VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÁ ANALÝZA PARALELNÍCH PROGRAMŮ NA PLATFORMĚ .NET FRAMEWORK

DYNAMIC ANALYSIS OF PARALLEL APPLICATIONS USING .NET FRAMEWORK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DAVID LING**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Ling David, Bc.**
Program: Informační technologie
Obor: Inteligentní systémy
Název: **Dynamická analýza paralelních programů na platformě .NET Framework**
Dynamic Analysis of Parallel Applications Using .NET Framework
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte principy instrumentace programů .NET Framework. Nastudujte testování paralelních programů. Zaměřte se na odhalování chyb uváznutí a souběh nad daty.
2. Navrhněte instrumentační framework pro vícevláknové aplikace v prostředí .NET Framework. Navrhněte dynamický analyzátor vedoucím vybraného druhu chyb paralelismu.
3. Implementujte instrumentační framework a dynamický analyzátor v jazyku C#.
4. Ověření základní funkcionality podpořte automatickými testy.

Literatura:

- J. Fiedor, V. Hrubá, B. Křena, Z. Letko,S.Ur, and T.Vojnar. Advances in Noise-based Testing of Concurrent Programs. Software Testing, Verification and Reliability, roč. 25, č. 3, s. 272-309. ISSN 1099-1689.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce se zabývá návrhem a implementací dynamického analyzátoru paralelních programů na platformě .NET Framework. V teoretické části práce je rozebrána problematika synchronizace v paralelních programech, instrumentace programů, testování paralelních programů a specifika těchto problémů pro jazyk C# a platformu .NET Framework. Podrobněji jsou popsány vybrané algoritmy pro detekci uváznutí (algoritmus Goodlock) a časově závislých chyb nad daty (algoritmy FastTrack a AtomRace). V následujících částech jsou sepsány požadavky na výsledný analyzátor a vytvořen návrh systému. Práce obsahuje také popis implementace navrženého řešení, způsob kompletního otestování implementovaného nástroje a v neposlední řadě ukázku použití dynamických analyzátorů v reálném aplikačním prostředí.

Abstract

The thesis deals with a design and implementation of the dynamic analyser of parallel applications on the .NET Framework platform. The problematic of synchronization in parallel applications, the instrumentation of such an applications, testing of parallel applications and a specifics of these problems for C# language and for the platform .NET Framework are discussed in the theoretical part. Selected algorithms for detection of deadlocks (the algorithm of Goodlock) and data-race errors (the algorithm of FastTrack and AtomRace) are described in detail in this part as well. Requirements for the dynamic analyser and the system design is made in the following part of this thesis. The thesis also contains a description of the implementation of the proposed solution, a description of the entire testing of the implemented tool. Last but not least, the thesis describes the sample of using dynamic analysers in a particular application environment.

Klíčová slova

Dynamická analýza, testování paralelních programů, detekce uváznutí, detekce časově závislých chyb nad daty, instrumentace programu, .NET

Keywords

Dynamic analysis, paraller programs testing, deadlock detection, data race detection, program instrumentation, .NET

Citace

LING, David. *Dynamická analýza paralelních programů na platformě .NET Framework.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Dynamická analýza paralelních programů na platformě .NET Framework

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D.. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Ling
16. května 2021

Poděkování

Na tomto místě bych rád poděkoval vedoucímu své práce panu Ing. Alešovi Smrčkovi, Ph.D. za odborné vedení, užitečné rady, připomínky a snahu pomoct s řešením problémů, které se v průběhu práce vyskytly. Dále bych rád poděkoval Mgr. Martinovi Kosmákovi a firmě UNIS a.s. za umožnění spolupráce v rámci praktické části mé práce.

Obsah

Kapitola 1

Úvod

Testování a ladění vícevláknových programů nemusí být jednoduchým úkolem. Vzhledem k povaze těchto programů může být nalezení chyb v nich složitým problémem, který vyžaduje použití speciálních technik a přístupů k jeho řešení. Mezi chyby, které se objevují v paralelních programech, patří také uváznutí (angl. deadlock) a časově závislé chyby nad daty (angl. data race). Především těmito druhy chyb se zabývá tato práce.

