plánovaných rozhodování (větvení) může být značně omezen, pokud seskupíme všechny instrukce v konkrétním vlákně, které nemohou způsobit změnu mimo toto vlákno. Takové instrukce jsou seskupeny do jednoho přechodu, pokud program obsahuje hodně vláken, které mezi sebou nesdílejí data, může být redukce stavového prostoru značná. Oproti tomu pokud program obsahuje množství sdílených dat nebo provázaných operací, redukce pomocí POR je minimální. JPF provádí POR „on-the-fly“, při provádění se nespoléhá na statickou analýzu, ale vyhodnocuje atomické sekce za běhu, resp. určuje, které instrukce musí být vyhodnoceny jako hraniční daného přechodu do nového stavu. Pokud je POR při spuštění JPF povolena, vykonávají se všechny instrukce v aktuálním vlákně do okamžiku než další instrukce způsobí změnu v plánování (scheduling relavant) nebo se může jednat o instrukci, která způsobuje nedeterminismus. Z Java byte-code instrukcí je pouze asi 10% instrukcí, které způsobují změnu v plánování, na obr. 4.2 jsou tyto instrukce vyobrazeny s jejich závislostmi.



Obrázek 4.2: Instrukce mající vliv na plánování [9]

- **Choice Generatory (CG)** jsou dalším důležitým mechanizmem pro práci s nedeterminismem. CG slouží k vytvoření všech hodnot, kterých mohou data v programu nabývat nebo k vytvoření všech možností plánování. CG jsou jedním z možných řešení jak se vyrovnat s problémem vstupních dat. Pomocí rozhraní gov.nasa.jpf.jvm.Verify lze zadat, jakých hodnot mají určité proměnné nabývat a takto specifikovanými možnostmi hodnot proměnných se provádí verifikace. Pokud jsou vstupní data typu `boolean` není problém provést verifikaci nad všemi možnostmi hodnot, u typu `integer` již nastává problém s vygenerováním všechn možností hodnot a u proměnné typu `float` je to již velmi nevhodné.

K řešení tohoto problému slouží CG, pomocí kterých můžeme zadat jakých hodnot má proměnná nabývat při verifikaci (explicitní určení „zajímavých“ hodnoty pro verifikaci). Pomocí CG lze vymezit interval možných hodnot Na obr. 4.3(a) typ boolean nabývá všech možností, typ integer nabývá hodnot ze zvoleného intervalu a u typu float i po vymezení intervalu zůstává velké množství možných hodnot. Proto je zde další mechanizmus jak určit interval i krok mezi jednotlivými hodnotami. Přesné hodnoty proměnné se pak definují až pomocí nastavení parametrů při spuštění JPF (obr. 4.3(b)). Tento mechanizmus lze použít například pokud z hlediska verifikace je potřeba pouze zjistit, zda je proměnná větší nebo menší než určitý práh. Podle výsledku se zvolí jedna ze dvou možností. Je tedy nadbytečné generovat všechny možné hodnoty proměnné, pokud je relevantní pouze vztah hodnoty k prahu).



Obrázek 4.3: Instrukce mající vliv na plánování [9]

- **Explicitní určení Atomicity** spočívá v provedení určitého kódu programu atomicky. Ta část kódu, která má být provedena atomicky je explicitně určena uživatelem pomocí`Verify.beginAtomic()` a `Verify.endAtomic()`. Zároveň musí uživatel zaručit, že díky této atomicitě nedojde k nenalezení chyby v systému.

4.4 Rozšiřitelnost

Pro využití JPF v projektu SHADOWS je podstatná jeho vlastnost rozšiřitelnosti. JPF je open source a je tedy možné doimplementovat jiné vlastní moduly nebo funkce. Pro získání dalších vlastností z verifikovaného systému je možné vytvářet nové listenery, nové prohledávací strategie. Ty mohou být modifikací již naimplementovaných a upraveny pro konkrétní účel nebo je možné vytvářet nové strategie spojením více přístupů dohromady.

Kapitola 5

Implementace

Jak již bylo uvedeno v úvodu, cílem práce bylo navrhnout a posléze implementovat metodu pro ověření opravy v projektu SHADOWS. Cílem opravy je automatické léčení chyb v Java programech. Léčení se specializuje na chyby vzniklé souběžností, kapitola 2.

Pro kontrolu opravy byla zvolena formální metoda Model Checking, kapitola 3. Z důvodu problému stavové exploze není možné provést celý model checking nad reálným systémem. Proto pro vlastní verifikaci byla zvolena modifikace – *bounded model checking* kapitola 3.5. Klasický bounded model checking spočívá v omezení hloubky stavového prostoru, který se má prohledávat. Pokud má být bounded model checking použit pro ověření opravy, není cílem provést model checking od začátku běhu programu do předem definované hloubky, ale zverifikovat konkrétní část programu, kde byla aplikována oprava. K tomuto účelu byly v kapitole 3.6 uvedeny i různé strategie pro navigaci stavovým prostorem do chybového stavu. Potom je již možné provést bounded model checking ne z počátečního stavu, ale z daného stavu (např. chybového stavu) a zverifikovat požadované vlastnosti. K navigaci stavovým prostorem byla vybrána strategie *Record&Replay trace*. Tato strategie byla vybrána pro možnost provést bounded model checking ne pouze z podezřelého (chybového) stavu, ale i z některého z předešlých stavů. Tato vlastnost je velmi výhodná z hlediska ověření opravy. Jako podezřelý stav je detekován chybový stav, ale oprava může provést změny v provádění programu nejenom po chybovém stavu, ale také před ním (např. přidání zámku). Proto je možnost začít verifikaci z některého z předešlých stavů výhodná. Tento fakt přebíjí nevýhody této metody jako náročnost na zaznamenané informace a zpomalení běhu aplikace.

Pro vlastní implementaci byl zvolen model checker *Java PathFinder*, který skýtá velké možnosti rozšiřitelnost o další funkce (Listenery a Search Strategie). *Listenery* lze využít k záznamu potřebných informací, lze tedy listenery použít k záznamu běhu programu. Další část spočívá v přehrání zaznamenané cesty a navigaci do konkrétního stavu. K tomuto účelu lze využít *vyhledávací strategie*, které je samozřejmě opět potřeba upravit a doimplementovat pro přehrání zaznamenané cesty. Pokud se dostaneme do požadovaného stavu stavového prostoru, je možné provést bounded model checking, který se provádí také pomocí modifikované prohledávací strategie, která obsahuje vlastnosti bounded model checkingu.

Kapitola je dále rozdělena do dílčích podkapitol, kde každá podkapitola popisuje určitou část implementace. V první podkapitole je uvedena implementace zvolené metody pro navigaci skrz stavový prostor. Jedná se o strategii Record&Replay Trace. V první fázi byla tato metoda celá implementována v Java PathFinderu, pro ověření a otestování funkčnosti metody a ověření použitelnosti pomocí zvoleného model checkeru. Dále je zde uveden jednoduchý příklad, který demonstruje funkčnost zvolené strategie.

Druhá podkapitola se věnuje bounded model checkingu, po přehrání cesty v JPF se z dosaženého stavu provede bounded model checking. V této podkapitole je popsána nová prohledávací strategie, která slouží k vykonání bounded model checkingu, a kterou je možné dále rozšiřovat o verifikaci dalších vlastností. I zde je uveden jednoduchý příklad pro lepší názornost funkčnosti bounded model checkingu.

Třetí podkapitola se věnuje modifikaci naimplementovaných metod pro účel projektu SHADOWS. Aby bylo možné výše popsaný mechanizmus využít v projektu SHADOWS, je nutné provádět nahrávání běhu programu mimo JPF a v JPF provést až samotnou verifikaci. Tato část implementace není zcela dokončená z důvodů různých peripetií, s kterými bylo potřeba se při implementaci vypořádat. Problémy a jejich návrh řešení jsou uvedeny v dané podkapitole. Zároveň je zde nastíněno, jakým směrem se bude dále pokračovat v dokončení začlenění naimplementovaných metod do projektu SHADOWS.

Poslední podkapitola obsahuje některé testy a výsledky, které jsou pouze dílčí. Celkové testování naimplementovaných metod bude možné až po integraci strategií do projektu SHADOWS.

5.1 Record&Replay trace v JPF

Metoda Record&Replay trace je založena na záznamu potřebných informací pro opětovné přehrání stejného běhu programu.

Aby bylo možné tuto metodu aplikovat pro ověření opravy, bylo nutné nejprve otestování, zda je tento mechanizmus – záznamu cesty a přehrání dané cesty v JPF – vůbec možný. Z toho důvodu byly vytvořeny listenery (`cz.vutbr.fit.tools.CG.SaveCurChoice` a `cz.vutbr.fit.tools.CG.LoadCurChoice`) pro ověření možnosti implementace navigace stavovým prostorem.

JPF je model checker, který provádí rozgenerování celého stavového prostoru a bylo tedy nutné nejprve pochopit jeho princip verifikace. Jak již bylo uvedeno JPF obsahuje řadu prohledávacích strategií. Pro pochopení principu generování stavového prostoru jsou důležité následující základní strategie *DFS* a *BFS*.

- *DFS* strategie: 1) Sestrojení zásobníku (stack), který obsahuje všechny uzly, které se mají expandovat. Na začátku zásobník obsahuje počáteční stav. 2) Je-li zásobník prázdný, prohledávání končí. Jinak se pokračuje v prohledávání. 3) Ze zásobníku se vybere první uzel. 4) Vybraný uzel se expanduje a všechny jeho bezprostřední následníci se uloží do zásobníku. Pokud se nejedná o stav, který již v zásobníku je nebo je již expandovaný, provede se návrat do bodu 2).

