VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE VÝSLEDKŮ ANALÝZY VÍCEVLÁKNOVÝCH APLIKACÍ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE DAVID ŠANTRŮČEK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE VÝSLEDKŮ ANALÝZY VÍCEVLÁKNOVÝCH APLIKACÍ

VISUAL PRESENTATION OF MULTI-THREADED ANALYSIS RESULTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE DAVID ŠANTRŮČEK
AUTHOR

VEDOUCÍ PRÁCE ING. ZDENĚK LETKO PH.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá vytvořením nástroje pro usnadnění vývoje vícevláknových aplikací v jazyce Java. Tento nástroj používá jako vstupní soubory výstupy z nástroje IBM ConTest. Tyto výstupy následně zpracuje a převede do grafické podoby. Nástroj dokáže vykreslit výsledek algoritmu goldilocks a různé vzory přístupů k vybraným synchronizačním primitivům.

Abstract

This thesis deals with creation of tool, which would make development of multithreaded aplications in Java easier. This tool uses IBM ConTest tool output files as input. It will take this files and transform them to visual form. This tool can draw out result of Goldilocks algorithm and various patterns of accesses to choosen synchronization elements.

Klíčová slova

Java, testování, vizualizace, souběžnost, synchronizační primitiva.

Keywords

Java, testing, visualization, concurrency, synchronization elements.

Citace

Šantrůček David: Vizualizace výsledků analýzy vícevláknových aplikací, bakalářská práce, Brno, FIT VUT v Brně, 2015

Vizualizace výsledků analýzy vícevláknových aplikací

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Zdeňka Letka, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
David Šantrůček
20.5.2015

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Zdeňku Letkovi za jeho ochotu, trpělivost a cenné rady a to hlavně při dokončování práce.

Obsah

1 Úvod

V dnešním světě, kde jsou vícejádrové procesory naprosto běžné, se stále více rozmáhají vícevláknové aplikace, které dokáží efektivněji využít výpočetní výkon těchto procesorů. S tím roste potřeba tyto programy a aplikace dokázat bezchybně vytvořit. To se může zdát jako jednoduchý úkol, avšak narozdíl od běžných jednovláknových aplikací je u těch vćevláknových potřeba řešit i otázky synchronizace jednotlivých vláken. Pokud aplikace není dobře napsána, tak může docházet k synchronizačním problémům jako například uváznutí nebo vyhladovění. Navíc tyto problémy se nemusí vyskytnout při každém běhu programu a velice špatně se odhalují. Z toho důvodu se tato práce zabývá vytvořením nástroje, který by pomohl při procesu tvorby a ladění vícevláknových aplikací a především usnadnil správnou synchronizaci jednotlivých vláken a odhalení problémů spojených se samotnou synchronizací. Tento nástroj zachytí běhy jednotlivých vláken a převede je do vizuální podoby, se kterou bude moci programátor jednodušeji pracovat. Ukáže mu například, v jakém pořadí se přistupovalo k synchronizačním primitivům, jako například zámky či sdílené proměnné. A právě vytvoření a vysvětlení takového nástroje je podstatou této práce.

Kapitola 2 obsahuje informace o použitých technologiích. Hned na začátku této kapitoly, konkrétně v podkapitole 2.1, se věnuji programovacímu jazyku Java a možnostem využívání souběžnosti v tomto jazyce. V následující podkapitole 2.2 jsou popsány jednotlivé problémy, které mohou vzniknout při synchronizaci vláken. V další podkapitole 2.3 se zabývám různými způsoby testování vícevláknových aplikací. Podkapitola 2.4 obsahuje informace o nástroji IBM ConTest, jehož výstupy jsou použity pro náš nástroj. V poslední podkapitole 2.5 je popis metrik pokrytí využívaných při testování aplikací. Následující kapitola 3 obsahuje přehled již existujících nástrojů, které umožňují nějakým způsobem graficky zobrazit průběh aplikace s více vlákny. Kapitolou 4 začíná praktická část této práce. Tato kapitola obsahuje podrobně rozepsaný seznam požadavků na výslednou aplikaci. Hned poté následuje kapitola 5, ve které je podrobně rozepsán návrh aplikace. Je zde popsán jak návrh architektury aplikace (podkapitola 5.1), tak i návrh grafického rozhraní (podkapitola 5.2). Další kapitola 6 se už zabývá samotnou implementací aplikace. V podkapitole 6.1 jsou popsány některé konvence, které byli při psaní kódu dodržovány. Podkapitola 6.2 už obsahuje implementační detaily architektury aplikace a v podkapitole 6.3 je možné naleznout implementační detaily o grafickém uživatelském rozhraní. V kapitole 7 je popsán průběh testování aplikace. Jsou zde popsány výsledky jednotlivých testů. Konečně v kapitole 8 je obsažen závěr této práce.

2 Použité technologie

Tato kapitola obsahuje základní informace o programovacím jazyce Java a také seznam chyb, kterých se programátor nejčastěji dopouští při tvorbě vícevláknových aplikací při procesu synchronizace vláken. Také jsou zde popsány různé možnosti testování vícevláknových aplikací. Dále je zde popsán nástroj IBM Contest, za jehož pomoci je vypracována praktická část práce. Nakonec jsou popsány metriky pokrytí pro testování vícevláknových aplikací.

2.1 Java

Java [14] je moderní a populární objektově orientovaný programovací jazyk. Hlavní výhodou tohoto jazyka je přenositelnost, která je dána myšlenkou *Java Virtual Machine* (dále jen JVM) a *Java bytecode*.

Java bytecode je instrukční sada JVM. JVM je sada počítačových programů a datových struktur, která využívá modul virtuálního stroje ke spouštění dalších počítačových programů a skriptů vytvořených v jazyce Java. Hlavním úkolem JVM je zpracovat a spustit java aplikace přeložené do mezikódu, který se skládá z instrukcí *Java bytecode*.

2.1.1 Souběžnost v Javě

Programovací jazyk Java a JVM podporují souběžnost, pro kterou existuje několik struktur, a ty jsou rozděleny do následujících kategorií:

Vlákna (*Threads*): Vlákna zastupují základní pojem dosažení souběžnosti v Javě. Vlákna v Javě existují uvnitř JVM, který se o ně stará. Podle implementace JVM a prostředí, v němž se nacházíme, mohou být vlákna mapována do systémových vláken, lehkých procesů (*lightweight process*), nebo procesů. Pokud je vlákno mapováno do systémových zdrojů, pak není obsluhováno plánovačem JVM, nýbrž systémovým plánovačem.

Java obsahuje několik metod pro obsluhu životního cyklu vláken. Nejdůležitější z nich jsou: `create()` (zavedení a spuštění vlákna), `join()` (čekání na konec jiného vlákna), `sleep()` (čekání předem určený časový úsek) a `intrrupt()` (přeruší vlákno). Je zde také několik zastaralých metod, které by neměli být používány a to např.: `suspend()`, `resume()` a `stop()`, místo nich by měly být používány jiné synchronizační mechanismy a struktury.

Zamykání objektů (*Object locking*): Vzájemné vyloučení je dalším ze základních pojmů potřebných pro dosažení souběžnosti. Java poskytuje podporu pro vzájemné vyloučení založené na monitorech. Každý objekt v Javě má svůj monitor, který může být použit za tímto účelem. Takovéto vzájemné vyloučení se nazývá bezpodmínečné uzamčení (implicit locking). Existují dva způsoby, jak získat zámek. Jeden způsob je použít konstrukci `synchronized(){}`, kde vlákno musí získat

zámek od objektu uvedeného v závorkách. Jiným způsobem je deklarace metody s klíčovým slovem „synchronized“. Pak bude monitor objektu zapouzdřujícího takovou metodu použit pro synchronizaci a každé vlákno jej musí nejprve získat a až poté je daná metoda provedena.

Bezpodmínečné zamykání vyžaduje, aby byl hlídaný blok napsán sekvenčně (tzn., že zámek nemůže být získán v jedné metodě a uvolněn v jiné). Někdy je toto problém, a proto Java 5 představila nový zamykací mechanismus, který je založen na objektech z knihovny `java.util.concurrent.locks`. Tento balík obsahuje pokročilejší zamykací mechanismy, které mohou být snadno rozšířeny pomocí dědičnosti. Použití těchto zámků se nazývá „výslovné zamykání“ (*explicit locking*)

Mezivláknová komunikace (thread notification): V některých případech je nutné synchronizovat vlákna tím, že si mezi sebou pošlou upozornění. Toto lze implementovat pomocí konstrukce „wait and notify“. Vlákno, které vyvolá metodu `wait()`, je pozastaveno, dokud jiné vlákno nezavolá `notify()`, nad tímto čekajícím vláknem. Avšak existuje zde možnost falešného úniku ze stavu čekání, což se dá způsobit přerušením vlákna. Kvůli tomu musí být použita další sdílená proměnná, která kontroluje tento systém upozornění.

Byly vyvinuty i některé další sofistikovanější synchronizační mechanismy používající mezivláknovou komunikaci a od Javy 5 jsou dostupné v balíku `java.util.concurrent` například bariérová synchronizace (*barrier synchronization*), nebo synchronizované fronty (*synchronized queues*).

Nestálé (*volatile*) a neměnné (*immutable*) objekty: Tyto dva vedlejší pojmy pomáhají Javě vypořádat se s některými speciálními problémy se souběžností. Pokud je proměnná deklarována jako nestálá, Java pro ni interně vypne veškerou „kešovací“ funkčnost. To znamená, že každý přístup k takovéto proměnné musí být proveden přímo v hlavní paměti, přesněji, „na haldě“. Tím je zajištěno, že každé vlákno vidí aktuální hodnotu proměnné. Toto se používá pro sdílení stavů mezi vlákny.

Neměnné objekty jsou objekty, které mají veškeré stavové proměnné deklarované jako finální (*final*). Toto zabrání Javě jakkoliv měnit tyto proměnné, tudíž jí zabraní i změnit vnitřní status objektů, které se skládají pouze z proměnných deklarovaných jako finální. Takové objekty jsou pro vlákna bezpečné a mohou být sdíleny bez nutnosti jakékoliv synchronizace.

Téměř všechny výše zmíněné mechanismy mohou být použity špatným způsobem, a to tak, že překladač nezahlásí žádnou chybu a program „zamrzne“ nebo „spadne“ až při běhu. Tímto se zabývá kapitola 2.2 Chyby při používání souběžnosti.