Problematice testování vícevláknových aplikaci se na FIT VUT v Brně věnuje výzkumná skupina VeriFIT. Nástroje jako ANaConDA[1] či Java Race Detecor and Healer[2] vyvinuté právě touto skupinou řeší problematiku verifikace paralelních programů v jazycích C/C++ či Java. Hlavním cílem práce je navázat na práci skupiny VeriFIT a rozšířit portfolio nástrojů o dynamický analyzátor, který bude sloužit pro testování programů napsaných v jazyce C# na platformě .NET Framework.

Před samotnou realizací je nutné představit problematiku paralelních programů, popsat druhy chyb, které se v těchto programech vyskytují a možnosti, jak lze takové programy testovat. Důležitá část kapitoly 2 se zabývá instrumentací programů na platformě .NET Framework a popisem algoritmů vhodných pro detekci chyb uváznutí a časově závislých chyb nad daty. Zároveň s tématy uvedenými výše jsou v této kapitole představeny existující používané nástroje pro testování paralelních programů.

V kapitole 3 jsou popsány především požadavky na instrumentační framework a dynamické analyzátory a poté jejich návrh. Detailněji je popsán především zvolený způsob instrumentace programu a důvody pro tuto volbu. Součástí návrhu je i popis komunikace mezi jednotlivými částmi systému a způsob práce jednotlivých analyzátorů.

Kapitola 4 je věnována implementačním detailům navrženého systému. Nejdůležitější pasáže popisují implementaci instrumentace potřebné pro následnou dynamickou analýzu, implementaci dynamických analyzátorů a možnosti konfigurace nástroje Taipan, který byl v rámci této práce vyvinut.

V předposlední kapitole číslo 5 je vysvětlen a prakticky ukázán způsob testování vyvinutého nástroje. Popsány jsou zvlášť jednotkové a integrační testy. V krátkosti jsou zde představeny také dva ukázkové programy, na kterých lze vidět použití nástroje včetně konfigurace a výstupů, jaké nástroj poskytuje.

Na závěr je zařazena kapitola 6 popisující využití nástroje Taipan při testování systému MES PHARIS. Jsou prezentovány výsledky, kterých bylo docíleno spojením dynamické analýzy nástroje Taipan s analýzou statickou za využití nástrojů Parallel Checker a Parallel

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda/
[2] http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/

Helper. V kapitole jsou popsány reálné chyby, které testovaný systém obsahoval a které byly analýzou odhaleny.

Kapitola 2

Paralelní programy a jejich testování

V posledních desetiletích lze pozorovat prudký vývoj v oblasti počítačových procesorů. Ve všech parametrech, kterými jsou běžné procesory popisovány (počet tranzistorů, frekvence či počet jader), lze zaznamenat mnohonásobné vylepšení procesorů.

Pro ilustraci lze porovnat parametry procesoru Intel 80286 z roku 1982 s procesorem od stejné firmy s označením i9 z roku 2020.

	Intel 80286	Intel i9
Počet tranzistorů	134 000	$> 10^9$
Frekvence	4 –12 MHz	až 5,2 GHz
Počet jader	1	10

Pro to, aby byl program schopen využít dvě a více jader procesoru zároveň (a stal se z něj program paralelní), může využít procesy či vlákna. Každý přístup má své výhody i nevýhody a volba přístupu je ovlivňována především požadavky na výsledný systém.

Na program využívající procesy lze nahlížet jako na několik současně probíhajících programů, kde každý z programů má svůj adresový prostor [1]. Výhody tohoto přístupu lze vidět v tom, že každý z procesů má své místo v paměti, se kterým pracuje, a tím pádem jsou jednotlivé procesy více nezávislé a umožňují tak systém lépe škálovat. Nevýhody mohou být naopak v tom, že si procesy musejí vyměňovat data, která chtějí mezi sebou sdílet. Díky tomu naopak komplexita roste, protože je nutné myslet i na konzistenci dat a operace zajišťující správnost dat v jednotlivých procesech většinou činí celkový systém pomalejší a složitější než v případě použití vláken.