  Implementace *DFS* metody v JPF (`gov.nasa.jpf.search.DFSearch`) je navržena takovým způsobem, aby nedocházelo ke generování všech následníků, ale do zásobníku jsou uloženy pouze všechny expandované stavy a informace o možných následovnících pomocí CG (ne celý stav). Další bezprostřední následovníci stavu se generují až před vykonáním stavu. Pro ošetření, aby se negeneroval jeden stav vícekrát, slouží mechanizmus *State matching*, kapitola 4.

- *BFS* strategie: 1) Sestrojení dvou seznamů, první tvoří frontu obsahující všechny uzly určené k vykonání a druhý obsahuje již expandované stavy. Do první fronty se na začátku vloží počáteční stav. 2) Pokud první fronta neobsahuje žádný stav pro rozgenerování prohledávání končí. Jinak se pokračuje v prohledávání stavového prostoru. 3) Z prvního seznamu se vybere čelní stav a uloží se do druhého seznamu

(již expandovaných uzlů). 4) Čelní stav z prvního seznamu se expanduje a jeho bezprostřední následovníci se uloží do prvního seznamu (pokud již nejsou vygenerovány buď v prvním nebo druhém seznamu).

Implementace *BFS* metody v JPF (`gov.nasa.jpf.search.heuristic.BFSearch`) je naimplementována podobným způsobem. V metodě dochází k vygenerování všech bezprostředních následovníků po expandování stavu (všechny možnosti z CG), tyto stavy jsou vytvořeny a před jejich vykonáním dochází pouze k jejich postupnému obnovení.

Ve strategii *BFS* je zajímavou myšlenkou rozgenerování všech možných následovníků (tím je vidět, co který vykonává). Nicméně ve strategii *BFS* se do fronty pro následované obnovení a vykonání stavu uloží všichni bezprostřední následovníci. Cílem strategie Record&Replay trace je vykonat pouze konkrétní možnost a ostatní nerozgenerovávat, resp. vykonat pouze jeden konkrétní běh programu. Toho lze docílit naplánováním a uložením do fronty pouze zvolené možnosti CG.

5.1.1 Record&Replay pomocí ChoiceGenerátorů

Aby bylo možné naplánovat pouze konkrétní možnost CG, je zapotřebí pochopit ovládání ChoiceGenrátorů a volba určitých možností. JPF obsahuje Listener, který demonstruje práci ChoiceGenerátorů `gov.nasa.jpf.tools.CGMonitor`. Ukázka části výpisu z daného listeneru je na obr. 5.1, listener vypisuje možnosti CG, kde došlo k volbě a která možnost byla zvolena. Výpis je ze vzorového verifikačního příkladu JPF (*trunk/examples/Rand.java*). Dalším naimplementovaným listenerem je `gov.nasa.jpf.tools.ChoiceSeletor`, který demonstruje provádění pouze jedné náhodně zvolené možnosti z CG.

```
============================================================ system under test
application:.../trunk/examples/Rand.java

============================================================ search started:...
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
.gov.nasa.jpf.jvm.choice.IntIntervalGenerator[0..1,delta=+1,cur=0]
a=0
..gov.nasa.jpf.jvm.choice.IntIntervalGenerator[0..2,delta=+1,cur=0]
  b=0
    c=0
..gov.nasa.jpf.jvm.choice.IntIntervalGenerator[0..2,delta=+1,cur=1]
  b=1
    c=0
..gov.nasa.jpf.jvm.choice.IntIntervalGenerator[0..2,delta=+1,cur=2]
  b=2
```

Obrázek 5.1: Část výpisu testového příkladu s listenerem `CGMonitor`

SaveCurChoice. SaveCurChoice je listener, který zaznamenává potřebné informace o zvolené možnosti CG do souboru. Pomocí předdefinovaného rozhraní VMListenerů je detekován začátek nové možnosti ChoiceGenerátoru. Po vykonání první instrukce do začátku nové možnosti CG (zvolená možnost se již vykonává), se uloží informace o jaký typ CG se jedná (*BooleanChoiceGenerator*, *ThreadChoiceGenerator*, atd.), o kolikátou možnost z CG se jedná a jaké je aktuální vlákno. Tyto informace jsou postačující k opětovnému přehrání správné možnosti CG. Listener umožnuje následující nastavení (properties):

save_cur_choice.trace – název výstupního souboru se záznamem běhu,

save_cur_choice.comment – znění komentáře, který bude zapsán do výstupního souboru,

vut.logging – (true/false) pro výpis vykonaných možností CG na příkazovou řádku (výpis určený pro testování).

LoadCurChoice Jedná se o listener, který podle vstupního souboru, který obsahuje CG a jejich zvolené možnosti provede načtení cesty a pokud je to možné opětovné přehrání programu koresponující s uloženou cestou (uložená cesta CG koresponduje s CG, které jsou voleny ve spuštěné aplikaci). Listener `gov.nasa.jpf.tools.ChoiceSelector` je základem uvedeného nového listeneru.

Během registrace listeneru při spuštění JPF se načte cesta ze souboru. Pokud se při vykonávání programu narazí na vytváření nového CG, provede se kontrola s načtenou cestou, pokud typ aktuálního CG z uložené cesty koresponduje s typem CG z vykonávaného programu naplánuje se provedení pouze uložené možnosti. Listener umožnuje následující nastavení (propeties):

load_cur_choice.trace – název vstupního souboru se zaznamenanou cestou,

load_cur_choice.after – (true/false) pro možnost pokračovat ve vyhledávání i pokud uložená cesta již skončila nebo uložená cesta nekoresponduje s vykonávaným programem,

load_cur_choice.randomSeed – zadání řady pro Random, která se použije pro náhodný výběr možnosti CG, nekorespondujících s uloženou cestou (určeno pro testování).

5.1.2 Record&Replay pomocí byte-code instrukcí

Z předchozí části vyplývá, že JPF lze použít pro implementaci metody Record&Replay trace. V první fázi byly pro záznam a přehrání cesty využity ChoiceGenerátory, ty lze ale získat pouze v případě záznamu běhu programu v JPF.

Podstata použitelnosti metody spočívá v možnosti přehrání běhu programu pomocí informací, které lze získat z „libovolného“ běhu programu.

Cílem je přehrát běh programu napsaného v Javě, zdrojový kód se nejprve přeloží do spustitelného mezikódu – Java byte-code. Tento kód se posléze spustí pomocí JVM (Java Virtual Machine). Jak již bylo uvedeno, JPF je speciální virtuální stroj, který také provádí běh programu pomocí byte-code instrukcí.

Pro přehrání určitého běhu programu je tedy potřeba zaznamenat pořadí vykonávání jednotlivých byte-code instrukcí a informace o tom, které vlákno dané instrukce vykonává. Záznam cesty je stále prováděn pomocí JPF, ale tentokrát se ukládají ty informace, které lze získat z programu i bez JPF. Mezi další informace, které je potřeba znát pro přesné přehrání cesty programu jsou vstupní data, čtení/zápis do souboru atd. Tyto informace nebyly prozatím brány v potaz a záznam/přehrání cesty se provádí nad programy, u kterých tyto informace nejsou zapotřebí. Čtení/zápis do souboru je složitý problém, JPF nepodporuje knihovny pro čtení/zápis do souboru. Další důležité informace, jako vstupní data, lze ošetřit načtením požadovaných hodnot pomocí CG. Použití CG ovšem vyžaduje zásah do zdrojového kódu programu. Z toho důvodu se zatím programy se vstupními daty neuvažují.

JPF je speciální virtuální stroj, jako takový potřebuje i speciální byte-code instrukce. Mechanizmus přehrání programu v JPF spočívá v načtení Java byte-codu a převedení standardních byte-code instrukcí na instrukce JPF, tyto instrukce plně korespondují s Java byte-code instrukcemi. Mají stejné názvy, stejné funkce i stejné parametry.

Díky faktu provádění programu přímo v JPF a dobrému rozhraní na listenery, je možné připojit listenery téměř kamkoliv. Rozhraní pro *VMListenery* obsahuje metodu pro napojení se na běh programu za každou vykonanou instrukci `instructionExecuted(...)`. Díky tomuto mechanizmu je jednoduché odchytávat důležité informace – všechny vykonané byte-code instrukce, běžící vlákna atd. Jelikož k záznamu potřebných informací dochází stále zatím v JPF, je možné zaznamenat více informací najednou a použít je pro ověření korektnosti přehrání cesty nebo pro informativní výpis.

RecordTrace. Pro záznam cesty, která obsahuje vykonané byte-code instrukce je vytvořen nový listener `cz.vutbr.fit.tools.RecordTrace`. Ten se podobá předešlé implementaci listeneru (`SaveCurChoice`). Rozdíl je v implementovaných metodách pomocí rozhraní *VMListener* a v zaznamenaných informacích během běhu programu. I tento listener umožňuje nastavení parametrů, které ovlivňují jeho funkci:

vut_record_trace.CG – (true/false) pro zaznamenávání informace o zvolené možnosti CG (určeno pro testování),

vut_record_trace.Insn – (true/false) pro zaznamenávání vykonané instrukce byte-codu,

vut_record_trace.Thread – (true/false) pro zaznamenávání informací o aktuálním vlákně (jméno, index),

vut_record_trace.FileName – název výstupního souboru se zaznamenanou cestou,

vut_record_trace.Info – (true/false) zaznamenávání dalších informací (krok vpřed, krok vzad, apod.).