2.1.2 Grafické rozhraní v Javě

Programovací jazyk Java má za sebou už bezmála 20 let vývoje, a proto je více než pochopitelné, že pro ni za tu dobu vzniklo hned několik rámců (*framework*) pro tvorbu grafického uživatelského rozhraní. V této podkapitole budou některé z nich představeny a popsány.

1. **AWT (Abstract Window Toolkit):** Je první z rámců pro tvorbu grafického uživatelského rozhraní. Je výkonný, ale nedokáže pracovat s vyspělejšími komponentami. Jeho hlavní výhodou je, že je velice dobře otestovaný a funkční pro malé aplikace s jednoduchým grafickým rozhraním. Pro složitější aplikace je však vhodné zvolit jiný rámec. Navíc jeho hlavní nevýhodou je jeho závislost na platformě. To znamená, že na každé platformě vykazuje odlišné chování, což popírá jeden z hlavních principů Javy, a to nezávislost na platformě.[7]
2. **Swing:** Rámec Swing je založený na AWT a vzniknul jako jeho náhrada. Na rozdíl od AWT jsou všechny jeho komponenty napsané v Javě a tím pádem už není závislý na platformě. Oproti AWT také nabízí lepší a flexibilnější komponenty, jako např. stromy, seznamy nebo tabulky. Momentálně je sice nahrazován výkonnějším rámcem JavaFX (viz níže), avšak pro středně náročné aplikace je Swing stále vynikající volbou.[8]
3. **SWT (Standard Widget Toolkit):** SWT je alternativa k AWT a Swingu, která s těmito dvěma rámci nijak nesouvisí. K vykreslování jednotlivých grafických položek využívá nativní knihovny operačního systému. To znamená, že programy používající SWT jsou sice multiplatformní, ale pro každou platformu mají unikátní implementaci.[9]
4. **SwingX:** je založené na klasickém Swingu. Prakticky je to Swing rozšířený o celou řadu vyspělých komponent. Tento rámec je však ještě stále ve vývoji a není vytvářen přílišný tlak na dokončení tohoto rámce.[10]
5. **JavaFX:** Účelem JavaFX je stát se náhradou pro Swing. Jedná se o moderní rámec obsahující mnoho komponent s možností jejich použití. Je určen především pro velké moderní a složité aplikace.[11]

2.2 Chyby při používání souběžnosti

Při využívání souběžnosti se plánovače (*scheduler*) chovají nedeterministicky. To znamená, že pro každý běh se program využívající souběžnost může chovat odlišně. Proto je třeba, aby byla pro program zajištěna vnější synchronizace programátorem, pokud spolu mají vlákna komunikovat a spolupracovat. Pokud toto není zajištěno, pak se může s vysokou pravděpodobností vyskytnout chyba v souběžnosti (*concurrent error*). Nedeterminismus způsobí, že se chyba nemusí vyskytnout při každém běhu programu. Často je to právě naopak, a proto je velice obtížné tyto chyby odhalit a odstranit. Následuje seznam nejčastějších chyb, kterých se programátoři dopouští:

- **Neatomická operace je považována za atomickou:** Operace se zdá být atomická (bude vykonána bez možnosti jejího proložení jinou operací), ale ve skutečnosti se skládá z několika nechráněných operací. Například: x++; vypadá jako atomická operace, ale ve skutečnosti se skládá ze tří různých operací, které může plánovač chybně proložit. [1]

- **Dvoustupňový přístup:** Posloupnost operací musí být chráněná dohromady, ale programátor chybně chrání každou zvlášť. Například: vyhodnocení podmínky a provedení nějaké operace. Chyba nastane tehdy, pokud je hodnota po vyhodnocení podmínky změněna jiným vláknem a do dalších operací vstupuje už chybně změněná hodnota. [1]
- **Špatný zámek nebo žádný zámek:** Část kódu je chráněna zámkem, ale ostatní vlákna neobdrží jeho stejnou instanci při vykonávání této části a je možné narušení mezi vlákny. Například: zámek *z* je uzamknut vždy, když se přistupuje k proměnné *p*. Pokud některé vlákno nezamkne *z* před přístupem k *p*, pak není synchronizováno s ostatními vlákny. [1]
- **Dvojité zamykání:** Když je inicializován objekt, jsou inicializovány i lokální kopie objektových polí. Navíc ne všechna objektová pole musí být nutně zapsána „na haldě“ (*heap*). Například: když statický ukazatel na objekt „uteče“ svému konstruktoru, jakékoliv jiné vlákno ho může použít k přístupu k částečně zinicializovanému objektu. [1]
- **Předpoklad, že nikdy nedojde k prokládání instrukcí:** Programátor špatně usuzuje, že některé prokládání instrukcí není možné. Je zde několik příkladů takových chyb. Např.: Programátor vloží příkaz `sleep()` do vlákna *v*, které má být pomalejší než ostatní vlákna, avšak vlákno *v* může být rychlejší dokonce i s příkazem `sleep()`.[1]
- **Blokování kritické sekce:** U vlákna vstupujícího do kritické sekce se předpokládá, že z ní taky časem vystoupí. Například: vlákno provádějící nějakou blokující vstup/výstupní operaci ji nemusí nikdy dokončit a tím pádem neuvolní ani žádné zámky, které právě vlastní. [1]
- **Osiřelé vlákno:** Pokud není hlavní vlákno ukončeno běžným způsobem, ostatní vlákna mohou pokračovat v běhu. Např.: fronta může být použita k synchronizaci tak, že hlavní vlákno do ní vkládá zprávy a ostatní vlákna je z ní čtou. Pokud je hlavní vlákno ukončeno, aniž by o tom informovalo ostatní vlákna, mohou pak běžet do „nekonečna“ a blokovat ukončení vykonávání programu. [1]
- **Uváznutí (*Deadlock*):** K uváznutí dojde, když se objeví cyklus v grafu získávání zdrojů. Např.: pokud vlákna *v1* a *v2* potřebují oba zámky *z1* a *z2*, a vlákno *v1* zamkne *z1* a čeká na *z2*, a vlákno *v2* zamkne *z2* a čeká na *z1*, pak žádné z těchto vláken nedojde k žádnému výsledku. [1]
- **Závod (Data race):** K této chybě dojde, pokud k jedné sdílené proměnné přistupují dvě nebo více vláken současně, alespoň jeden z těchto přístupů je zápis a vlákna nepoužívají žádné zámky pro zajištění výlučného přístupu. Pokud jsou splněny tyto tři podmínky, je pořadí přístupu nedeterministické a výsledek se může lišit s každým během programu.[2]

2.3 Testování vícevláknových aplikací

Stěžejním krokem při testování vícevláknových aplikací je otestování co nejvyššího počtu různých proložení instrukcí, kolik je jen možné. Pokročilé metody pro testování vícevláknových aplikací,

které jsou popsány níže, umožní otestovat velké množství způsobů proložení i s vynaložením menšího úsilí programátora. Techniky jsou založené na opakovaném vykonávání stejného testu se stejnými vstupy a detekováním, zda během vykonávání došlo k chybě. Při každém opakování testu se metody snaží ovlivnit plánovač instrukcí takovým způsobem, aby došlo k proložení instrukcí, které při předchozím vykonávání testu ještě nenastalo. Množství odlišných proložení se zvyšuje buď vložením takzvaného šumu (*noise*) do testu nebo vynucením systematického plánování (*deterministic scheduling*).

2.3.1 Systematické testování

Systematické testování [15] ovládá plánování instrukcí vláken při vykonávání testu a systematicky prozkoumává všechny možnosti proložení. Aby se vyhnuly nechtěným změnám vykonávajícího prostředí, tak se moderní nástroje pro systematické testování soustředí na aplikační rozhraní, které poskytuje testovaným programům funkčnost synchronizace. Volání synchronizačních metod prostředí, pod kterým je aplikace spuštěna, je zachyceno a předáno systematickému plánovači (*deterministic scheduler*). Tento plánovač je schopný zastavit vlákna, která by neměla pokračovat. Systémový plánovač poté plánuje jenom vlákna, kterým bylo povoleno pokračovat systematickým plánovačem. Systematický plánovač nechá pokračovat přerušená vlákna, až když je to nutné.

Běžný způsob, jakým tyto techniky pracují je následující: Nejdříve je testovaný program vykonám bez jakéhokoliv zásahu od systematického plánovače a vykonaná synchronizace je zaznamenána. Následně je použit prohledávací algoritmus, aby odvodil, jaký scénář by se měl naplánovat a následně vynutit systematickým plánovačem během příštího vykonávání testu. Prohledávací algoritmus je nejsložitější část všech nástrojů pro systematické testování, protože vybraný scénář musí být v testovaném programu dosažitelný a měl by vést k synchronizačnímu scénáři, který není ekvivalentní s žádným již proběhlým synchronizačním scénářem. Prohledávací algoritmus je také schopen odhalit dokončení procesu testovaní, neboli situace, kdy už byly otestovány všechny dosažitelné scénáře. V takové situaci je ověřovací proces dokončen.

Vždy když je odvozený následující scénář prohledávacím algoritmem, je znovu vykonán testovaný program. Nejdříve je použit systematický plánovač k přehrání vybraného scénáře. V nějakém bodě vykoná systematický plánovač volbu, která ještě nebyla vybrána v žádném z předchozích průběhů a dostane tak vykonávání do ještě neprozkoumaného synchronizačního scénáře. V tomto bodě povolí systematický plánovač všem vláknům pokračovat v běhu. Zbytek průběhu je zaznamenán a uložen mezi ostatní již proběhlé scénáře. Následně proběhne prohledávací algoritmus znovu. Pro systematické testování existuje spousta nástrojů. Těmi významnějšími z nich jsou například CHESS[3] a Maple[4].

2.3.2 Testování pomocí vkládání šumu

Techniky pro testování pomocí vkládání šumu [19] vkládají buď náhodně anebo na základě nějakého předpisu šum do vykonávaného testu. Šum způsobí prodlevy v průběhu vybraného vlákna a tím dá ostatním vláknům šanci běžet déle.

Výhodou přístupu vkládání šumu je, že tato metoda nevyžaduje žádné úpravy prostředí a ani žádné úpravy testu. Místo toho je testovaný program automaticky instrumentován. Instrumentace vloží do cílové aplikace úseky vlastního kódu. Program může být instrumentován na více úrovních: zdrojový kód, mezikód jako například j*ava bytecode*, nebo binární úroveň. Instrumentace může být provedena buď před provedením kódu, nebo až za běhu aplikace. V případě technik pro vkládání šumu instrumentace obvykle vkládá volání metody pro vytváření šumu (*noise maker*) do kódu aplikace. Vlákna vykonávající upravený kód poté vstoupí do metody pro generování šumu a ta rozhodne, zda bude šum generován. Důležité je, že už jenom instrumentace samotná způsobí šum, protože vlákna musejí vykonávat instrukce navíc.