V programu, který je tvořen jedním procesem, ale více vlákny (nazýván *vícevláknový*) naopak všechna vlákna sdílí jeden adresový prostor. Tento přístup je efektivnější z pohledu využití prostředků. To, že všechna vlákna sdílí stejná data, ovšem přináší i komplikace spojené s implementací těchto programů. Je nutné zajistit správnou synchronizaci mezi vlákny nad sdílenými prostředky, což v komplexních systémech není vůbec jednoduchý úkol. Korektní přístup ke sdíleným prostředkům je ve vícevláknových programech zajišťován synchronizačními primitivy, které jsou popsány v části 2.2.

Část programu, kde dochází k přístupu ke sdíleným prostředkům z různých vláken či procesů je nazývána *sdílenou kritickou sekcí*. V těchto sekcích je nutností zařídit, aby nebyly prováděny paralelně z více vláken či procesů najednou.

Problému zajištění korektní synchronizace mezi sdílenými kritickými sekcemi se říká *problém kritické sekce*. Problém zahrnuje také:

- Problém vzájemného vyloučení – nanejvýš jeden z procesů se v daném momentě nachází ve sdílené kritické sekci.
- Problém dostupnosti kritické sekce – je-li kritická sekce dostupná, proces nesmí čekat neomezeně dlouhou dobu na povolení přístupu do ní.

Při řešení problému kritické sekce je zapotřebí se vyhnout synchronizačním chybám, které jsou více popsány v kapitole 2.3.

2.1 Logický čas a relace Happens-before

V paralelních či distribuovaných systémech je jednou ze základních otázek určení časové kauzality mezi jednotlivými událostmi, které se provádějí souběžně nezávisle na sobě.

Relace uspořádání mezi dvěma nezávisle na sobě vzniknuvšími událostmi se nazývá relací *Happens-before*. Tato relace byla poprvé představena Lesliem Lamportem v článku Time, Clocks, and the Ordering of Events in a Distributed System [16]. Lamport se zaměřoval především na prostředí distribuovaných systémů a algoritmů, ale definice této relace se dá využít jak pro oddělené procesy, tak vlákna. Relace *Happens-before* $\rightarrow$, která umožňuje definovat časové uspořádání mezi dvěma událostmi v paralelním programu, je definována jako [16]:

$$\forall e_i^x, \forall e_j^y \in H, e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow e_j^y, \text{ pokud } i = j \wedge x < y \\ \text{nebo} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{nebo} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$$

kde e_i^x značí x-tou událost procesu i, H značí množinu všech událostí provedených v paralelním programu a relace $\rightarrow_{msg}$ značí závislost mezi událostmi typu $send(m)$ (odeslání zprávy m) nebo $recv(m)$ (přijmutí zprávy m).

Pro jakékoliv dvě události e_i a e_j platí podle relace *Happens-before* následující pravidla:

$$e_i \nrightarrow e_j \nRightarrow e_j \nrightarrow e_i$$

$$e_i \rightarrow e_j \Rightarrow e_j \nrightarrow e_i$$

Souběžnými událostmi jsou nazývány události e_i a e_j, pro které platí $e_i \nrightarrow e_j \wedge e_j \nrightarrow e_i$. Takové události se mohou v různých bězích programu provést pokaždé v jiném pořadí.

Logickým časem je v paralelním programu chápáno označení události časovým razítkem. Tento logický čas nikterak nesouvisí s časem fyzickým [16]. Slouží pouze pro určení globálního uspořádání událostí mezi jednotlivými vlákny/procesy.