Z nastavení listeneru plyne možnost zaznamenávat různé informace podle požadavků. Aby pří načítání inforamcí zaznamenané cesty bylo patrné, o jakou informaci se jedná, obsahuje každý typ informace svůj předem definovaný prefix. Příklad části souboru se zaznamenanou cestou, která obsahuje všechny možné informace, je na obr. 5.2.

```
/* Record trace, search started. */
application vut.delivereble.TwoThreadsTest
[0] gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet 1 main 0
# main@0 "invokestatic java.lang.Object.<clinit>()V"
# main@0 "invokestatic java.lang.Object.registerNatives()V"
# main@0 "return  java.lang.Object.<clinit>()V"
...
# main@0 "return  vut.delivereble.SimpleThread.<init>(Ljava/lang/String;)V"
% started Martin index 2 %
# main@0 "invokevirtual vut.delivereble.SimpleThread.start()V"
[5] gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet 1 main 0
# main@0 "invokevirtual vut.delivereble.SimpleThread.start()V"
% terminated main index 0 %
# main@0 "return  vut.delivereble.TwoThreadsTest.main([Ljava/lang/String;)V"
[6] gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet 2 main 0
# Marek@2 "runstart"
# Marek@2 "iconst_0"
...
```

Obrázek 5.2: Zaznamenaná cesta pomocí listeneru `cz.vutbr.fit.tools.RecordTrace`

Každý výpis obsahuje na začátku informace o aplikaci, ke které byl pořízen – název aplikace a vstupní parametry, pokud nějaké byly. Dále obsahuje informace o CG (prefix $[x]$, kde x je číslo CG), typ CG, jaká možnost byla zvolena a číslo vybrané možnosti.

Nejčastější položkou záznamu jsou informace, podstatné pro přehrání cesty. Tyto informace jsou: jméno aktuálního vlákna, index aktuálního vlákna a popis byte-code instrukce (prefix #). Posledním typem informace jsou informace o stavech a prohledávání stavovým prostorem (prefix %).

Pomocí listeneru `RecordTrace` lze tedy provést záznam cesty v JPF. Nicméně pokud by byla aplikace spuštěna v JPF pouze s tímto listenerem, nedošlo by k záznamu jedné cesty, ale JPF automaticky prohledává celý stavový prostor. Implicitní prohledávací strategií je DFS. Pokud je tedy cílem zaznamenat pouze jeden běh aplikace, je nutné spustit JPF s jinou vyhledávací strategií, která provádí resp. simuluje pouze jeden běh programu. JPF obsahuje dvě jednoduché naimplementované strategie `RandomSearch` a `PathSearch`. Ty simulují náhodný výběr možnosti CG a jeho provedení. Jinou možností je využit naimplementovaných listenerů `SaveCurChoice` a `LoadCurChoice`. JPF se spustí s aplikací v základním nastavení a s listenerem `SaveCurChoice`. Pomocí něho se zaznamenají všechny možné cesty, jedna z nich se vybere a uloží do souboru, který bude na vstupu druhého listeneru `LoadCurChoice`. Při spuštění JPF s listenerem `LoadCurChoice` se přilinkuje i listener pro záznam relevantních informací o cestě `RecordTrace`. Díky tomu se docílí zaznamenání námi zvolené zajímavé cesty a ne pouze náhodné cesty.

ReplayTrace. Přehrání cesty již nelze implementovat jako listener. Je třeba vybrat a provést stavy ve stavovém prostoru, které korespondují se zaznamenanou cestou. Pro tento mechanizmus je naimplementována nová prohledávací strategie. Tato prohledávací strategie, jako všechny naimplementované prohledávací strategie v JPF, je založena na superclass `Search`. Nová strategie ReplayTrace má základ v prohledávacích strategiích – `DFSearch` a `BFSearch`.

Princip ReplayTrace spočívá v generování všech bezprostředních následovníků aktuálního stavu a výběrem pouze těch, kteří korespondují se zaznamenanou cestou ve vstupním souboru (který je výstupem z `RecordTrace`). Tento mechanizmus se opakuje u každého stavu a tím se docílí rozgenerování pouze té části stavového prostoru, která koresponduje s uloženou cestou. Prohledávání stavového prostoru pro určení korespondujících následovníků se provádí pomocí prohledávací strategie BFS.

`ReplaySearchHeuristic` je tedy nová vyhledávací strategie založená na principu `Search` strategií, a její základ je převzat z vyhledávacích strategií `DFSearch` a `BFSHeuristic`. Zásadní rozdíl spočívá v rozgenerovávání jednotlivých následovníků a jejich provádění. Stejně jako listenery i vyhledávací strategie skýtají možnost nastavení parametrů. U této strategie lze nastavit následující:

vut.replay_search.QueueLimit – maximální délka fronty stavů, které se ukládají ke generování (omezení použití paměti),

vut.replay_search.FileName – název vstupního souboru se zaznamenanou cestou,

vut.replay_search.ShowInfo – (true/false) podmínka pro vypisování průběžných informací během prohledávání (určeno pro testování),

vut.replay_search.HeurisiticName – volba prohledávací strategie stavovým prostorem (DFS, BFS).

Při zaregistrování vyhledávací strategie po spuštění JPF se nejprve načte zaznamenaná cesta ze vstupního souboru. Pro načítání cesty byla implementována nová třída `cz.vutbr.fit.tools.LoadRecord`. `LoadRecord` slouží k načtení cesty ze souboru a současnému rozdělení jednotlivých informací. Ty jsou zapsány do struktury `Trace_ti_insn`,

která má formu seznamu. `Trace_ti_insn` umožnuje uchovávat příznak o následnosti jednotlivých informací, a také umožnuje jednoduchý přístup k dílčím datům.

Dalším krokem je načtení struktury `Trace_ti_insn` do třídy `CheckStep`, která provádí samotné porovnání zaznamenané cesty s během programu. Porovnání běhu se provádí po jednotlivých krocích. Po vygenerování nového stavu se zavolá metoda `CheckState()`, která porovná aktuální krok programu s aktuálním krokem zaznamenané cesty. Aktuální krok záznamu, je první krok záznamu, který ještě nebyl přehrán prováděným programem.

Po spuštění prohledávání stavového prostoru pomocí strategie `ReplaySearchHeuristic` se začne prohledávat stavový prostor stejným způsobem jako u strategie BFS. Po vykonání počátečního stavu dojde k rozgenerování bezprostředních následovníků. Po vygenerování následovníků dojde k jejich zařazení do fronty, kde čekají na vykonání, zároveň dojde k nastavení příznaku`generateChildren`. Ten určuje, zda má být stav vykonán a jeho následovníci generováni. Pomocí příznaku dochází k vykonávání pouze těch stavů, které korespondují s uloženou cestou v souboru.

Určení, zda aktuální stav (krok) programu koresponduje s uloženou cestou, se provádí v metodě `CheckState()`, třídy `CheckStep`. Na začátku prohledávání byla zaznamenaná cesta předána do třídy `CheckStep`. Po záznamu cesty je možné se pohybovat po jednotlivých krocích, aktuálním krokem zaznamenané cesty je poslední nevykonaný krok v programu. Parametrem metody `CheckState()` je aktuální krok prováděného programu, ten se porovnává s aktuálním krokem zaznamenané cesty. Určení korespondujících kroků se provádí podle shody jednotlivých byte-code instrukcí a vlákna, které je vykonává. Porovnání byte-code instrukcí a identifikace vlákna je jednoduchá, mají stejnou signaturu (záznam i přehrání se provádí pomocí JPF). Problém pro porovnání jednotlivých kroků nastává u hranice přechodu mezi jednotlivými stavy. Hranicí přechodu bývá typicky instrukce pro načtení/uložení proměnné nebo změnu aktuálního vlákna. Pokud v místě načtení/uložení proměnné dochází k větvení (generování nových stavů). Je třeba dávat pozor, zda k vykonání instrukcí dochází před ukončením přechodu, a je tudíž třeba získat načtenou/uloženou hodnotu proměnné i ve všech rozgenerovaných stavech, vykonání instrukce se opakuje na začátku nového přechodu. Může nastat i situace, kdy je vykonání instrukce naplánované až na začátku nových přechodů. V případě jednoho běhu JPF tento problém nevzniká, problém nastává při porovnávání přechodů dvou běhů JPF. Je tedy nutné s tímto problémem počítat při porovnávání jednotlivých kroků programu.

Při určování korespondence jednotlivých kroků je nutné také počítat s faktem, kdy může skončit zaznamenaná cesta a prohledávání stavového prostoru není ukončeno. V takovém případě lze provést buď ukončení prohledávání s koncem zaznamenané cesty nebo dále provádět náhodný výběr a dokončení jednoho běhu programu.