Metody pro vkládání šumu se musí vypořádat se dvěma základními problémy, a to kdy vkládat šum (*noise placement problem*) a jak ovlivnit plánovač (*noise seeding problem*). První problém zahrnuje označení míst, kam vložit instrumentační kód a rozhodnutí, zda způsobit šum. Instrumentovat kód, který neovlivňuje synchronizaci nebo komunikaci mezi vlákny by nemělo žádný smysl. Proto generátory šumu vybírají pouze místa, která mají nějakou spojitost se souběžností, jako například zamykání zámku a přístupy ke sdílené paměti. Rozhodnutí, zda způsobit šum na vybraném místě se buď rozhodne na základě náhody, nebo na základě nějakých dalších znalostí, například odhalený vzor v kódu, informace o pokrytí, nebo informace získané za běhu.

Způsob, jakým může být ovlivněn plánovač, závisí na možnostech přístupných v prostředí, ve kterém je aplikace testována, avšak základní principy zůstávají stejné. Tato práce se zaměřuje především na Javu a detaily ohledně ovlivňování plánovače jsou popsány v kapitole 2.4.

Významným nástrojem pro testování pomocí vkládání šumu je IBM Concurrency Testing tool (zkráceně IBM ConTest). Tento nástroj je využíván i v této práci a blíže je popsán v kapitole 2.4.

2.3.3 Dynamické testování

Dynamická analýza je velice populární způsob verifikace vícevláknových aplikací, protože analyzovat pouze jeden průběh testu je o dost jednodušší než analyzovat všechny možné průběhy vláken. Obrovské množství všech možných proložení je v tomto případě sníženo zkoumaným průběhem testu. Dynamická analýza může být samozřejmě provedena opakovaně. Techniky pro vkládání šumu a systematické testování mohou být použity pro zvýšení množství různých proložení zkoumaných dynamickou analýzou.

Dynamická analýza může být, podobně jako vkládání šumu, založena na instrumentaci. V tomto případě jsou volání metod, které provádějí dynamickou analýzu, vloženy na vybraná místa

v kódu. Doplňující informace (například identifikátor proměnné, ke které se právě vlákno chystá přistoupit) mohou být předány jako parametry metod dynamické analýzy. Vykonávání těchto metod je však náročné na systémové zdroje. Tím dynamická analýza ovlivňuje výkon testované aplikace. Co se vkládání šumu týče, tak dynamická analýza představuje jeden ze zdrojů šumu a tím ovlivňuje plánování proložení pro instrukce vláken.

Každoročně je publikována spousta nových přístupů, frameworků nebo nástrojů pro dynamickou analýzu vícevláknových aplikací. V této části se ovšem zaměřím pouze na algoritmy použité v této práci a to jsou *Lockset-based* [5] algoritmy, *Happens-before-based* [20] algoritmy a jejich kombinace.

Techniky založené na locksetech staví na myšlence, že každý přístup ke sdílené proměnné by měl být hlídaný zámkem. Lockset je definován, jako soubor zámků které hlídají danou proměnnou. Pokud je lockset přidružený k dané proměnné neprázdný, pak existuje alespoň jeden zámek, který hlídá veškeré přístupy k této sdílené proměnné a tím pádem neexistuje možnost, že by docházelo k přístupu k této proměnné zároveň ze dvou či více vláken současně. Prvním algoritmem, který používal myšlenku locksetů byl *Eraser* [5].

Happens-before-based techniky těží ze vztahu *happens-before* (happens-before relation), který je definován jako nejméně přesný částečný řád, který zahrnuje všechny dvojce kauzálně seřazených událostí. Detektory staví (nebo aproximují) happens-before vztah mezi přístupy ke sdíleným proměnným a zajišťují, aby žádné dva přístupy (z nichž by alespoň jeden byl zápis) nemohly nastat současně, aniž by mezi nimi byl happens-before vztah.

Výhodou výše zmíněných algoritmů je jejich přesnost. Avšak jejich velká náročnost na systémové zdroje inspirovala mnoho výzkumů, aby našli kombinaci mezi happens-before-based a lockset-based algoritmy. Tyto kombinace se často nazývají hybridní algoritmy (*hybrid algorithms*). Jedním z nepokročilejších hybridních algoritmů je Goldilocks [12]. Hlavvní myšlenkou tohoto algoritmu je, že locksety mohu obsahovat nejen zámky, ale i nestálé proměnné a hlavně vlákna. Výskyt vlákna *v* v locksetu sdílené proměnné znamená, že *v* je správně synchronizované pro použití této proměnné. Informace o vláknech synchronizovaných pro použití daných proměnných je pak použita k vytvoření tranzitivního uzávěru happens-before vztahu pomocí locksetů. Výhodou Goldilocku je, že umožňuje locksetům růst během výpočtů, jakmile je vytvořen vztah happens-before mezi operacemi proměnné.

2.4 IBM Concurrency testing tool

Concurrency Testing tool (dále jen ConTest) je automatizovaný nástroj určený na testování vícevláknových aplikací. Hlavním úkolem nástroje ConTest je odhalit chyby v programu, spojené se synchronízací jednotlivých vláken, s minimálním dopadem na výkon samotného programu. Průběh testování se skládá ze tří kroků a to instrumentace, vkládání šumu a nakonec odchycení výsledku.

Proces instrumentace pracuje na úrovni *Java bytecode*. Při tomto procesu se do kódu vloží volání metod ConTestu. Následně je ConTest zavolám při vykonávání instrumentovaného programu. Toto řešení umožňuje ConTestu sledovat a ovlivňovat chování testovaného programu. Místa, odkud je ConTest volán jsou vybrána s ohledem na souběžné zpracování. Taková místa se pak nazývají instrumentační body. Pod tímto bodem si můžeme představit například sdílené proměnné, pole, volání nebo opouštění metody nebo všechny možné operace ovlivňující běh vlákna. Princip techniky vkládání šumu je podrobně popsán v kapitole 2.3.2.

Odchycení výsledku probíhá za pomoci externích zásuvných modulů. Tyto zásuvné moduly reaguji na různé události vyvolané instrukcemi vloženými instrumentací. Jakmile dojde k určité události, provede se kód zásuvného modulu (skupiny modulů), který na tuto událost čeká. Některé instrukce mohou vyvolat dvě události, a to událost před, která je vyvolána ještě před provedením instrukce, a událost po, která je provedena až po provedení dané instrukce. Toto chování umožňuje zásuvným modulům mnohem větší kontrolu na testovaném programu.

2.5 Metriky pokrytí

Při testování potřebuje programátor nějaké měřítko, které by mu ukázalo, jak dobře byl program otestován, jak dobrý test je, nebo jak moc je potřebné další testování. K tomuto účelu je použit koncept metriky pokrytí (*coverage metrics*) [13]. Metriky pokrytí jsou založeny na úkolech pokrytí (coverage tasks), které reprezentují odlišné jevy, jejichž výskyt v chování testovaného programu je považován za zajímavý. Metriky pokrytí lze rozdělit na tři základní skupiny. Jsou to metriky založené na chybách, metriky založené na testech a metriky založené na programovém kódu.

Metriky založené na chybách vycházejí z počtu nalezených chyb při testech, a vzhledem k tomu, že samotný počet chyb není příliš vypovídající údaj, tak vycházejí i ze závažnosti těchto chyb. Takovou metrikou je například efektivnost testu, která se vypočítá jako počet chyb nalezených v testu děleno počet chyb nalezených celkem. Výsledek vyjadřuje, kolik procent chyb bylo pomocí tohoto testu nalezeno. Tato metrika slouží především pro porovnání jednotlivých testů.

Metriky založené na testech určují, kolika procenty testů už aplikace prošla. Pro určení výsledku z metrik založených na testech je nejprve nutné znát dopředu počet všech připravených testů. Z nich už pak lze snadno dopočítat, jak je aplikace pokročila s testováním. Aplikace projde všemi připravenými testy za předpokladu, že neobsahuje žádné chyby, nebo testy dostatečně nepokrývají všechny funkce aplikace (což je v praxi častější případ). Jako jednotka počtu testů se používá testovací případ.

Metriky založené na programovém kódu jsou také často nazývány jako metriky pokrytí kódu (code coverage). Tyto metriky měří, jak velké množství kódu (počet řádků, počet vykonaných příkazů, počet prozkoumaných větví programu, atd.) bylo vykonáno během testu. Vysoká míra pokrytí kódu je nutná podmínka pro spolehlivé ověření testu. Avšak fakt, že test prošel přes všechny

řádky kódu, ještě neznamená, že byly odhaleny všechny možné způsobům chování programu. Jasným příkladem toho jsou aplikace využívající souběžnost. Existující metriky založené na souběžnosti mohou být rozděleny do dvou skupin, a to *interleaving-based* metriky a *scheduling-based* metriky. Interleaving-based metriky jednoduše sledují pořadí, v jakém se dvojce (nebo skupina) událostí objeví, tedy sledují, kolik různých proložení se objevilo. Scheduling based metriky také sledují, jaké další akce navíc proběhly mezi vlákny. Mimo to ještě existuje metrika zvaná pokrytí synchronizace (*synchronization coverage*), kterou není možno zařadit ani do jedné z těchto skupin. Tato metrika se soustředí na plnění scénářů synchronizačních konstrukcí.

3 Existující vizualizační nástroje

Vzhledem k populárnosti tohoto tématu již vzniklo několik nástrojů, které se zabývají problematikou vizualizace vláken ve vícevláknových aplikacích. Některé, z těchto již existujících nástrojů, jsou uvedeny a popsány v této podkapitole.

Prvním z těchto nástrojů je Microsoft Visual Studio 2010 [18]. Toto vývojové prostředí umožňuje zobrazit spoustu užitečných informací jako zátěž jednotlivých vláken na jádra procesoru a to i v porovnání s ostatními procesy, nebo časové diagramy zobrazující běhy jednotlivých vláken. Ovšem velkou nevýhodou je, že tento nástroj je spustitelný pouze pod operačním systémem Windows, což z něj pro programátory pracující pod jinými systémy, jako třeba Linux, dělá nepoužitelnou aplikaci.