Systém logického času se skládá z časové domény T a logických hodin C [15]. T je množina prvků, nad kterou je definována relace částečného uspořádání $<$. Tato relace je tou

samou relací Happens-before, která je definována výše. Logické hodiny jsou reprezentovány funkcí $C(e)$, která mapuje událost e na prvek z časové množiny T. Funkce je nazývaná taktéž časové razítko a je definována následovně:

$$C : H \mapsto T$$

kde H je množina všech událostí. Funkce musí splňovat podmínku konzistence:

$$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

Implementace logického času se dá převést na nalezení odpovědí na dvě otázky:

- Jaké datové struktury zvolit pro uchovávání logického času lokálně pro jednotlivé procesy?
- Jak navrhnout protokol pro aktualizaci těchto struktur, aby byla naplněna podmínka konzistence?

Návrh protokolu spočívá v určení dvou základních pravidel [15]:

- Pravidlo 1 určuje, jak proces aktualizuje své lokální logické hodiny při provedení události.
- Pravidlo 2 určuje, jak proces aktualizuje svůj pohled na globální logické hodiny (logický čas celého systému).

Logický čas v paralelních systémech je obvykle implementován jako čas skalární, vektorový či maticový [15].

Skalární čas

Časovou doménou T je množina nezáporných celých čísel. Logické hodiny procesu/vlákna p_i i jeho pohled na globální čas jsou reprezentovány jedinou nezápornou celočíselnou hodnotou C_i. Hodnota C_i je posílána procesem/vláknem v každé zprávě. Před provedením události proces/vlákno p_i aktualizuje svou hodnotu C_i následovně:

$$C_i := C_i + d \qquad (d > 0)$$

Hodnota d může být jakékoliv nezáporné celé číslo, ale většinou je rovna 1.
Při přijetí zprávy z jiného procesu/vlákna je hodnota C_i upravena takto:

$$C_i := max(C_i, C_{msg})$$

kde C_{msg} je hodnota C_j procesu j, který zprávu odeslal.
Protokol je jednoduchý na implementaci, ale nezaručuje podmínku silné konzistence ($C(e_i) < C(e_j) \nRightarrow e_i \rightarrow e_j$).

Vektorový čas

V systému s vektorovým časem je časová doména reprezentována jako množina n-rozměrných vektorů nezáporných celých čísel. Každý proces p_i si uchovává vlastní vektor $vt_i[1..n]$, kde $vt_i[i]$ je lokální hodnota logických hodin procesu p_i a $vt_i[j]$ reprezentuje poslední známou informaci procesu p_i o lokálním čase procesu p_j. Tento vektor je zasílán s každou zprávou příjemci zprávy. Před provedením události proces/vlákno p_i aktualizuje svou hodnotu podobně jako u skalárního času:

$$vt_i[i] := vt_i[i] + d$$

Po přijetí zprávy proces p_i aktualizuje všechny hodnoty vektoru následovně:

$$1 \leq k \ \leq n : vt_i[k] := max(vt_i[k], vt_{msg}[k]),$$

kde vt_{msg} je vektor odesilatele zprávy.

Protokol zajišťuje silnou konzistenci, ale pro velké počty procesů/vláken v systémů narůstají paměťové nároky pro každý proces/vlákno a roste i objem zprávy, která je přenášena, protože musí vždy obsahovat celý vektor lokálních hodin odesilatele. Existují techniky, které se snaží tento nedostatek řešit tím, že se ke zprávě připojují jen hodnoty, které se změnily od poslední odeslané zprávy.

Maticový čas

Maticový čas je podobný času vektorovému s tím rozdílem, že každý proces uchovává matici $n \times n$ a má tak povědomí o lokálních hodinách všech ostatních procesů v systému.

2.2 Synchronizační primitiva a jejich implementace v .NET Framework

Synchronizační primitiva jsou ve vícevláknových programech používána pro řízení paralelního přístupu ke sdíleným datům. V této sekci je uveden základní výčet synchronizačních primitiv, přičemž mnoho dalších prostředků z nich může být odvozeno.

Semafor

Semafory lze chápat jako celočíselné hodnoty (integer) s následujícími rozdíly [9]:

1. Při inicializaci semaforu je hodnota nastavena jakkoliv, ale dále jsou povolenými operacemi pouze `decrement` (odečtení čísla jedna) a `increment` (přičtení čísla jedna), tzn. nelze číst aktuální hodnotu semaforu.