5.1.3 Příklad na Record&Replay trace

Pro ukázku funkčnosti strategie Record&Replay, je zde uveden příklad `TwoThreads`. Jedná se o jednoduchý příklad, který provádí prokládání dvou vláken. Každé z vláken v cyklu vypisuje na standardní výstup svoje jméno a index průchodu cyklem. Na obr. 5.3 je uveden zdrojový kód programu.

```
class SimpleThread extends Thread {
  ...
  public void run() {
    for (int i = 0; i < 2; i++) {
      System.out.println(i + " " + getName());
    }
    System.out.println("DONE! " + getName());
  }
}
public class TwoThreads ... {
  public static void main (String args[]) {
    new SimpleThread("First-thread").start();
    new SimpleThread("Second-thread").start();
  }
}
```

Obrázek 5.3: Zdrojový Java kód příkladu pro demonstraci strategie Record&Replay

V prvním kroku dojde k záznamu jednoho běhu programu pomocí JPF. Na obr. 5.4 je uveden jeden z možných běhů programu v JPF s listenerem pro záznam běhu do souboru.

```
...
-----------------------------------listener recordTrace Started.
0 First-thread
1 First-thread
DONE! First-thread
0 Second-thread
1 Second-thread
DONE! Second-thread
-----------------------------------listener recordTrace Finished.
...
```

Obrázek 5.4: Jeden z možných výstupů programu při záznamu cesty v JPF

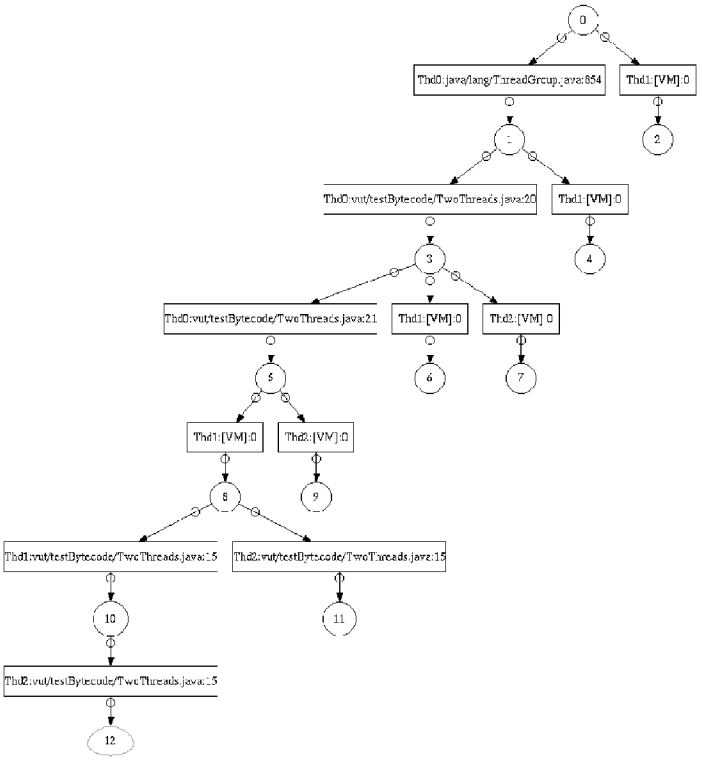
Pomocí nastavení listeneru bylo zvoleno uložení pouze nejnutnějších informací pro přehrání cesty programu. Zaznamenané informace jsou jméno aktuálního vlákna, jeho index a vykonané byte-code instrukce. Ukázka části záznamu cesty je na obr. 5.5.

```
/* Record trace, search started. */
application: vut.testBytecode.TwoThreads
# main@0 "invokestatic java.lang.Object.<clinit>()V"
# main@0 "invokestatic java.lang.Object.registerNatives()V"
...
# First-thread@1 "runstart"
# First-thread@1 "iconst_0"
# First-thread@1 "istore_1"
# First-thread@1 "goto 17"
# First-thread@1 "iload_1"
# First-thread@1 "iconst_2"
...
# Second-thread@2 "invokevirtual java.io.PrintStream.println(Ljava/lang/String;)V"
# Second-thread@2 "return  vut.testBytecode.SimpleThread.run()V"
/* Record trace, search finished. */
```

Obrázek 5.5: Zaznamenaná cesta korespondující s během programu z obr. 5.4

Přehrání zaznamenané cesty v JPF demonstruje následující obr. 5.6. Na obrázku je patrné generování stavového prostoru. Z každého stavu došlo k vygenerování všech bezprostředních následovníků. Nicméně vykonávání vygenerovaného stavu a jeho následovníků pokračuje pouze tím stavem (stavy), které korespondují se zaznamenanými kroky cesty.

V uvedeném příkladě se jedná vždy pouze o jeden korespondující následující stav. Generování cesty končí po dosažení posledního stavu uložené cesty.



Obrázek 5.6: Graf vygenerovaného stavového prostoru pomocí cesty z obr. 5.5

5.2 Bounded model checking v JPF

Po implementaci metody Record&Replay trace je dalším krokem provedení bounded model checkingu v JPF.

Bounded model checking v JPF je možné aplikovat na různé prohledávací strategie, např. na často jmenované BFS a DFS. Pomocí parametrů strategií je možné zadat požadovanou největší hloubku prohledávání k. Standardně se bounded model checking provádí z počátečního stavu do zvolené hloubky k, zároveň je možné zadávat další nastavitelné parametry prohledávání, a také přidání dalších listenerů pro verifikaci dalších vlastností sledovaného systému.

Pokud vezmeme formální zápis bounded model checkingu z kapitoly 3.5, kde je verifikovaný systém popsán pomocí Kripkeho struktury, je možné najít podobnou strukturu systému i v JPF. V JPF se nejprve nevytváří model verifikovaného systému, ale verifikace se provádí přímo nad reálným systémem. Nicméně i zde se dá najít podobnost v průchodu stavovým prostorem s výpočetním stromem Kripkeho struktury. JPF pří prohledávání stavového prostoru vytváří stavy systému i přechody. Přechod mezi stavy je definován pomocí vykonaných byte-code instrukcí. Stav systému je charakterizován aktuálním stavem celého systému (hodnoty proměnných, aktuální vlákno, atd.). Vlastnosti, které se mají u systému verifikovat jsou formálně zapsané pomocí temporálních formulí. Při verifikace pomocí JPF lze nadefinovat verifikované vlastnosti a ověřit je pomocí listenerů. Ty se dají připojit na ve-

rifikovaný systém do požadovaného místa a s generováním stavového prostoru kontrolují požadované vlastnosti.

JPF umožňuje získat informaci, zda omezená cesta obsahuje smyčku (*k-loop*) pro verifikaci nekonečných vlastností. V JPF nedochází k opětovnému vytváření shodného stavu. Pokud byl jednou stav vygenerován, nedochází k jeho opětovnému generování v jiném místě stavového prostoru, ale pouze se využiji již existující stav. Proto je snadné zjistit, zda na generované cestě existuje zpětná smyčka.

JPF implicitně umožňuje provádět bounded model checking z počátečního stavu. Cílem práce je ovšem provést bounded model checking z předem určeného stavu a ne pouze ze stavu počátečního. K tomu účelu byly vytvořeny předcházející listenery a prohledávací strategie.

První možností je využití prvních listenerů (`LoadCurChoice` a `SaveCurChoice`). Pomocí listeneru `SaveCurChoice` se zaznamená cesta až do podezřelého (chybového) stavu. Záznam cesty tedy končí v místě, kde se posléze má začít provádět bounded model checking nebo je možné určit, zda se před začátkem bounded model checkingu má provést pár kroků zpět (`backtrack()`) pro zverifikování celého okolí chyby. Druhým krokem je přehrání záznamu cesty pomocí `LoadCurChoice`, navigace stavovým prostorem končí po vygenerování posledního stavu zaznamenané cesty.

Druhou možností je využití listeneru pro záznam cesty `RecordTrace`, jehož záznam cesty také končí v podezřelém stavu. Potom se JPF spustí s novou prohledávací strategií `ReplaySearchHeurisicBMC`, která vykonává navigaci stavovým prostorem pomocí zaznamenané cesty. Jedná se o modifikaci prohledávací strategie `ReplaySearchHeurisic`. Rozdíl spočívá v doimplementování funkce, která po ukončení navigace stavovým prostorem umožnuje spustit bounded model checking. Strategie obsahuje další nastavení určená pro bounded model checking, nastavení slouží k určení počátečního stavu bounded model checkingu, k určení hloubky provádění generování stavového prostoru, atd.

Proto, aby bounded model checking sloužil k účelu ověření opravy, je třeba verifikovat specifické vlastnosti chování systému v okolí chyby. K tomuto účelu jsou implementovány další listenery, které sledují požadované vlastnosti. Přilinkované listenery k běhu programu v JPF sledují požadovanou vlastnost od začátku běhu až do jeho konce. Tento mechanizmus v případě bounded model checkingu není žádoucí.

Například pokud listener slouží k sledování přístupů ke sdílené proměnné, není cílem zaznamenávat veškeré přístupy ke sdílené proměnné, ale pouze přístupy v okolí chyby. Je snahou sledovat pouze ty přístupy, které mohou mít vliv na detekovanou chybu. Cílem je zapnout funkci listenerů až při bounded model checkingu a ne již při průchodu stavovým prostorem pomocí prohledávací strategie. Docílení tohoto mechanizmu si vyžaduje přidání parametru (začátku verifikace) do listeneru. Tento příznak musí být možné nastavit pomocí vyhledávací strategie a jeho příznak předán do přilinkovaných listenerů. Implementace tohoto mechanizmu si vyžaduje zásah do navrženého rozhraní mezi vyhledávacími strategiemi a listenery.

ReplayTraceAndBoundedModelChecking. Jak bylo zmíněno výše, modifikací prohledávací strategie `ReplaySearchHeuristic` je možné provádět bounded model checking po průchodu stavovým prostorem do podezřelého stavu. Jedná se o strategii `ReplaySearchHeuristicBMC`, která obsahuje rozšíření nastavení parametrů o parametry bounded model checking:

vut.replay_search.bmcBackStep – počet kroků, o které se má strategie vrátit před spuštěním bounded model checkingu,

vut.replay_search.bmcStartDepth – explicitní určení hloubky, ze které se začíná provádět bounded model checking, explicitní určení hloubky na cestě, kde se začíná provádět bounded model checking, (pokud je hodnota rovna -1, bounded model checking začíná na konci zaznamenané cesty),

vut.replay_search.bmcMaxDepth – maximální hloubka provádění bounded model checkingu (verifikace může skončit i v menší hloubce – konec programu).

Do prohledávací strategie byly tedy přidány zmíněné parametry, které řídí navigaci stavovým prostorem a následné vykonání bounded model checkingu. Zadané parametry určí, kde skončí navigace a začne verifikace, resp. způsobí generovování a vykonávání všech následníků stavů, které mají být provedeny. Prohledávací strategie se řídí pomocí již popsaných heuristik, ty jsou také součástí nastavení vstupních parametrů. Prozatím jsou implementovány pouze dva prohledávací mechanizmy BFS – `HeuristicBFS_VUT` a DFS – `HeuristicDFS_VUT`.