Dalším podobným nástrojem je Pajé [17]. Tento samostatný nástroj dokáže analyzovat vícevláknovou aplikaci a následně zobrazit časové diagramy pro jednotlivá vlákna. V těchto diagramech jsou zobrazeny začátky i konce jednotlivých vláken a dokáže odhalit i některé synchronizační problémy, jako třeba uváznutí. Je vytvořený tak, aby zvládal i velká množství vláken najednou. Problém však je, že tato aplikace vznikla již v roce 2000 a od té doby se nedočkala updatu a dle dostupných informací funguje opět pouze pod Windows.

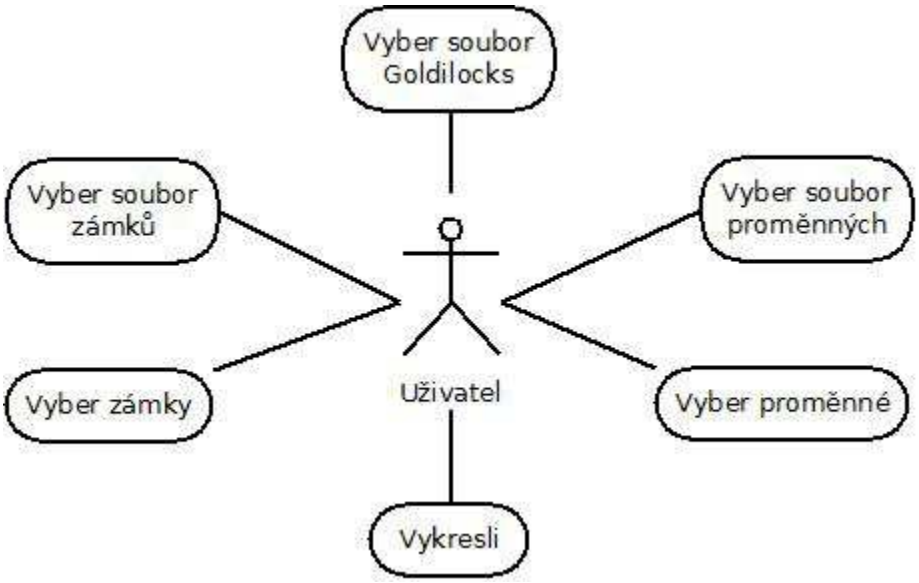
Posledním ze zde uvedených nástrojů je VisualSync [16]. Tento nástroj slouží hlavně jako pomůcka pro výuku a pochopení problematiky synchronizace vláken při tvorbě vícevláknových aplikací. Dokáže spustit aplikaci napsanou v Javě a následně krok po kroku ukazovat uživateli, co právě vykonávají jednotlivá vlákna a také stav jednotlivých synchronizačním primitiv, jako jsou semafory. Tento nástroj však není veřejně dostupný a nedá se o něm zjistit o moc více informací.

4 Specifikace požadavků na aplikaci

Aplikace, která vznikne v rámci této bakalářské práce, bude sloužit k usnadnění procesu tvorby vícevláknových aplikací. Docílí toho tak, že poskytne uživateli náhled na vnitřní komunikaci vláken pomocí grafického znázornění sdílených proměnných a synchronizačních primitiv. To by mělo uživateli pomoci odhalit chyby v synchronizaci, představené v kapitole 2.2, a zároveň uživateli umožnit lépe pochopit, jak k dané chybě vůbec došlo.

Uživatelem této aplikace se rozumí programátor, který vyvíjí vícevláknové aplikace v jazyce Java, o kterém pojednává kapitola 2.1. U tohoto uživatele se předpokládají alespoň základní znalosti práce s jazykem Java a vývojovým prostředím, ve kterém pracuje. Pro plné využití aplikace by také měl být schopný používat nástroj IBM Concurrency Testing tool představený v kapitole 2.4.

Aplikace by měla být schopná vykreslit graf přístupu ke zvolené proměnné, graf přístupu ke zvolenému synchronizačnímu primitivu a graf znázorňující výsledky algoritmu Goldilocks, který je popsán v kapitole 2.3.3. Vstupními soubory aplikace budou textové výstupní soubory z nástroje IBM ConTest. Uživatel bude mít možnost si zvolit jednotlivé vstupní soubory a následně jednotlivé položky uvedené v těchto souborech (např. konkrétní zámky nebo sdílené proměnné). Možnosti použití aplikace jsou podrobněji znázorněny v diagramu případu užití na obrázku 4.1.



Obrázek 4.1 - Diagram případů užití

Následuje popis jednotlivých případů užití:

- Vyber soubor Goldilocks. Aplikace umožní uživateli vybrat si soubor, který obsahuje textový výstup z testu algoritmu Goldilock nástroje IBM ConTest. Soubor půjde vybrat buď přímým zadáním cesty k souboru, nebo vybráním souboru pomocí dialogového okna.

- **Vyber soubor proměnných.** Aplikace umožní uživateli vybrat si soubor, který obsahuje výpis práce se sdílenými proměnnými z nástroje IBM ConTest. Soubor půjde vybrat buď přímým zadáním cesty k souboru, nebo vybráním souboru pomocí dialogového okna.
- **Vyber soubor zámků.** Aplikace umožní uživateli vybrat si soubor, který obsahuje výpis práce se zámky z nástroje IBM ConTest. Soubor půjde vybrat buď přímým zadáním cesty k souboru, nebo vybráním souboru pomocí dialogového okna.
- **Vyber proměnné.** Aplikace umožní uživateli vybrat si, ke kterým proměnným chce nechat graf přístupů vykreslit. Uživateli bude nabídnut seznam proměnných, který aplikace získá z načteného souboru.
- **Vyber zámky.** Aplikace umožní uživateli vybrat si, ke kterým zámkům chce nechat graf přístupů vykreslit. Uživateli bude nabídnut seznam proměnných, který aplikace získá z načteného souboru.
- **Vykresli.** Aplikace umožní uživateli vykreslit zvolené položky. Princip získání dat pro vykreslení je popsán v kapitole 5.1.

5 Návrh aplikace pro vizualizaci výsledků analýzy vícevláknových aplikací

V této kapitole je návrh samotného řešení vlastní aplikace pro vizualizaci výsledků analýzy vícevláknových aplikací, která je předmětem této bakalářské práce. V kapitole 6.1 je popsán návrh architektury aplikace na vysokém stupni abstrakce. Jsou zde znázorněna řešení dílčích problémů. Popis není zaměřen na implementační detaily, ale na obecnou funkci jednotlivých částí aplikace. Dále v kapitole 6.2 je i návrh grafického uživatelského rozhraní.

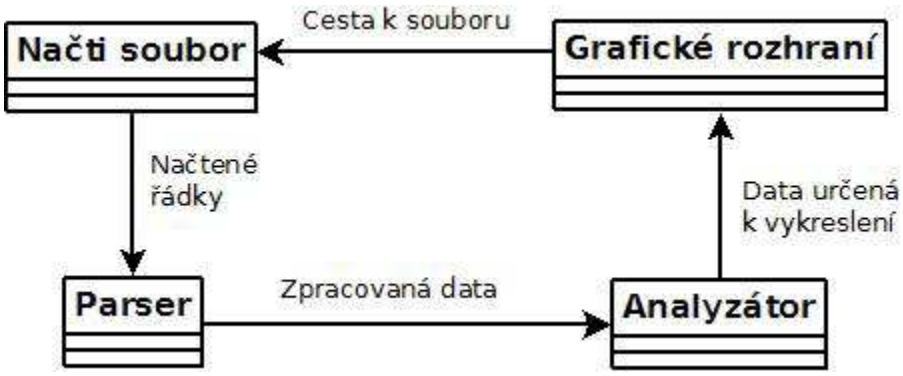
5.1 Návrh architektury

Z požadavků specifikovaných v kapitole 4 je patrné, že aplikace musí být schopná načítat soubory v textovém formátu a dále je umět zpracovat. Dále musí mít grafické rozhraní, které bude zajišťovat veškeré vstupy a zároveň výstupy aplikace. Proto byl návrh aplikace rozdělen do několika samostatně funkčních celků. Jsou to:

1. **Načítání souborů.** Tato část je schopná načíst libovolný textový soubor po řádcích a uložit načtené řádky do vhodné datové struktury (pole, seznam). Stará se o korektní otevření a uzavření souboru, aby nedocházelo ke zbytečným chybám. Načtená data zapouzdřuje a umožňuje k nim přístup pouze pomocí k tomu určených funkcí. Dále je schopna se vypořádat s chybami spojenými s načítáním souboru (např. soubor neexistuje). Ostatním částem umožňuje buď číst načtený soubor po řádcích, tento řádek upravit nebo dokonce odstranit, nebo převzít celou datovou strukturu obsahující načtené řádky.
2. **Parser.** Vstupem do části je buď pole, nebo seznam, obsahující jednotlivé řádky souboru a typ zpracovávaného souboru. Typem souboru je myšleno, zda se jedná o soubor obsahující informace o zámcích, sdílených proměnných nebo algoritmu Goldilocks. Parser tímto polem projde a získá z něj všechna pro uživatele zajímavá data. Tyto data poté uloží do datových struktur k tomu určených (budou popsány dále). Parser bude dále schopný odhalit chyby ve formátu načítaného souboru.
3. **Analyzátor.** Toto je nejdůležitější část aplikace, co se zpracování dat týče. Jejím vstupem jsou datové struktury obsahující data načtená parserem. Tato data jsou poté prozkoumána, jsou v nich nalezeny opakující se vzory, a tyto vzory jsou poté uloženy do vhodné datové struktury a jsou zároveň výstupem této části.

4. **Grafické rozhraní.** Grafickému rozhraní se věnuje celá kapitola 6.2, proto zde o něj nejsou žádné informace, avšak v rámci ucelenosti seznamu je zde uvedeno.

Tyto funkční celky jsou vzájemně provázány a komunikují spolu. Jejich propojení je znázorněno na obrázku 5.1.



5.1 Schéma propojení částí aplikace

Architektura struktur pro uložení dat záleží na typu a počtu dat, které chci ukládat. Proto je u každé datové struktury uveden i příklad vstupních dat a u každé ukládané položky je i zdůvodněno, proč je uchovávána. Zde bych ještě zdůraznil, že vstupní soubory pro tuto aplikaci jsou výstupem nástroje IBM ConTest popsaném v kapitole 2.4 a zásuvného modulu pro sběr Goldilocks metriky[12].