2. Pokud vlákno sníží hodnotu semaforu na zápornou hodnotu, je zablokováno do té doby, než jiné vlákno hodnotu nezvýší.

3. Pokud vlákno zvýší hodnotu semaforu a existují vlákna, která na toto zvýšení čekají, je jedno z těchto čekajících vláken odblokováno.

Binární semafor je semafor, jehož hodnota je po inicializaci rovna 1.

V .NET Framework je semafor reprezentován třídou `System.Threading.Semaphore` [20]. Tato třída v konstruktoru přijímá počáteční počet položek a maximální počet souběžných položek. Každý semafor může mít také svůj vlastní název.

Mutex

Mutex je binární semafor, který zajišťuje výlučný přístup procesů či vláken do kritické sekce. Mutex si lze představit jako token, který je předáván mezi vlákny. Pokud chce vlákno vstoupit do kritické sekce, musí tento token získat a po vystoupení ze sekce jej opět uvolní pro ostatní vlákna.

V .NET Framework jsou mutexy reprezentovány třídou `System.Threading.Mutex`, která zapouzdřuje mimo jiné metody `WaitOne` a `ReleaseMutex`, kde první z nich zablokuje vlákno dokud neobdrží signál pro vstup do kritické sekce a druhá uvolní mutex pro ostatní vlákna [20].

Základní třídou pro **semafory** a **mutexy** je v .NET Framework abstraktní třída `WaitHandle`[1], která zapouzdřuje prvky operačního systému pro synchronizaci nad sdílenými prostředky.

Zámek

Zámky jsou podobné semaforům, ale jelikož jsou reprezentovány sdílenou proměnnou, lze je využít pouze pro vlákna. V .NET Framework jsou implementovány například pomocí struktury `System.Threading.SpinLock`, která při snaze vlákna o získání zámku, který je ve vlastnictví jiného vlákna, čeká ve smyčce a zkouší opakovaně obdržet zámek, dokud není uvolněn.

Monitor

Monitor je abstraktní datový typ (zapouzdřuje data a poskytuje metody pro práci s těmito daty). Metody monitoru zajišťují výlučný přístup k datům pro různá vlákna [27].

V .NET Framework je implementován třídou `System.Threading.Monitor`, v níž jsou definovány metody `Enter` a `Exit`, pomocí kterých může vlákno získat nebo uvolnit zámek, který je reprezentován jakýmkoliv objektem. V praxi se pro práci s monitory v jazyce C# doporučuje využít klíčového slova `lock` [20]. Zápis:

```
lock(x)
{
    Kritická sekce ...
}
```

pak odpovídá

```
object _lockObj = x;
bool _lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(_lockObj, ref _lockWasTaken);
    Kritická sekce ...
```

[1] https://docs.microsoft.com/cs-cz/dotnet/api/system.threading.waithandle?view=netframework-4.6.1

```
}
finally
{
    if (_lockWasTaken) System.Threading.Monitor.Exit(_lockObj);
}
```

2.3 Chyby v synchronizaci

Nekorektní použití synchronizačních prostředků (například těch, které byly popsány v předešlé kapitole) nebo jejich naprosté opomenutí může vést k chybám v paralelních výpočtech. Základní typy těchto chyb jsou v následující části popsány a vysvětleny na jednoduchých příkladech.