5.2.1 Příklad na Bounded Model Checking

Pro ukázku funkčnosti bounded model checkingu byl zvolen příklad `Bank Account` [11]. Jedná se o vzorový příklad, který je implementován jako program s více vlákny. Používá se pro demonstraci souběžného přístupu ke sdílené proměnné pomocí více vláken. Souběžný přístup může způsobit chybu v programu – data race. Tento příklad je použit i jako vzorový v projektu SHADOWS pro opravy chyby – data races. Jednou možností, jak využít bounded model checking spočívá v určení, zda v programu opravdu v daném místě dojde k chybě nebo zda se jedná pouze o false alarm. Data race – je chyba, která vzniká u souběžného přístupu dvou vláken ke sdílené proměnné, přičemž jedno vlákno se snaží o zápis do sdílené proměnné. Problém nastává v případě nekonzistence přečtených a uložených dat.

Program `Bank Account` obsahuje dvě třídy. Hlavní třídou je třída `Bank`, která vytváří a spouští další vlákna. Tato vlákna reprezentují jednotlivé účty – třída `Account`. Každé vlákno může simulovat různé operace nad účty. Každá z operací nad účty je ukončena voláním metody `Service()` z třídy `Bank`. Zdrojový kód metody je na obr. 5.7. Metoda má dva parametry: id je identifikace účtu a sum reprezentující celkový obnos transakce.

```
public static void Service(int id, int sum) {
    // operace s~obnosem na~ucte
    accounts[id].Balance += sum;
    // operace s~celkovym obnosem v~bance
    Bank_Total += sum;
}
```

Obrázek 5.7: Zdrojový kód metody z třídy `Bank`

Na první pohled uvedený zdrojový kód neobsahuje chybu. Ale problém nastává při změně celkové částky v bance (`Bank_Total`). Programátor se mylně domnívá, že uvedená operace se provádí jako atomická. Při převodu do mezikódu, který se posléze vykonává, má tato jednoduchá operace (přičtení) čtyři Java byte-code instrukce: (1) získání hodnoty Bank_Total, (2) získání hodnoty sum, (3) sečtení hodnot a (4) uložení nové hodnoty Bank_Total.

Problém tedy nastává pokud zadanou operaci provádí dvě vlákna najednou. Pokud si obě vlákna načtou hodnotu `Bank_Total`, aktualizují její hodnotu a posléze zapíší novou

hodnotu sekvenčně. Druhý zápis do `Bank_Total` přepíše aktualizovanou hodnotu prvním vláknem, a tím dochází ke ztrátě informace a k chybě ve výpočtu.

Cílem bounded model checkingu je nyní ověřit, že v uvedeném místě programu může opravdu dojít k „TrueRace“. Uvedené místo programu je považováno za podezřelé.

Nejprve se provede běh programu s přilinkováním listeneru pro záznam cesty. Jedná se o testový příklad, je možné provést nejprve záznam celého běhu programu pomocí `SaveCurChoice`. Ze záznamu listeneru je vidět v jaké hloubce a v kterém běhu se nachází podezřelý stav, který může způsobit problém. Tato hloubka a cesta je poznamenaná do souboru. Dále jsou aplikovány listenery `LoadCurChoice` a `RecordTrace`, které jsou využity pro zaznamenání cesty do požadované hloubky.

Druhým krokem je provedení přehrání uložené cesty, která vede do podezřelého stavu. K provedení je použita prohledávací strategie `ReplayTraceHeuristicBMC` a dále je přilinkován nový listener `Races`, který je navržen pro záznam jednotlivých přístupů k podezřelé proměnné. Proměnná je identifikována podle svého umístění (balíček + třída) a identifikátoru proměnné. V tomto případě je podezřelou proměnou `Bank_Total`.

Listener `Races` začíná uchovávat informace o přístupu ke sdílené proměnné v okamžiku spuštění bounded model checkingu. Po každé provedené instrukci se kontroluje, zda nešlo o instrukci přístupu ke sledované proměnné. Pokud došlo k přístupu k proměnné, zaznamenají se následující informace: hloubka – aktuální vlákno – aktuální metoda – stav systému – typ instrukce. Část záznamu se zaznamenanými informacemi je na obr. 5.8. Z těchto informací lze na závěr vyčíst, zda došlo k „TrueRace“.

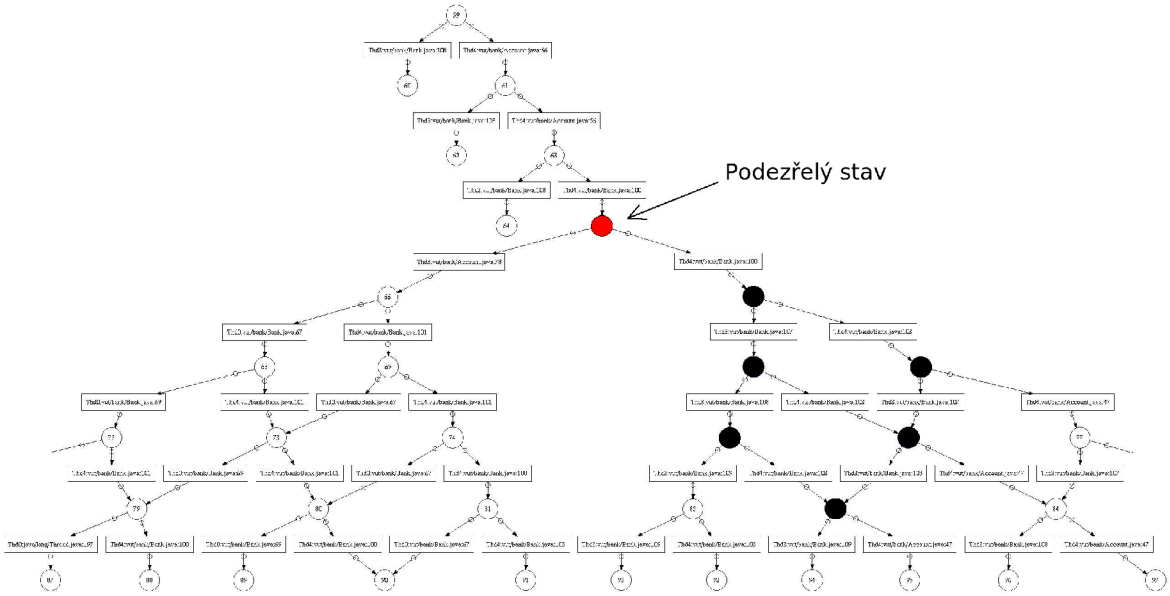
```
First access:
Depth:threadName@methodName: StepID-insn

33:Thread-2@vut.bank.Bank:67-getstatic
33:Thread-3@vut.bank.Bank:67-getstatic
33:Thread-3@vut.bank.Bank:67-putstatic
------------------------------------------------------------
34:Thread-2@vut.bank.Bank:70-getstatic
34:Thread-2@vut.bank.Bank:70-putstatic
34:Thread-3@vut.bank.Bank:70-getstatic
34:Thread-3@vut.bank.Bank:70-putstatic
34:Thread-2@vut.bank.Bank:71-getstatic
34:Thread-3@vut.bank.Bank:71-putstatic
...
```

Obrázek 5.8: Část výstupu z listeneru `Races` pro ověření race

Celý proces přehrání cesty, bounded model checking a vyhodnocení specifikace je znázorněn na obr. 5.9. Obrázek znázorňuje pouze část celého grafu, který reprezentuje generovaný stavový prostor. Nejprve došlo k navigaci stavovým prostorem do podezřelého stavu. Z podezřelého stavu se rozgenerovává stavový prostor a zároveň se pomocí listeneru ukládají informace o přístupu ke sdílené proměnné. Nakonec byly ručně vyhodnoceny zaznamenane informace pomocí listeneru. Na grafu je červeně uveden podezřelý stav, ze kterého byl zahájen bounded model checking. Černě jsou vybarveny ty stavy, ve kterých byl detekován „TrueRace“.

Celý proces přehrání cesty (bounded model checking a vyhodnocení specifikace) je znázorněn na obr. 5.9. Obrázek znázorňuje pouze část celého grafu, který reprezentuje generovaný stavový prostor. Nejprve došlo k navigaci stavovým prostorem do podezřelého

stavu. Z podezřelého stavu se rozgenerovává stavový prostor a zároveň se pomocí listeneru ukládají informace o přístupu ke sdílené proměnné. Nakonec byly ručně vyhodnoceny zaznamenané informace pomocí listeneru. V grafu je označen podezřelý stav, ze kterého byl zahájen bounded model checking. Černě jsou vybarveny ty stavy, ve kterých byl detekován „TrueRace“.



Obrázek 5.9: Graf bounded model checkingu a detekce TrueRace

5.3 Modifikace Replay trace pro projekt SHADOWS

Poslední implementační část se zabývá úpravou strategie Record&Replay trace pro využití v projektu SHADOWS. Nahrávání cesty programu se již neprovádí pomocí JPF, ale mimo model checker. Pro záznam cesty se využívá nástroj ConTest, který byl již zmíněn na začátku práce v kapitole 2. Jedná se o nástroj, pomocí kterého se detekují a posléze léčí nalezené chyby v programu.

IBM ConTest je testovací nástroj, který provádí instrumentaci Java byte-codu programu. ConTest obsahuje také listenerovou architekturu a umožňuje zanesení šumu do programu pro lepší odhalování chyb vzniklých souběžností. Instrumentace kódu spočívá ve vkládání byte-code instrukcí do byte-codu monitorovaného programu. Pomocí těchto vložených instrukcí je možné přilinkovávat listenery do běhu programu, zanášet do programu šum a další. Na architektuře listenerů je založen i mechanizmus pro detekci a následnou opravu chyb v programu.