1. **Struktura pro uložení informací o zámcích.** Příklad vstupního souboru je na obrázku 5.2. Na příkladu je vidět, že soubor obsahuje identifikaci vlákna, které právě komunikuje se zámkem. Dále je tu samozřejmě i identifikátor zámku a informace o tom, zda je zámek zamykán nebo odemykán. Nakonec je zde i informace, ve kterém zdrojovém soubor se tato komunikace nachází. Z těchto dat je pro uživatele nejzajímavější a tím i pro aplikaci nejdůležitější identifikátor zámku (v příkladu je to demo.Airlines$Flight@1a2cc7) a informace o tom, zda je zámek odemykán nebo zamykán. Označení zámku za zaviná čem uživateli nic neřekne, proto může být pro další zpracování vypuštěno. Informace o tom, které vlákno zabírá zámek také pro uživatele není příliš podstatná, vzhledem k tomu že se jedná o název generovaný nástrojem IBM ConTest, avšak aplikace pro pozdější analýzu tuto informaci vyžaduje. Poslední údaj o tom, kde k práci se zámkem došlo má už jen informační charakter a může být pro uživatele zajímavá.

   Z výše uvedeného rozboru tedy vyplývá, že z každého řádku výpisu bude nutné uložit čtyři údaje. Jsou to identifikátor vlákna, informace o tom zda je zámek odemčen nebo zamčen, identifikátor zámku a umístění zámku ve zdrojovém kódu. Z toho plyne, že datová struktura pro uložení informací o zámcích musí být schopna uchovat velké množství řádků, a zároveň mít možnost pro každý řádek ukládat vlastní informace. Dále musí být možnost k těmto informacím jednoduše přistupovat a měla by zde být i nějaká možnost rychlého procházení uložených dat.

```
Thread-13(java.lang.Thread@10014f0) has taken lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 98
Thread-13(java.lang.Thread@10014f0) has released lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 102
Thread-31(java.lang.Thread@e5376a) has taken lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 98
Thread-31(java.lang.Thread@e5376a) has released lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 102
Thread-27(java.lang.Thread@27f394) has taken lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 98
Thread-27(java.lang.Thread@27f394) has released lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 102
Thread-29(java.lang.Thread@a87e7b) has taken lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 98
Thread-29(java.lang.Thread@a87e7b) has released lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 102
Thread-25(java.lang.Thread@18c668c) has taken lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 98
Thread-25(java.lang.Thread@18c668c) has released lock object: demo.Airlines$Flight@1a2cc7 at /demo/Airlines.java 102
```

5.2 Příklad souboru obsahujícího informace o zámcích

2. **Struktura pro uložení informací o sdílených proměnných.** Příklad vstupního souboru je na obrázku 5.3. Řádky vstupního souboru jsou bohužel příliš dlouhé, a proto jsou v příkladu z důvodu úplnosti zalomené. Navzdory přehlednosti je na příkladu vidět, že každý řádek obsahuje identifikátor sdílené proměnné a informace o páru instrukcí, které se týkali dané proměnné a proběhli současně nebo v těsném sledu. O každé instrukci je zde informace o tom, odkud byla vyvolána a zda se jedná o zápisovou instrukci (příznak "bw") nebo čtecí instrukci (příznak "br"). Pro uživatele a zároveň pro aplikaci je samozřejmě zajímavý identifikátor sdílené proměnné, avšak nejzajímavější informace zde je příznak určující, jestli se jedná o zápisovou nebo čtecí instrukci. Pokud by totiž došlo k situaci, že jsou vedle sebe dvě zápisové instrukce (poslední řádek příkladu), pak dochází k chybě v synchronizaci a v proměnné může být (a bude) chybná hodnota. Informace o původu instrukcí nemají pro aplikaci žádný význam, avšak pro uživatele mohou být zajímavé.

   Z tohoto rozboru tedy plyne, že z každého řádku výpisu bude nutné získat a uložit pět různých údajů. Jsou to identifikátor sdílené proměnné, příznak druhu první i druhé instrukce a původ první i druhé instrukce. Proto musí být datová struktura pro uložení dat o sdílených proměnných schopna pojmout velké množství řádků a být schopna pro každý řádek ještě uložit navíc rozšiřující údaje. Stejně jako u struktury pro uložení informací o zámcích musí existovat možnost k datům jednoduše přistupovat.

```
demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java Flight(String,int) bw 88 1:/home/
david/BP/rv09_demo/demo Airlines.java bookTicket() br 105 1
demo.Airlines.Flight@1.correctSoldSeats:/home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 101 1:/
home/david/BP/rv09_demo/demo Airlines.java bookTicket() br 99 2
demo.Airlines.numLoop@static:/home/david/BP/rv09_demo/demo Airlines.java main(String[]) bw 39 1:/home/david/
BP/rv09_demo/demo Airlines.java run() br 146 1
demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1:/home/
david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1
```

5.3 Příklad souboru obsahujícího informace o sdílených proměnných

3. **Struktura pro uložení výsledku algoritmu Goldilocks.** Příklad vstupního souboru je na obrázku 5.4. Už na první pohled je zřejmé, že výstupní soubor algoritmu Goldilocks je velmi podobný výpisu sdílených proměnných. Opět je zde identifikátor sdílené proměnné a dvě instrukce. Avšak na rozdíl od výpisu sdílených proměnných jsou zde vypsány pouze proměnné a instrukce, kde už přímo dochází k nějaké synchronizační chybě. Oproti výpisu sdílených proměnných je zde také navíc informace o tom, k jaké chybě dochází (např. v

ukázce je to Data race, neboli závod). Tato informace není pro aplikaci nijak důležitá, ale má poměrně vysokou informační hodnotu pro uživatele.

Z každého řádku výpisu tedy bude nutné získat stejné údaje jako u sdílených proměnných, a to identifikátor sdílené proměnné, příznak druhu první i druhé instrukce, původ první i druhé instrukce a k nim navíc ještě identifikátor synchronizační chyby. Na rozdíl od struktury pro uložení informací o sdílených proměnných zde však není nutnost ukládat větší množství řádků, neboť vstupní soubor obsahuje pouze seznam druhů chybných přístupů. Opět ale platí, že pro každý řádek musí jít uložit všechny potřebné údaje.

```
Data race: demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java bookTicket() br 105 1:/
home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1
Data race: demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1:/
home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1
Data race: demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1:/
home/david/BP/rv09_demo/demo Airlines.java bookTicket() br 105 1
Data race: demo.Airlines.Flight@1.soldSeats:/home/david/BP/rv09_demo/demo Airlines.java getStatus() br 114 1:/
home/david/BP/rv09_demo/demo Airlines.java bookTicket() bw 105 1
```

5.4 Příklad souboru obsahujícího výsledky algoritmu Goldilocks

5.2 Návrh grafického uživatelského rozhraní

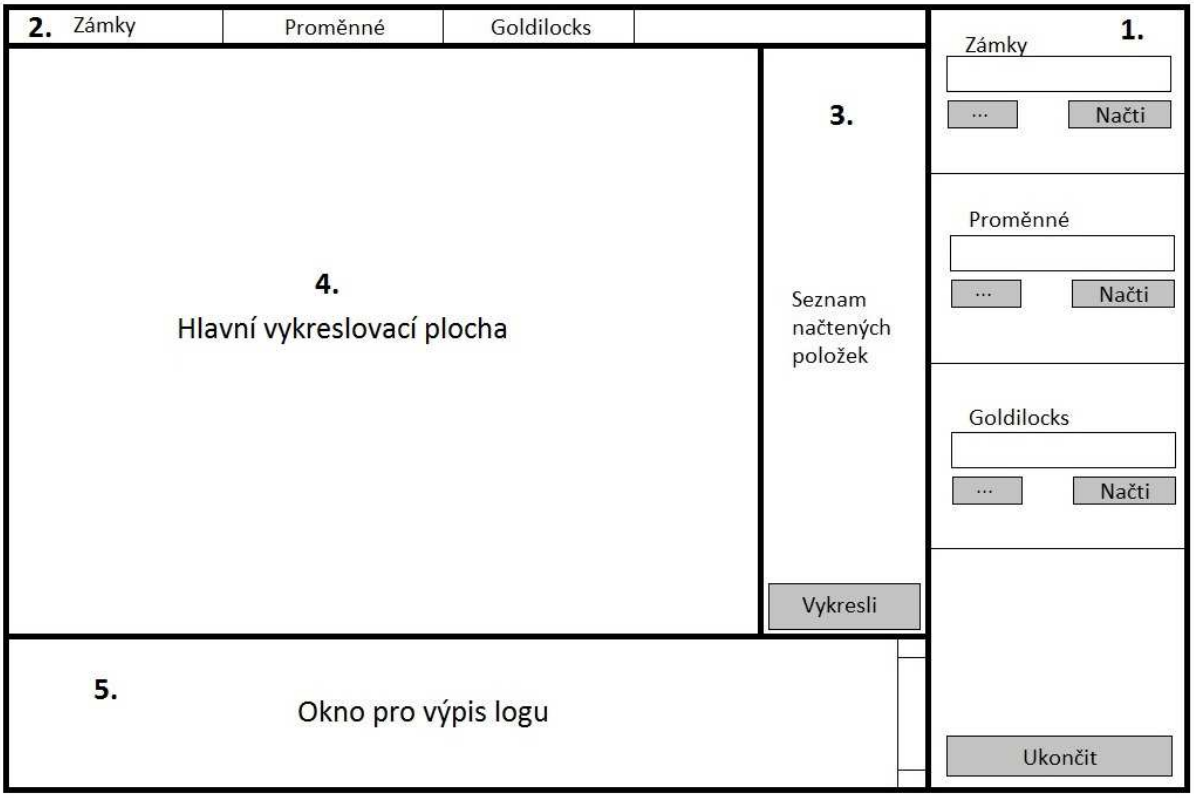
Grafické uživatelské rozhraní aplikace musí umožňovat uživateli jednoduše a přehledně ovládat aplikaci. Musí zde být možnost vybrat vstupní soubory, vybrat jednotlivé položky ze vstupních souborů a hlavně zde musí být plocha, na kterou se bude vykreslovat výsledek. Uživateli by určitě pomohl i nějaký log, který by informoval o událostech v aplikaci. Na obrázku 5.2.1 je návrh, jak by mělo grafické uživatelské rozhraní vypadat.