2.3.1 Uváznutí

Uváznutí (angl. deadlock) lze definovat jako trvalé blokování skupiny procesů, které mezi sebou komunikují, nebo mezi sebou „soutěží“ o systémové prostředky. Uváznutím se nazývá stav, kdy každý proces ve skupině procesů čeká na událost, která může být vyvolána pouze jiným z procesů v této skupině. Tento stav trvá nekonečně dlouho, protože žádná z událostí nemůže nastat [28]. Pro to, aby k uváznutí došlo, musí dojít k naplnění následujících čtyř podmínek:

- Vzájemné vyloučení – pouze jeden proces smí v daný okamžik přistupovat ke sdílenému prostředku.
- Proces smí být držitelem jednoho sdíleného prostředku a zároveň žádat o přidělení jiného prostředku.
- Neexistence preempce – žádnému z procesů nesmí být odebrány prostředky bez jeho spolupráce, prostředky uvolňuje sám proces po dokončení využití těchto prostředků.
- Cyklická závislost mezi procesy – mezi procesy musí existovat cyklická závislost, kdy všechny procesy z dané skupiny čekají na uvolnění prostředků jiného procesu z této skupiny.

Pro vypořádání se s uváznutími existují různé přístupy, mezi které patří především:

- Prevence uváznutí – zajištění toho, aby nemohlo dojít k naplnění všech podmínek pro uváznutí (vždy alespoň jedna podmínka nesmí být splněna).
- Vyhýbání se uváznutí – systém dynamicky provádí rozhodnutí na základě aktuálního stavu alokovaných zdrojů.
- Detekce uváznutí – monitorování toho, jestli nebyly naplněny všechny podmínky pro uváznutí a pokud ano, spuštění obnovy systému do stavu bez uváznutí.

V praxi může uváznutí vypadat následovně. Mějme program, který obsahuje deklaraci dvou zámků L_1 a L_2 a který po svém spuštění vytvoří vlákna T_1 a T_2 a v těchto vláknech volá metodu `start1` respektive `start2`.

```
Lock L1, L2;

void start1() {                    void start2() {
    L1.lock();                        L2.lock();
    L2.lock();                        L1.lock();
    L2.unlock();                      L1.unlock();
    L1.unlock();                      L2.unlock();
}                                  }
```

Kód 2.1: Pseudokód programu obsahujícího chybu uváznutí

Pokud by po spuštění takového programu nastala situace, že vlákno T_1 zamkne zámek L_1 a vlákno T_2 ve stejném okamžiku zamkne zámek L_2, došlo by k uváznutí, jelikož vlákno T_1 by se snažilo v dalším kroku zamknout zámek L_2, který je ale vlastněn vláknem T_2 a vlákno T_2 by se v tom stejném okamžiku snažilo zamknout zámek L_1, který je ale vlastněn vláknem T_1. V této situaci by ani jedno z vláken nemohlo další zámek zamknout, a tudíž by obě vlákna nekonečně dlouho čekala na přístup, který nemohou v dané situaci získat.

2.3.2 Blokování

Blokováním (angl. blocking) je nazývána situace, kdy vlákno (nebo skupina vláken) čekají na stav či událost, ke které nemůže dojít nebo na kterou není nutno čekat. Rozdíl oproti uváznutí je v tom, že blokováno může být pouze jedno vlákno, ale ostatní vlákna můžou běžet v pořádku (a držet ve vlastnictví prostředky, o které žádá blokované vlákno).

2.3.3 Stárnutí

Stárnutí (angl. starvation) je situace podobná blokování s tím rozdílem, že čekající vlákno čeká na naplnění podmínky, která může být za určitých okolností platná (pokud by nebyla platná nikdy, jednalo by se o blokování).

2.3.4 Časově závislá chyba nad daty

Časově závislá chyba nad daty (angl. data race) nastane v případě, že dvě různá vlákna přistupují ke stejnému místu v paměti, tento přístup není správně synchronizován (může probíhat konkurentně nebo v různém pořadí) a alespoň jeden z těchto přístupů je zápis [22]. Z praktického hlediska dochází k tomuto druhu chyb především při nesprávném (či žádném) použití synchronizačních prostředků v rámci kritických sekcích. Pokud jedno vlákno pracuje s daty, které jiné vlákno může konkurentně měnit, může nastat situace, kdy jedno či druhé vlákno bude počítat s daty, která už v daný okamžik nejsou validní.