Pomocí těchto listenerů je možné získat různé informace o vykonávání programu přímo za běhu. Instrumentovaný kód programu se spouští v běžném Java virtuálním stroji. Pomocí instrumentace a parametrů se za běhu programu přilinkovávají požadované listenery ConTestu, které vykonávají zadané události. ConTest tedy může stejným způsobem jako JPF pomocí listenerů sledovat určité vlastnosti nebo částečně modifikovat provádění programu za běhu.

Záměrem je zaznamenávat cestu programu při jeho provádění s přilinkovaným ConTestem. Po aplikaci opravy ConTestem provést v okolí opravy bounded model checking v JPF a ověřit korektnost opravy. Cílem tedy je přehrát zaznamenanou cestu ConTestem v JPF.

Na obr. 5.10 je uvedena část zaznamenané cesty z ConTestu. Záznam cesty se provádí pomocí speciálního listeneru, který byl pro tento účel navržen. Listener zaznamenává všechny možné informace, které lze z ConTestu o běhu programu získat. Záznam cesty a implementace zmíněného listeneru není součásti diplomové práce, ale je převzat z projektu SHADOWS.

```
3:main(java.lang.Thread@19134f4):BASICBLOCK:.../bank Bank.java <clinit>() 14-14,20-20,24-24 1
9:main(java.lang.Thread@19134f4):THREAD_BEGIN:.../bank Bank.java main(java.lang.String[]) 35 1
10:main(java.lang.Thread@19134f4):BASICBLOCK:.../bank Bank.java main(java.lang.String[]) 35-35 1
....
22:Thread-4(bank.Account@f4a24a):THREAD_END:null
23:Thread-3(bank.Account@17182c1):BASICBLOCK:.../bank Bank.java Service(int,int) 85-85,87-88 1
33:Thread-3(bank.Account@17182c1):BASICBLOCK:.../bank Account.java run() 54-54 2
34:Thread-3(bank.Account@17182c1):BASICBLOCK:.../bank Account.java run() 54-54 1
34:Thread-3(bank.Account@17182c1):BASICBLOCK:.../bank Account.java run() 62-63 1
...
531:main(java.lang.Thread@19134f4):BASICBLOCK:.../bank Bank.java main(java.lang.String[]) 73-73 1
531:main(java.lang.Thread@19134f4):BASICBLOCK:.../bank Bank.java main(java.lang.String[]) 77-77 1
533:main(java.lang.Thread@19134f4):THREAD_END:null
```

Obrázek 5.10: Část záznamu cesty programu pomocí ConTest listeneru

Záznam cesty obsahuje informace o aktuálním vlákně (jméno, třída, index), další informací je událost ConTestu (bude dále rozepsané), informace o zdrojovém kódu (umístění a jméno zdrojového souboru, aktuální metoda), poslední informací je rozsah události ConTest a pořadí byte-code instrukce na řádku, kterou daná událost ConTestu začíná.

Z hlediska přehrání cesty je pro nás důležité zaznamenávat místa nedeterminismu, která určují směr dalšího provádění programu. Nástroj ConTest program rozděluje na tzv. BasicBlocky. BasicBlock tvoří část programu, ve které nedochází k nedeterminismu. Na uvedeném záznamu cesty programu je vidět několik BasicBlocků s různou délkou. Čísla na konci každého řádku záznamu odpovídají číslům řádků ve zdrojovém kódu programu odkud – kam sahá BasicBlock. Posledním číslem záznamu cesty je pořadové číslo první instrukce BasicBlock na zdrojovém řádku kódu.

Další zaznamenanou událostí ConTestu je začátek vlákna Thread_Begin nebo ukončení vlákna Thread_End. Tyto události jsou také vznikem nedeterminismu a jsou tedy potřebné v záznamu cesty.

V záznamu cesty je patrný rozdíl mezi informacemi získanými z JPF a informacemi, které jsou získány z ConTestu. Další odlišností je rozdíl mezi BasicBlocky ConTestu a stavy v JPF. BasicBlocky v podstatě tvoří jednotlivé stavy a přechody v systému, ty jsou ovšem odlišné od stavů a přechodů v JPF.

Po získání záznamu cesty z ConTestu je třeba danou cestu přehrát v JPF. Pro přehrání cesty je třeba najít korespondující informace v běhu programu pomocí JPF. Po získání návaznosti mezi uloženými informacemi a informacemi za běhu programu je možné aplikovat strategii pro navigaci stavovým prostorem a následný bounded model checking.

Pro přehrání korespondující cesty v JPF jsou dvě možnosti. Při první možnosti se v JPF spustí původní program (bez instrumentace ConTestu) a vyhledají se souhlasné informace se zaznamenanou cestou ConTestem. Tyto informace slouží k navigaci stavovým prostorem. Druhou možností je přehrání instrumentovaného kódu v JPF. Instrumentace ConTestem sice přidává do byte-codu programu další instrukce, ale běh programu se provádí pomocí

standardního virtuálního stroje. JPF je speciálním virtuálním strojem a je tedy teoreticky možné přehrání instrumentovaného kód.

5.3.1 Replay trace pomocí původním kódu

Při prozkoumávání první možnosti byl postup následující. Postupně se specifikovaly informace získané ze záznamu cesty a hledaly se odpovídající instance těchto informací v JPF. Protože JPF vezme původní byte-code instrukce a transformuje je na vlastní byte-code instrukce, dochází ke změně nejen instrukce, ale také se mění parametry a informace, které lze získat.

Ke všem zaznamenaným informacím z ConTestu byly nalezeny korespondující informace až na identifikaci instrukce. V ConTestu se jednotlivé byte-code instrukce identifikují pomocí čísla řádku ve zdrojovém kódu (tato informace je v JPF obsažena) a pomocí pořadí instrukce na řádku, tuto informaci se nepodařilo získat. Z důvodů přepsání původních byte-code instrukcí na odpovídající JPF instrukce se ztrácí některé informace obsažené v původním mezikódu. Pro získání pořadí instrukce na řádku v původním kódu by byl nutný hlubší zásah do fungování JPF. Požadovaná informace by se musela získat v průběhu tranformace původních instrukcí na instrukce JPF. Transformace původního kódu na instrukce JPF má několik fází. Nejprve dojde k rozdělení kódu na dílčí části jako jsou metody, v další fázi se postupně prochází jednotlivé instrukce, a ty se postupně tranformují na instrukce JPF. Zároveň dochází ke změně potřebných parametrů, které každá byte-code instrukce potřebuje pro správnou funkci. V této fázi dochází k zásadní změně. Nelze již zpětně získat původní informace instrukce.

Pro uchování informace, kolikátou insturkcí na řádku daná instrukce je, by bylo třeba zasáhnout do superclass všech JPF instrukcí a přidat nový parametr. Nastavení tohoto parametru by bylo třeba přidat do „správných“ konstruktorů JPF instrukcí. Dále by bylo nutné zasáhnout do jendnotlivých částí tranformace původního kódu na instrukce JPF a na všechna potřebná místa zadat nejprve získání požadovaného parametru a pak jeho předání do nové instrukce JPF. Tento mechanizmus je náročný a způsobil by rozsáhlé změny v JPF.

Vzhledem k tomu ,že byte-code instrukce jsou jednou z hlavních informací, která je potřebná pro opětovné přehrání běhu programu, byla snaha tento mechanizmus naimplementovat tak, aby byly získány potřebné informace bez velkého zásahu do JPF. Nicméně po delší době testování bylo od této možnosti upuštěno a nebyla naimplementována.

5.3.2 Replay trace pomocí instrumentovaného kódu

Druhou možností je přehrání instrumentovaného kódu přímo v JPF. Cílem je přehrát původní Java byte-code bez instrumentace. Je nutné se nějakým způsobem s přidanými instrukcemi ConTestu vypořádat. Ponechání instrukcí kódu při verifikaci není možné. Pomocí přidaných instrukcí ConTestu se volají různé moduly ConTestu, které není možné pomocí JPF zverifikovat. Vyřazení těchto modulů z verifikace pomocí standardního mechanizmu JPF si vyžaduje zásah do zdrojového kódu a změnu metod těchto modulů na nativní metody. Nativní metody již nepodléhají verifikaci. Nástroj ConTest ovšem není možné modifikovat a tudíž nelze daný mechanizmus JPF využít.

Jinou možností, jak se vypořádat s přidanými instrukcemi ConTestu, je jejich přeskakování namísto vykonání. Princip přeskočení (skip) naplánované instrukce je podobný jako u listeneru `LoadCurChoice`, kde se z naplánovaných možností CG provedla pouze jedna možnost a ostatní se přeskočily.

Principem nového listeneru je přeskakování všech instrukcí ConTestu, které byly do byte-codu přidány a zároveň přeskakování všech souvisejících instrukcí (parametrů, návratových hodnot, atd.).

Instrumentace ConTestem přidává do kódu instrukce pro volání statických metod ConTestu. Statické metody jsou volány pomocí instrukce `invokestatic`. Všechny metody ConTestu jsou tedy volány pomocí instrukce `invokestatic`, nicméně každá metoda má jiný počet parametrů a jiné typy parametrů a návratové hodnoty. Je tedy nutné ošetřit jakékoliv volání metody ConTestu a přeskočit správný počet byte-code instrukcí před a za touto instrukcí.

Každá instrukce, která volá metodu Contestu obsahuje informace o tom jaká metoda ConTestu se volá a odkud. Díky tomu lze přesně určit, které instrukce byly přidány ConTestem. Zároveň tato instrukce obsahuje informaci před kterou původní instrukci byla přidána a také informaci o původní instrukci (na jakém řádku kódu se nachází a kolikátou instrukcí řádku je). Cílem je dané instrukce během provádění programu v JPF pouze přeskakovat a ne je zcela vymazat.