Okno je rozděleno na tři části. První z nich je nastavení v pravé části okna. Druhou je "hlavní vykreslovací okno" spolu se "seznamem načtených položek" zabírající většinu okna. Navíc tato část je řízena záložkami v horní části okna. Pro každý typ souboru existuje vlastní vykreslovací okno, což umožňuje aplikaci vykreslit výsledek pro všechny typy souboru zároveň a jen mezi nimi přepínat pomocí záložek. Poslední částí je okno pro výpis logu v levé dolní části okna. Následuje popis jednotlivých částí okna. Části mají pro větší přehlednost popisky a jsou očíslované:

1. **Výběr vstupních souborů.** V pravé části okna je pro každý typ souborů vyhrazené pole, do kterého půjde zadat cesta k vstupnímu souboru. Po stisknutí tlačítka označeného "..." se otevře dialogové okno, které umožní vybrat vstupní soubor pomocí průzkumníka a ten cestu k souboru vyplní za uživatele. Po stisknutí tlačítka "Načti" dojde k načtení souboru a zároveň je naplněn i seznam načtených položek v odpovídající záložce. Úplně dole je tlačítko "Ukončit", které, jak už popisek napovídá, ukončí aplikaci.
2. **Záložky pro přepínání výstupu.** V levé horní části okna jsou tři záložky. Pro každý vstupní soubor jedna. Tyto záložky přepínají pohled na hlavní vykreslovací plochu a seznam

načtených položek. Jak už bylo výše zmíněno, toto umožňuje mít vykreslené všechny tři typy vstupních souborů najednou a jednoduše mezi nimi přepínat.

3. **Seznam načtených položek.** Po načtení souboru bude toto podokno obsahovat seznam položek načtených ze souboru (např. seznam zámků). Položky bude umožněno hromadně vybrat. Ve spodní části podokna je tlačítko "Vykresli". Po jeho stisknutí bude vykreslen výsledek s ohledem na vybrané položky na hlavní vykreslovací ploše.
4. **Hlavní vykreslovací plocha.** Zde bude po řádcích vykreslen výsledek. Po řádcích je myšleno tak, že položky budou vykreslovány vodorovně pod sebe. Pro zámky to budou různé kombinace, v jakých byli zámky uzamčeny a (v lepším případě) následně odemčeny. Pro proměnné bude výsledek vypadat dost podobně jako u zámků. U algoritmu Goldilocks bude pouze graficky znázorněna chyba, k jaké došlo.
5. **Okno pro výpis logu.** V tomto okně se budou zobrazovat zprávy pro uživatele. Zprávy budou spíše technického rázu, jako například informace o správném načtení souboru, nebo jak dlouho trvalo zpracování a vykreslení dat. Také se zde budou vypisovat případné chybové hlášky. Zřejmě bude existovat možnost log uložit do souboru, nebo vyčistit výpisové okno.



5.2.1 Návrh grafického uživatelského rozhraní

6 Implementace

V předchozí kapitole byl představen návrh architektury a grafického uživatelského rozhraní. V této kapitole jsou blíže popsány implementační detaily samotné aplikace. Podkapitola 6.1 popisuje způsob, jakým byl napsán zdrojový kód. V podkapitole 6.2 je podrobněji popsána architektura aplikace a způsob, jakým aplikace funguje. Podkapitola 6.3 se věnuje grafickému uživatelskému rozhraní.

6.1 Jazyk zdrojového kódu aplikace

V úvodu této podkapitoly budou přiblíženy obecné implementační detaily, jako například jmenné konvence. Následovat bude podrobný diagram tříd. V závěru podkapitoly je podrobný rozbor některých zajímavých metod.

Aplikace je napsána v jazyce Java. Tento jazyk jsem zvolil hned z několika důvodů. Hlavním důvodem bylo, že se jedná o na platformě nezávislý objektově orientovaný jazyk, který umožňuje velmi přehledně a jednoduše tvořit grafická rozhraní. Dalším důvodem bylo, že celá tato práce je úzce spjatá s testováním aplikací napsaných právě v Javě, a proto mi připadalo jen přirozené držet se tohoto jazyka.

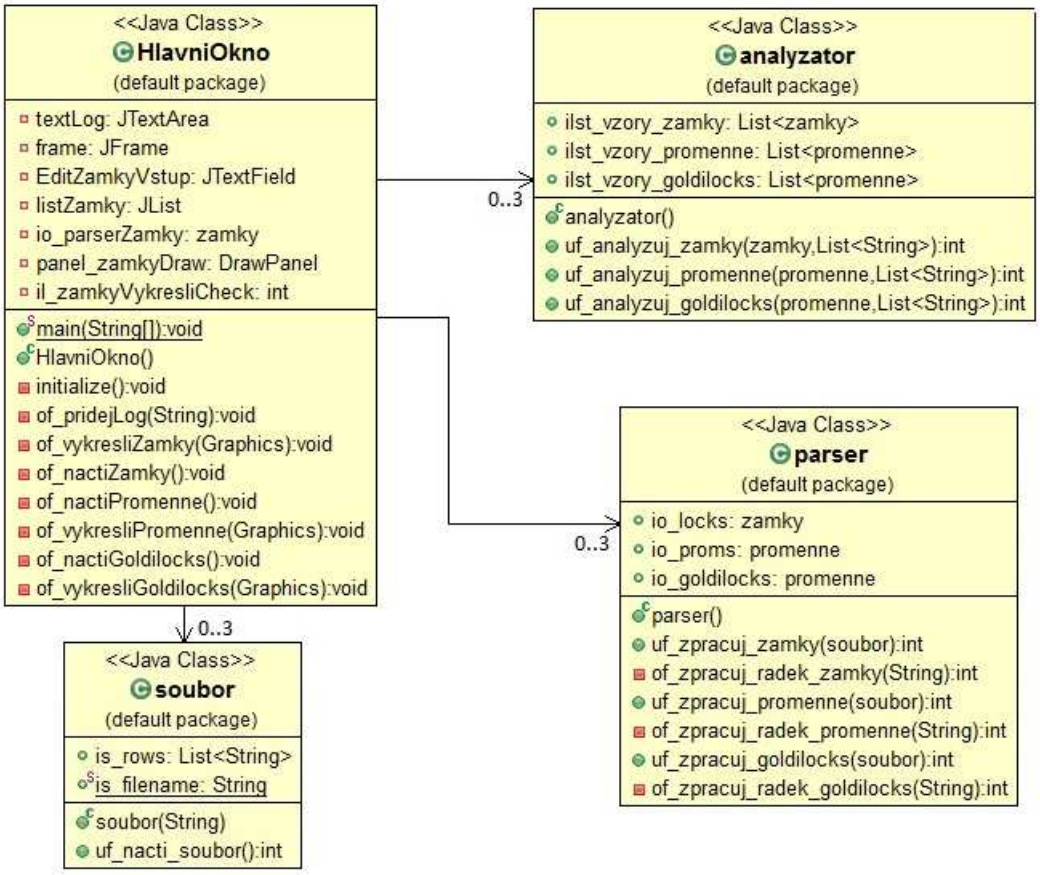
Při psaní zdrojového kódu byla pro udržení přehlednosti dodržována jmenná konvence pro pojmenovávání proměnných a funkcí. Tato konvence je založena na prefixu v názvu, který je většinou dvouznakový a následovaný podtržítkem ("_"). První z těchto znaků u proměnných určuje, zda se jedná o argument metody, instanční, nebo lokální proměnnou. U metod první znak určuje, zda se jedná o veřejnou (*public*) nebo soukromou (*private*) metodu. Druhý znak je většinou počáteční písmeno názvu datového typu u proměnných a "f" u metod a funkcí. Většinou proto, že u některých datových typů jsou počáteční písmena názvu shodná. V takových případech je prefix delší, aby bylo na první pohled zřejmé, o jaký datový typ se jedná.

Dále platí, že pro každou třídu vznikl vlastní zdrojový soubor, který se jmenuje stejně jako samotná třída. Tento zdrojový soubor následně obsahuje pouze metody, které se přímo vztahuj k dané třídě a tím značně zvyšuje schopnost orientovat se v kódu.

Samozřejmostí jsou i hlavičky u jednotlivých zdrojových souborů. Tyto hlavičky obsahují především hrubý popis toho, co třída vlastně dělá. To stejné platí i pro všechny metody, což umožňuje vygenerovat přehlednou programovou dokumentaci, která umožní rychlý přehled o existujících třídách a metodách a o tom k čemu tam jsou.

6.2 Architektura aplikace

Při tvorbě vnitřní struktury aplikace bylo postupované podle návrhu. Každá třída aplikace reprezentuje některou z části aplikace představených v kapitole 5.1. Na obrázku 6.1 je digram tříd, který je hned následován popisem jednotlivých tříd.



6.1 Diagram tříd aplikace

1. **`HlavniOkno`.** Třída obsahuje metodu `main()`, což znamená, že v této třídě začíná vykonávání samotného programu. Třída se také stará o všechny úkony spojené s grafickým rozhraním a zajišťuje obsluhu jednotlivých komponent (viz kapitola 6.3). Kromě `main()` obsahuje třída i některé další metody. Metoda `initialize()` se stará o samotné zobrazení okna a správné vykreslení všech komponent. Metoda `of_pridej_log()` se stará o výpis zpráv do okna pro výpis zpráv pro uživatele. Další metody tvoří tři dvojíce. Pro každý vstupní soubor jedna. Princip u každého souboru je stejný, proto je zde pro zjednodušení popsána pouze jedna dvojíce. Metoda `of_nactiZamky()` spustí načtení souboru pomocí třídy `soubor` a hned poté jeho zpracování pomocí třídy `parser`. Na konec ještě naplní seznam nalezených položek (pro podrobnosti viz kapitola 6.3). Metoda `of_vykresliZamky()`pošle data načtená třídou `parser` spolu s položkami vybranými v seznamu dohledaných položek k dalšímu zpracování ve třídě `analyzator`. Poté se volá vykreslení výsledku.

2. **`soubor`.** Tato třída reprezentuje vstupní soubor. Funkčností odpovídá části aplikace pro načítání souboru představené v návrhu. Má pouze jednu metodu `uf_nacti_soubor()`, která zajišťuje veškerou funkčnost definovanou v návrhu. Jednotlivé řádky souboru jsou uloženy do seznamu řetězců `is_rows: List<String>`.
3. **`parser`.** Třídá `parser` opět odpovídá částí programu představené v návrhu, a to, jak už jméno napovídá, část pojmenovanou jako parser. Třída obsahuje tři soukromé metody a tři veřejné metody, které mezi sebou tvoří dvojíce. Tři proto, že každá metoda odpovídá jednomu typu vstupních souborů. Všechny tři dvojíce pracují na stejném principu. Veřejná funkce (např. `uf_zpracuj_zamky`) prochází seznam řádků načtených ve třídě `soubor` a tyto řádky následně posílá jako argument soukromé funkci (např. `of_zpracuj_radek_zamky`), která se už stará o jeho rozdělení a uložení jeho obsahu do připravené struktury (např. `io_locks`).
4. **`analyzator`.** Tato třída svojí funkčností odpovídá části analyzátor představené v návrhu. Obsahuje tři veřejné metody. Pro každý typ souboru jedna. Tyto metody jako argument obdrží data načtená ve třídě `parser` a seznam položek označených v seznamu dohledaných položek. Následně data projde a dohledá opakující se vzory které uloží do odpovídající struktury.

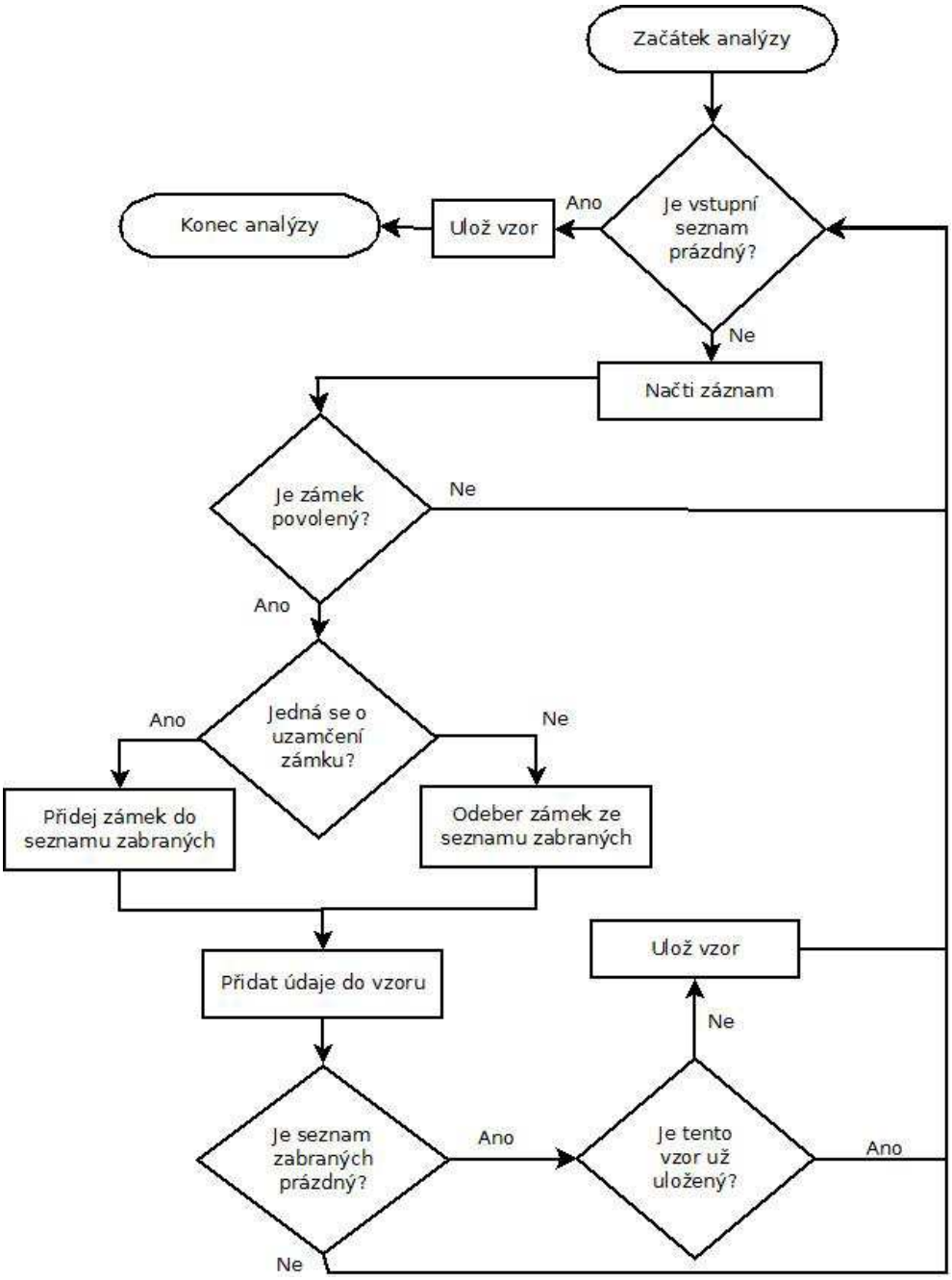
Z diagramu tříd je vidět, že třída `HlavniOkno` vytváří instance ostatních tříd. Instanci každé třídy může vytvořit maximálně třikrát, a to v případě že jsou zpracovávány všechny tři typy vstupních souborů najednou.

6.2.1 Průběh analýzy dat zámků

V této podkapitole bude podrobněji přiblížen proces analýzy dat zámků, protože se dle mého názoru jedná o nejsložitější a nejzajímavější část celého programu. Z toho důvodu se zde nachází vývojový diagram, který je na obrázku 6.2. Následuje podrobný popis průběhu vývojového diagramu.

Analýza začíná ve stavu "Začátek analýzy". Předpokládá se, že před začátkem analýzy byl naplněn vstupní seznam data ze třídy `parser` a seznam s povolenými zámky. Hned poté následuje test na prázdnost vstupního seznamu. Pokud je vstupní seznam prázdný, uloží se aktuálně rozpracovaný stav a analýze je ukončena. Pokud vstupní seznam není prázdný, jsou z něj načtena další data. Tato načtená data se následně porovnávají se seznamem povolených zámků. Pokud není aktuálně načtený zámek povolený, přesune se výpočet zpět na začátek. V opačném případě se pokračuje dále. Nyní se zkoumá, jaký typ záznamu byl načten. Pokud se jedná o zabrání zámků, pak se přidá záznam do seznamu zabraných zámků. Naopak, pokud se jedná o uvolnění zámku, je tento záznam nalezen a odstraněn ze seznamu zabraných zámků. Ať už se jednalo o jakýkoliv typ, jsou následně data zapsána do záznamu o vzoru přístupů. Následuje test na prázdnost seznamu zabraných zámků. Pokud není seznam prázdny, pokusí se aplikace načíst další záznam. Pokud je seznam

prázdný, tak ihned následuje kontrola na duplicity, která zjišťuje, zda nebyl aktuálně načtený vzor již uložen. Pokud nebyl, pak se tento vzor uložený do seznamu nalezených vzorů. Pokud byl, pak je tento vzor zahozen a aplikace se vrátí zpět na začátek, kde se pokusí načíst další záznam. Tento postup se pak opakuje, dokud není vstupní seznam prázdný.



6.2 Vývojový diagram procesu analýzy

6.3 Grafické uživatelské rozhraní aplikace

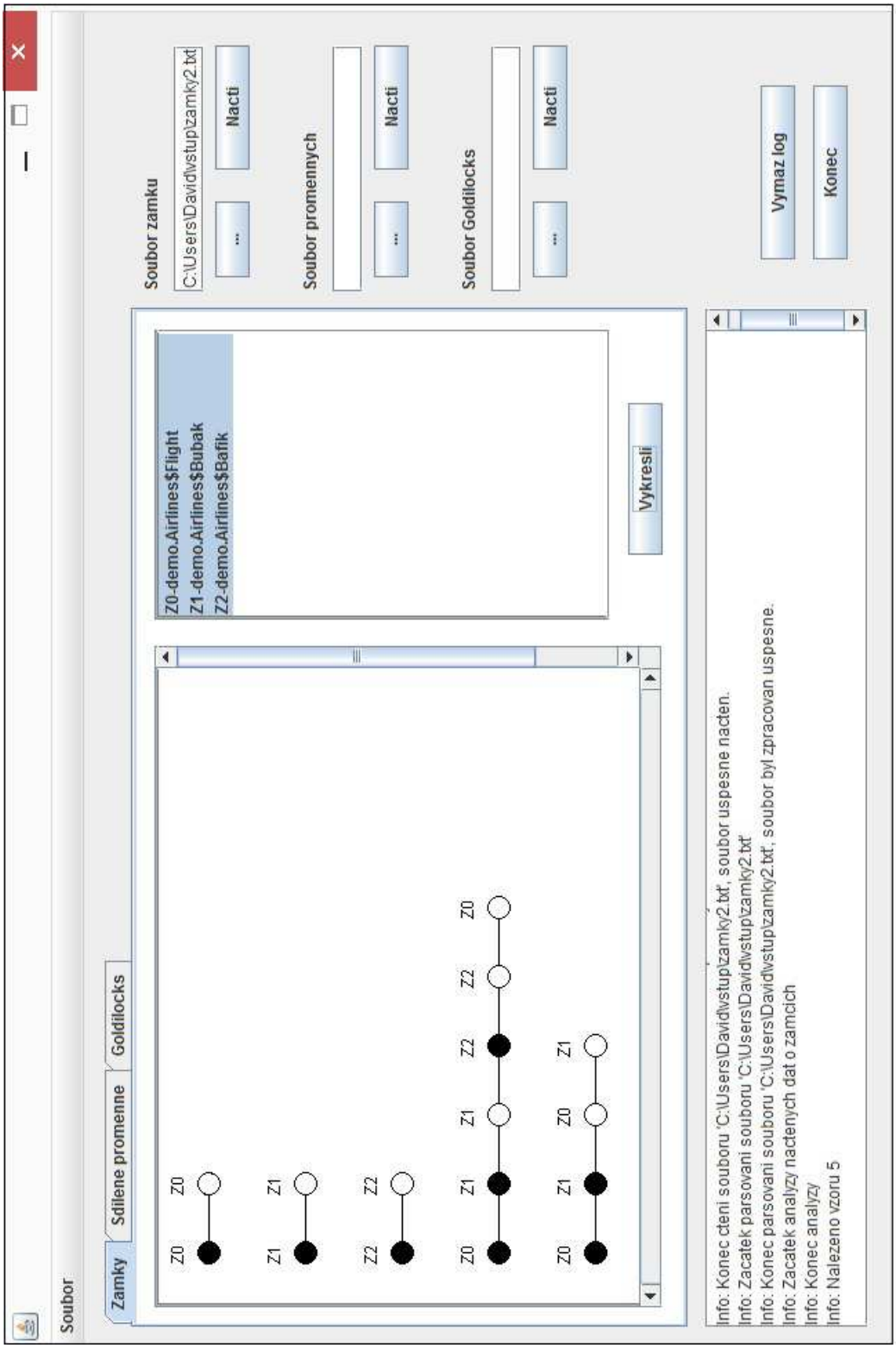
Pro vytvoření grafického uživatelského rozhraní byl použit rámec Swing, který byl představen v kapitole 2.1.2.Tento rámec byl zvolen především kvůli tomu, že umožnil splnit všechny požadavky, které byly na grafické rozhraní kladeny (např. dynamické vykreslování). V celém grafickém rozhraní byla vynechána diakritika z důvodu rozdílných znakových sad na různých operačních systémech.