Nový listener `SkipConTestInstruction` je založen na prohledávání naplánovaných instrukcí a detekcí instrukcí ConTestu s jejich parametry. Po jejich detekci dojde k nastavení jejich příznaku na přeskočení při vykonávání naplánovaných instrukcí.

ConTest vždy přidává instrukci pro volání statické metody, nicméně každá metoda může mít jiný počet parametrů, a proto je potřeba určit, jaká metoda má být volána.

V listeneru `SkipConTestInstruction` dochází pouze k detekci, zda se jedná o instrukci přidanou ConTestem. Určení počtu parametrů volané metody a tedy počtu přeskakovaných instrukcí dochází ve třídě `Skipping`. Jednotlivé typy metod, které mohou být pomocí ConTestu přidány, jsou obsaženy v interní dokumentaci ConTestu. Během implementace přeskakování jednotlivých typů metod se objevil následující problém.

Během instrumentace zdrojového kódu programu nedochází pouze k přidání dalších instrukcí ConTestu, ale také ke změně určitých byte-code instrukcí. Změna se týká instrukcí pro vlákna (start, stop, yield, join, atd.). ConTest původní instrukce, které pracují s vlákny, nahradil voláním vlastních metod, které mají dané vlákno jako parametr. K provedení původní instrukce dojde během vykonání metody ConTestu. Pokud jsou tyto metody v JPF přeskakovány, původní instrukce se neprovedou a nastává problém.

Je tedy třeba implementovat mechanizmus, pomocí kterého se docílí provedení, resp. obnovení původních instrukcí, které byly během instrumentace kódu odstraněny a správně je zpět začlenit do kódu. Z principu fungování JPF se nabízejí dvě možnosti, jak vytvořit požadovanou instrukci:

- **Byte-code instrukce JPF.** Princip spočívá v přetransformování instrukce ConTestu na původní instrukci zdrojového kódu nebo vytvoření nové instrukce JPF. Vždy na začátku spuštění programu v JPF dojde k transformaci původních byte-code instrukcí na byte-code instrukce JPF. JPF obsahuje mechanizmus pro vytváření vlastních instrukcí z původních instrukcí.

  Pro transformaci instrukce ConTestu na jinou instrukci JPF je potřeba znát informace, které již instrukce JPF neobsahují. Pro vytvoření resp. transformaci JPF instrukce je zapotřebí zadat jako parametry odkaz do ConstatPool tabulky (více informací fungování JVM je uvedeno v [18]). Tento odkaz je znám pouze u původních byte-code instrukcí a není již k dispozici u instrukcí JPF (ty mají vlastní mechanizmus). Mechanizmus vytváření resp. modifikace instrukce JPF na jinou bez využití

naimplementovaných metod by vyžadovala velký zásah do JPF. Z toho důvodu není tato možnost pro řešení problému využita.

- **Standardní byte-code instrukce.** Druhou možností je vytvoření nové byte-code instrukce ještě před změnou na instrukce JPF. Instrukce ConTestu se přetransformuje nebo se vytvoří nová byte-code instrukce místo původní a teprve posléze se transformuje na instrukci JPF. Vzhledem ke změně instrukce ještě před převodem na JPF instrukci, jsou k dispozici všechny potřebné informace pro vytvoření nové byte-code instrukce.

Druhá z uvedených možností byla hlouběji prostudována. Pro vytvoření nové fungující instrukce je třeba předat správné odkazy do ConstantPool tabulky, zadat správně parametry a v neposlední řadě správně začlenit mezi ostatní instrukce.

Tato část implementace – vytvoření nové instrukce, není zcela funkční. Metoda pro vytváření nové instrukce je v rozpracované fázi.

Listener `SkipConTestInstruction` se také nachází v rozpracované fázi. Listener je v tuto chvíli schopný přeskočit vykonávání všech instrukcí ConTestu i s jeho parametry, které jsou do kódu přidány navíc. Avšak stále neumožňuje nahrazení instrukcí ConTestu pro práci s vlákny za původní instrukce. Je tedy možné v JPF přehrát instrumentovaný program, který neobsahuje vlákna.

5.4 Výsledky a Testy

Funkčnost jednotlivých částí implementace listenerů a vyhledávácích strategií byla testována během jejich návrhu a po jejich dokončení. Každá z výše uvedených částí implementace si vyžadovala jiný výběr testových programů. Testové programy byly vybírány podle požadavků, které mají jednotliví listenery a strategie splňovat. Některé z testových příkladů jsou součástí přiloženého DVD media.

Pro zhodnocení dosažených výsledků byly vytvořeny různé běhy vybraných programů. Jednotlivé běhy se vykonávály v uvedených nástrojích: běžný Java virtuální stroj (JVM), Java PathFinder (JPF) a ConTest. Získané informace z jednotlivých běhů programů jsou posléze porovnány. Vyhodnocení běhů programů slouží k určení časové náročnosti ověření opravy chyby v programu. V každém z uvedených nástrojů trvá běh programu různou dobu a z hlediska verifikace, která je typicky náročná na čás, je porovnání těchto běhů zajímavé. Jednotlivé testy, které zde budou uvedeny je možné opětovně spustit, jsou součástí přiloženého DVD. Pouze nástroj ConTest není volně šířitelný a přiložené DVD tedy obsahuje pouze výsledky těchto testů. Informace o ConTestu je možné získat na [16].

Prvním příkladem je již zmiňovaný program `Bank Account` uvedený v kapitole 5.2.1. Jeho zdrojový kód je také součásti přiloženého DVD. Příklad je založen na vzájemném prokládání jednotlivých vláken, které simulují jednotlivé účty v bance (`Account`). Díky vzájemnému prokládání a přístupu do sdílené proměnné je možné pomocí tohoto příkladu pozorovat vliv nedeterminismu na rychlost provádění programu. Aplikace je vhodným příkladem programu s nutností plánování prokládání jednotlivých instrukcí různých vláken. Pro testové účely bylo v programu vytvořeno deset vláken simulujících jednotlivé účty. Díky nezávislosti vykonávání jednotlivých vláken na sobě vzniká velké množství možných prokládání instrukcí jednotlivých vláken.

Druhým příkladem je velice jednoduchý program `SkipInstruction`. Jedná se o úměle vytvořený program pro testovnání funkčnosti listeneru a pro přeskakování instrukcí Con-

Testu, přidaných do programu pomocí instrumentace. Program byl vytvořen za účelem otestování přeskakování všech možný instrukcí, které mohou být ConTestem do programu přidány. ConTest vkládá instrukce při vytváření/změně proměnných v programu, při volání metod v programu a samozřejmě u volání metod vláken (`start()`,`stop()`,`join()`, atd.). Program tedy obsahuje různé typy proměnných jako integer, double, tyto proměnné mohou být statické nebo nestatické (instance proměnných). Stejně taky i volání metod může nabývat různých typů (statické, ...) a jednotlivé metody mohou mít různé typy parametrů.

Výsledky testů, které jsou uvedené v tabulkách 5.1 a 5.2 byly získány jako průměrné hodnoty časů běhu programů. Každý typ běhu byl spuštěn tisíckrát a z naměřených časů byla vytvořena průmerná hodnota. Běhy programu byly spuštěny na osobním počítači: Intel Centrino Duo, T2250 1.73GHz, 1GB RAM.

5.4.1 Rychlost běhu programu ve použitých nástrojích

Cílem následujícího testování je porovnat rychlost jednotlivých běhů programu v následujících nástrojích. Nejprve byl program spuštěn v obyčejném Java virtuálním stroji (JVM). Průměrná délka běhu v JVM je brána jako referenční. Následně byly programy spuštěny v model checkeru Java PathFinder (JPF), který byl nastaven pro vykonání jednoho náhodného běhu programu. Pak byla vytvořena instrumentace kódu programu pomocí nástroje ConTest a program opět spuštěn v JVM s instrumentací. Posledním testem běhu bylo přehrání instrumentovaného kódu v JPF. Tento test byl spuštěn pouze pro příklad `SkipInstruction`, který obsahuje pouze ty instrumentované instrukce, které je možné v této fázi implementace přeskakovat. Průměrné hodnoty naměřených časů běhu programů jsou uvedeny v tabulce 5.1.

Tabulka 5.1: Běh zvolených příkladů v použitých nástrojích (JVM, JPF, ConTest)

	Bank Account		**SkipInstruction**	
JVM	217 ms	1 ×	50 ms	1 ×
JPF (RandomSearch)	324 ms	1,5 ×	596 ms	11,9 ×
JVM s ConTestem	217 ms	1 ×	58 ms	1,2 ×
JPF s ConTest	—	—	674 ms	13,5 ×

Z naměřených časů v různých nástrojích je vidět, že JPF není navržený pro jednoduchý běh programu. Při simulování běhu programu v JPF dochází ke značnému zpomalení celé aplikace. Oproti tomu běh programu, který byl instrumentován pomocí ConTestu není tolik zpomalen. Je ovšem nutné brát v potaz fakt, že při testovém běhu nebylo instrumentace využito (ConTest nepřidal k běhu programu žádny listener ani nezasahoval do běhu aplikace). Posledním z uvedených časů je přehrání instrumentovaného kódu v JPF. Z dosažených časů je vidět, že při přehrávání programu v JPF s instrumentací již k velkému zpoždění nedochází. To je výhodným faktem pro další vývoj. Je možné verifikovat původní zdrojový kód programu nebo instrumentovaný kód za stejnou cenu.

Z uvedené tabulky také vyplývá závěr, že doba provádění běhu programu v JPF značně záleží na struktuře programu a na typu nedeterminismu. Pokud program obsahuje velké množství nedeterminismu a prokládání instrukcí, které je potřeba řešit, dochází ke značnému

zpomalení běhu programu. U prvního příkladu `Bank Account`, který obsahuje velké množství možného prokládání, dochází k menšímu zpomalení aplikace, z důvodu velkého počtu nezávislých instrukcí.

5.4.2 Rychlost metody Record&Replay trace

Další testování bylo zaměřeno na zjištění rychlosti zaznamenávání a přehrání cesty programu v JPF nebo ConTestu.

Testy byly opět prováděny nad stejnými programy jako předchozí. Referenční rychlost provádění zůstává stejná – jednoduchý běh programu v JVM. Pro testování strategie Record&Replay trace byly použity naipmlementované listenery kapitola 5. První testy běhu programů (v tabulce 5.2 označeny jako SaveChoice v JPF) byly spuštěny v JPF s listenerem `SaveCurChoice`, který během provádění programu zaznamenává volby ChoiceGenerátorů. Druhé testy (Record v JPF I) byly spuštěny v JPF s listenerem `LoadCurChoice` a `RecordTrace`. Pomocí těchto testů tedy docházelo zároveň k přehrávání běhu aplikace pomocí ChoiceGenerátorů a zároveň k záznamu cesty s byte-code instrukcemi. Třetí testy (Record v JPF II) byly spuštěny v JPF pouze s listenerm `RecordTrace`, nedocházelo tedy k navigaci stavovým prostorem pomocí cesty, ale k náhodnému běhu pomocí implementované strategie `RandomSearch`. Ve čtvrtých testech (Record v ConTestu) byla cesta nahrávána pomocí listeneru ConTestu. Poslední dva druhy testů (Replay v JPF I, II) byly opět spuštěny v JPF se strategií pro přehrání cesty `ReplayTraceHeuristic`. U programu `Bank Account` byl nejprve přehráván náhodný běh programu, který vedl ke korektnímu ukončení aplikace. Poslední testy jsou přehráváním chybového běhu programu. U druhého programu `SkipInstruction` nedochází k chybovému běhu programu, a proto nebylo druhé testování přehrávání cesty zapotřebí. Naměřené časy běhu programů jsou uvedeny v tabulce 5.2.

Tabulka 5.2: Record&Replay trace v JPF a ConTestu

	Bank Account		**SkipInstruction**	
SaveChoice v JPF	326 ms	1,6 ×	1 028 ms	20,6 ×
Record v JPF I	434 ms	2 ×	1 258 ms	25,2 ×
Record v JPF II	438 ms	2,1 ×	1 276 ms	25,5 ×
Record v ConTestu	269 ms	1,3 ×	375 ms	7,5 ×
Replay v JPF I	1 135 ms	5,3 ×	648 ms	11,7 ×
Replay v JPF II	1 128 ms	5,2 ×	—	—

Z tabulky jsou opět vidět rozdílné poměry časů zvolených programů. Rozdíl v čase pro simulování přehrávání programu v JPF zůstává i u implementované strategie Record&Replay trace. Z tabulky je patrné další zpomalení, které nastává v důsledku nahrávání běhu programu. Velikost zpomalení nahrávání v JPF je zhruba dvojnásobné oproti náhodnému běhu v JPF. Oproti tomu poměr přehrávání cesty oproti zaznamenávání cesty je u různých programů odlišný. Tento fakt vyplývá z principu strategie přehrávání cesty. Při přehrávání cesty je třeba zkontrolovat, které kroky programu korespondují s uloženou cestou. Pokud program obsahuje velké množství nedeterminismu a velké množství možných jednotlivých

voleb nedeterminismu, je třeba určit korespondující cestu z mnoha možností a tím dochází k většímu zpomalení, než pokud se volí pouze ze dvou nebo tří možností.

Nahrávání cesty programu pomocí ConTestu, který nezanamenává všechny vykonané instrukce byte-codu, ale pouze místa větvení, je rychlejší. Je také cílem zaznamenat běh programu mimo JPF a v JPF provést pouze přehrání cesty.

Kapitola 6

Závěr

Cílem diplomové práce bylo navrhnout a posléze implementovat metodu pro ověření opravy v projektu SHADOWS. Oprava chyb v uvedné části projektu SHADOWS se specializuje na chyby v Java programech. Konkrétně se jedná o chyby, které vznikají v důsledku parelelismu.

Pro kontrolu opravy chyby byla zvolena modifikace formální metody model checkingu. Nejprve bylo nutné nastudovat model checking, jaké má vlastnosti a principy. Po prostudování model checkingu bylo možné navrhnout metodu založenou na modifikaci model checkingu takovým způsobem, aby splňovala zadané požadavky pro ověření opravy.

Navržená metoda pro ověření opravy je založena na vykonávání bounded model checkingu v okolí opravy. Bounded model checking vznikl omezením provádění verifikace pomocí model checkingu do zadané hloubky stavového prostoru. Tato modifikace model checkingu lze pro ověřování opravy chyby využít. Oprava se věnuje chybám vzniklým souběžností, které jsou založeny na způsobu prokládání jednotlivých instrukcí. Tím tedy dojde časem k vykonání všech relevantních instrukcí, které mohou mít vliv na opravu. K vykonání těchto relevatních instrukcí většinou dochází v přijatelném čase a je tedy možné použít bounded model checking.

Nicméně zásadním problémem pro využití bounded model checkingu v okolí chyby je navigace stavovým prostorem do požadovaného stavu systému. Pro navigaci stavovým systémem existují různé strategie, které byly studovány a z nich byla zvolena strategie Record&Replay trace. Podsatou metody je uložení běhu programu, který vede k chybě a následné přehrání tohoto běhu ve zvoleném model checkeru. Uložená cesta tedy slouží k navigaci stavovým prostorem do požadovaného stavu, ze kterého je vykonán bounded model checking pro zverifikování požadovaných vlastností.

Diplomová práce obsahuje jednotlivé části implementace metody Record&Replay trace a strategie pro vykonání bounded model checkingu v JPF.

Implementaci metody Record&Replay trace, lze rozdělit na dvě hlavní části, na implementaci mechanizmu pro záznam cesty a na strategii pro přehrání zaznamenané cesty. Pro implementaci mechanizmu pro záznam cesty bylo hlavním bodem určení jaké informace je třeba zaznamenávát, aby bylo možné zaznamenanou cestu později přehrát. Potom byly postupně navrženy a implementovány listenery model checkeru, které slouží pro záznam cesty, která obsahuje informace podle použitého listeneru. Po naimplementování mechanizmu pro záznam cesty bylo možné přistoupit k implementaci strategie pro přehrání cesty. Přehrání cesty bylo úspěšné u cesty zaznamenané pomocí listenerů Java PathFinderu. Zatím není hotové přehrávání cesty, která byla zaznamenána pomocí nástroje ConTest. Ten v pro-

jektu SHADOWS slouží pro opravu chyby. Tato část implementace pro vytvoření strategie přehrávající cestu zaznamenanou pomocí ConTestu je v rozpracované fázi.

Další práce bude věnována dokončení této strategie. Po dokončení implementace strategie pro přehrání cesty bude možné celou metodu otestovat na reálných datech. A provést bounded model checking pro ověření opravy. Pro vlastní verifikaci byl zvolen model checker Java PathFinder, který umožnuje různé rozšíření. Umožnuje tedy nejen verifikaci vlastností pro které Java PathFinder již obsahuje moduly, ale také doimplementování dalších modulů pro verifikaci konkrétních speciálních vlastností systému. Implementace těchto modulů do Java PathFinderu již není obtížná. Například modul pro detekci „TrueRace“ je již implementován.

Literatura

[1] B. Křena and Z. Letko and R. Tzoref and S. Ur and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07*, pages 54–64. ACM, 2007.

[2] W. Beaton and J. d. Rivieres. Eclipse platform technical overview. Technical report, The Eclipse Foundation, 2006.

[3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking, 2003.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. 15(3–5):485–499, March/April 2003.

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

[7] E.M. Clarke, O. Grumberg, M. Minea, D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer 2*, 3:279–287, 1999.

[8] M. Dwyer et al. Bandera. `http://bandera.projects.cis.ksu.edu`.

[9] P. Mehlitz et al. Java pathfinder. `http://javapathfinder.sourceforge.net`.

[10] Robby et al. Bogor software model checking framework. `http://bogor.projects.cis.ksu.edu`.

[11] Z. Letko et al. Java race detector & healer. `http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/document.html`.

[12] B. Křena et.al. Deliverable D3.2.3: Report on Safety of Program Modifications. Technical Report FP6 IST-035157 SHADOWS: Deliverable D3.2.3, Brno University of Technology and IBM Haifa Research Lab, Aug 2007.

[13] G. Lindstrom,P.C. Mehlitz, and W. Visser. Model checking real time java using java pathfinder. pages 444–456. 2005.

[14] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 85–94, New York, NY, USA, 2007. ACM Press.

[15] G.J. Holzmann, and D. Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[16] IBM Haifa Labs Home. Contest - a tool for testing multi-threaded java applications. `http://www.haifa.il.ibm.com/projects/verification/contest/index.html`.

[17] K. Havelund, and T. Pressburger. Model checking java programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[18] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition).* Prentice Hall PTR, April 1999.

[19] M.Mansouri-Samani et al. *Program Model Checking, A Practitioner's Guide.* Intelligent System Division NASA Ames Research Center, April 2007.

[20] V. Hrubá, and B. Křena and Z. Letko and T. Vojnar. Shadows - deliverable d3.2.3: Report on safety of program modifications. Technical report, 2007.

[21] A. Valmari. The state explosion problem. *In Lectures on Petri Nets I*, 1998.

[22] P. Mehlitz W. Visser. Model checking java programs using java pathfinder. `http://www.visserhome.com/willem/presentations/ase06jpftut.ppt` .

[23] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42, New York, NY, USA, 2005. ACM.