Grafické rozhraní aplikace se z velké většiny drží návrhu, který byl představen v kapitole 5.2, což lze vidět na obrázku 6.3. Vpravo se nachází panel, na kterém se zadávají vstupní soubory. Pro každý typ vstupního souboru je zde pole pro zadání cesty, tlačítko označené "...", které vyvolá dialog pro vybraní souboru, a tlačítko "Nacti". Toto tlačítko volá načítací funkci, která byla představena v kapitole 6.2 a je pro každý typ souboru trochu jiná (např. pro zámky `of_nactiZamky()`). V tomto panelu se dále nachází i tlačítko "Konec" a oproti návrhu je zde i navíc tlačítko "Vymaz log", které vymaže všechny záznamy v okně pro výpis zpráv pro uživatele.

Ve spodní části okna aplikace se nachází stejně jako v návrhu již zmíněné okno pro výpis zpráv pro uživatele. Jak lze vidět na obrázku 6.3, vypisují se zde různé informace o běhu aplikace. Kromě těchto informací jsou zde vypisovány i chybové hlášky a různá varování. Jak je také vidět na obrázku, pokud už je zde příliš mnoho zpráv, než aby se dali najednou zobrazit, objeví se v pravé části okna posuvná lišta, která umožní zobrazit starší zprávy, které už "přetekly za okraj". Obsah okna je možné označovat a kopírovat dle libosti, avšak přímo zadávat text do tohoto okna je pro uživatele zablokováno.

Zbytek okna aplikace zabírá panel přepínatelný pomocí záložek, což také odpovídá návrhu. Každá záložka tohoto panelu obsahuje již zmiňovaný seznam dohledaných položek, tlačítko "Vykresli" a panel pro samotné vykreslování. Seznam dohledaných položek je naplněn po načtení souboru jednotlivými položkami daného typu (např. jednotlivé zámky). Z těchto položek pak je možné vybrat, které chceme vykreslit. Výběr probíhá klasicky pomocí myši. Pro výběr více položek je třeba podržet klávesu "CTRL" pro vybírání po jednom, nebo klávesu "SHIFT" pro intervalové vybírání. Panel pro vykreslování se při vykreslování více výsledků dynamicky zvětšuje a pohled na zakryté části je možný pomocí lišt na pravé a dolní straně. Po stisknutí tlačítka "Vykresli" se spustí analýza načtených dat a následně vykreslení výsledku analýzy. Při každém stisknutí tlačítka vykresli dojde k nové analýze a překreslení celého vykreslovacího okna.

Oproti návrhu je v aplikaci ještě navíc i rozbalovací menu, které se nachází v horní části okna. Tato funkčnost však není příliš rozpracovaná, a tak se po kliknutí na nápis soubor nabídne pouze možnost ukončit aplikaci.



Obrázek 6.3 Grafické uživatelské rozhraní aplikace

7 Testování aplikace

Výsledná aplikace byla testována několika způsoby. Nejdříve byly provedeny testy funkčnosti aplikace na předem připravené testovací sadě. Při těchto testech byla také sledována rychlost vykreslování. Následně byla testována intuitivnost grafického uživatelského rozhraní aplikace. Výsledky testování jsou popsány dále.

Načtení vstupních souborů a vykreslení výsledků dopadlo u všech provedených testů dle očekávání. Aplikace si dokázala poradit i s poměrně velkým počtem údajů ve vstupních souborech. Byly jí předány ke zpracování i vstupní soubory obsahující různé chyby, což mělo za úkol otestovat, zda aplikace dokáže tyto chyby odhalit. Ověření správnosti výsledku bylo provedeno metodou systematického porovnávání vykresleného výsledku se vstupními soubory. Vykreslení při všech testech proběhlo naprosto okamžitě, tudíž jsem od plánovaného měření rychlosti upustil. Vstupní soubory, na kterých byly prováděny testy jsou obsaženy na přiloženém CD.

Test intuitivnosti grafického rozhraní byl proveden tak, že aplikace byla předložena osmi programátorům ve věku od dvaceti do čtyřiceti let. Tito programátoři si aplikaci vyzkoušeli a následně mi sdělili své názory. Nejčastější připomínkou byla absence nápovědy v programu. Druhou nejčastější připomínkou bylo, že okno aplikace nejde zvětšovat, popřípadě zmenšovat. Poslední častější připomínkou bylo, že by horní rozbalovací nabídka mohla nabízet více možností. Kromě těchto drobností byli všichni schopní s aplikací po krátké instruktáži pracovat. Z toho plyne, že grafické rozhraní aplikace je dostatečně pochopitelné a použitelné.

8 Závěr

Cílem této bakalářské práce bylo vytvořit nástroj, který by usnadnil vývoj vícevláknových aplikací tak, že by umožnil programátorovi grafický pohled na to, jakým způsobem jeho program zachází se synchronizačními primitivy a tím mu pomoci odhalit různé chyby spojené se synchronizací jednotlivých vláken.

Aby se toho dalo docílit, bylo potřeba se podrobně seznámit s nástrojem IBM ConTest (viz kapitola 2.4), jehož výstupy byli použity jako vstupní soubory pro aplikaci vytvořenou v rámci této bakalářské práce. Nejprve bylo nutné provést podrobnou analýzu všech požadavků a také vymyslet formu zobrazování jednotlivých typů vstupních souborů. Následně bylo potřeba vytvořit návrh grafického rozhraní a celé architektury aplikace (viz kapitola 5).

Vytvořený program má intuitivní grafické uživatelské rozhraní, je jednoduchý na používání a zpracování a vykreslování dat je velice rychlé. V rámci této bakalářské práce proběhlo i testování, jehož cílem bylo dokázat bezproblémovou funkčnost výsledné aplikace (viz kapitola 7), což se nakonec podařilo.

Možným rozšířením do budoucna by mohlo být přidání dalšího typu zpracovatelných souborů. Například sledování životního cyklu konkrétního vlákna, kde by bylo vidět, k jakým synchronizačním primitivům vlákno přistupovalo.

Literatura

[1] FARCHI, E., Y. NIR a S. UR. 2003. Concurrent bug patterns and how to test them. *Proceedings International Parallel and Distributed Processing Symposium* [online]. IEEE Comput. Soc, : 7- [cit. 2015-05-18]. DOI: 10.1109/IPDPS.2003.1213511. ISBN 0-7695-1926-1. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1213511

[2] What is a Data Race? 2010. *Oracle Help Center* [online]. [cit. 2015-05-18]. Dostupné z: http://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

[3] CHESS: Systematic Concurrency Testing. 2015. *CodePlex: Project Hosting for Open Source Software* [online]. [cit. 2015-05-18]. Dostupné z: https://chesstool.codeplex.com/

[4] YU, Jie, Satish NARAYANASAMY, Cristiano PEREIRA a Gilles POKAM. 2012. Maple. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12* [online]. New York, New York, USA: ACM Press, : 485- [cit. 2015-05-18]. DOI: 10.1145/2384616.2384651. ISBN 9781450315616. Dostupné z: http://dl.acm.org/citation.cfm?doid=2384616.2384651

[5] SAVAGE, Stefan, Michael BURROWS, Greg NELSON, Patrick SOBALVARRO a Thomas ANDERSON. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* [online]. New York: Association for Computing Machinery, **15**(4): 391-411 [cit. 2015-05-18]. DOI: 10.1145/265924.265927. ISSN 07342071. Dostupné z: http://portal.acm.org/citation.cfm?doid=265924.265927

[6] ELMAS, Tayfun, Shaz QADEER a Serdar TASIRAN. *Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets* [online]. : 193 [cit. 2015-05-18]. DOI: 10.1007/11940197_13. Dostupné z: http://link.springer.com/10.1007/11940197_13

[7] Java Swings/AWT. 2015. *Wikibooks: Open books for open world* [online]. [cit. 2015-05-18]. Dostupné z: http://en.wikibooks.org/wiki/Java_Swings/AWT

[8] Swing (Java). 2015. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation [cit. 2015-05-18]. Dostupné z: http://en.wikipedia.org/wiki/Swing_%28Java%29

[9] SWT: The Standard Widget Toolkit. 2015. *Eclipse: The Eclipse Foundation open source community website* [online]. [cit. 2015-05-18]. Dostupné z: http://www.eclipse.org/swt/

[10] SwingLabs: Swing Component Extension - Project Kenai. *Java.net: The Source for Java Technology Collaboration* [online]. 2014 [cit. 2015-05-18]. Dostupné z: https://swingx.java.net/

[11] JavaFX - The Rich Client Platform. *Oracle: Hardware and Software, Engineered to Work togehter* [online]. 2014 [cit. 2015-05-18]. Dostupné z: http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

[12] B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In Proc. of Runtime Verification—RV'11, San Francisco, CA, USA, volume 7186 of LNCS, pages 177–192, 2012. Springer-Verlag.

[13] Hodnocení testů – metriky. *Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování* [online]. 2014 [cit. 2015-05-18]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/hodnoceni-testu-metriky/

[14] Peter Haggar. Practical Java: Programming Language Guide. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] J. Šimša, R. Bryant, and G. Gibson. dbug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In Proc. of Model Checking and Software Verification—SPIN'00, London, UK, volume 1885 of LNCS, pages 188–193, 2000. Springer-Verlag.

[16] Hernandez, C., Rivera, M. & Lopez, J. (2005). VisualSync: An Interactive Visualization Tool for Teaching/Learning Multithreaded Programming with Synchronization Primitives. In G. Richards (Ed.), Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2005 (pp. 2943-2949). Chesapeake, VA: Association for the Advancement of Computing in Education (AACE). Retrieved May 19, 2015 from http://www.editlib.org/p/21647.

[17] CHASSIN DE KERGOMMEAUX, J., B. STEIN a P.E. BERNARD. 2000. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing* [online]. **26**(10): 1253-1274 [cit. 2015-05-19]. DOI: 10.1016/S0167-8191(00)00010-7. ISSN 01678191. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0167819100000107

[18] HILLAR, Gaston. Visualizing Parallelism and Concurrency in Visual Studio 2010 Beta 2. *Dr. Dobb's: Good stuff for serious developers* [online]. 2009 [cit. 2015-05-19]. Dostupné z: http://www.drdobbs.com/windows/visualizing-parallelism-and-concurrency/220900288

[19] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, volume 41, pages 111–125, January 2002.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications ACM*, volume 21, pages 558–565, January 1978.