Časově závislé chyby nad daty se dají vždy zařadit do jedné ze tří kategorií [13]:

- WAR (Write-After-Read) – čtení z paměti je v konfliktu s pozdějším zápisem.
- RAW (Read-After-Write) – zápis z paměti je v konfliktu s pozdějším čtením.
- WAW (Write-After-Write) – dva nebo více konkurentních zápisů na stejné místo v paměti.

```
int amount, balance, interest;
Lock L1;
```

```
void start1() {
  amount = read_amount();
  lock(L1);
  balance = balance + amount;
  interest = interest + rate*balance;
  unlock(L1);
}
```

```
void start2() {
  amount = read_amount();
  if (balance < amount) {
    printf("Not enough money.");
  } else {
    balance = balance - amount;
    interest = interest + rate*balance;
  }
}
```

Kód 2.2: Nekorektní zápis obsahující časově závislou chybu nad daty

```
int amount, balance, interest;
Lock L1;

void start1() {
  amount = read_amount();
  lock(L1);
  balance = balance + amount;
  interest = interest + rate*balance;
  unlock(L1);
}
```

```
void start2() {
  amount = read_amount();
  lock(L1);
  if (balance < amount) {
    printf("Not enough money.");
  } else {
    balance = balance - amount;
    interest = interest + rate*balance;
  }
  unlock(L1);
}
```

Kód 2.3: Korektní zápis bez časově závislé chyby nad daty

V kódech 2.2 a 2.3 lze vidět 2 zápisy „totožného“ programu pro vklad/výběr z bankovního účtu. Program obsahuje deklaraci zámku L_1 a celočíselných hodnot `amount`, `balance` a `interest` a po svém spuštění vytvoří vlákna T_1 a T_2 a v každém z nich metodu `start1` resp. `start2`. První zápis obsahuje potenciální časově závislou chybu nad daty, druhý zápis je korektní. V zápisu 2.2 totiž vlákno T_1 pracuje se sdílenými proměnnými `balance` a `interest` v kritické sekci chráněné zámkem L_1, zatímco vlákno T_2 zapisuje do stejných proměnných bez užití zámku, tudíž může přepsat data vláknu T_1. V zápisu 2.3 lze vidět, že přidáním kritické sekce do metody `start2` vlákna T_2 chráněné stejným zámkem, jakým je chráněna kritická sekce ve vlákně T_1, vznikne korektní verze programu, jelikož obě kritické sekce nemohou být prováděny současně.

Snaha o detekci časově závislých chyb nad daty je nedílnou součástí ladění paralelních programů. Konkrétní algoritmy, které s odhalením tohoto druhu chyb mohou pomoci, jsou popsány v kapitole 2.6.

2.3.5 Porušení atomicity

K porušení atomicity (angl. atomicity violation) dochází v případě, kdy je kritická sekce uzavřena dříve, než je korektní, a přístupy do paměti, které na sebe navazují, se tak neobjeví v rámci jedné kritické sekce, ale jsou rozděleny do více sekcí [18]. První vlákno v rámci jedné kritické sekce může získat hodnotu sdílené proměnné, se kterou bude dále počítat, ale druhé vlákno tuto hodnotu může v průběhu výpočtu prvního vlákna změnit.

Jako příklad je uveden program, který obsahuje definici celočíselné hodnoty `value` a zámku L_1. Po spuštění programu jsou vytvořena vlákna T_1 a T_2 a v nich metoda `start`. Jednotlivá volání této funkce mohou být proložena tak, že hodnota `value` je inkrementována pouze jednou a ne dvakrát. Korektní zápis by spočíval v tom, že funkce `start` by obsahovala pouze jednu kritickou sekci chráněnou zámkem L_1 a v ní příkazy pro načtení, inkrementaci i uložení proměnné `value`.

```
int value;
Lock L1;

void start() {
    int temp;
    lock(L1);
    temp = value;
    unlock(L1);
    temp++;
    lock(L1);
    value = temp;
    unlock(L1);
}
```

Kód 2.4: Ukázka programu obsahujícího porušení atomicity

2.4 Testování paralelních programů

V následující sekci jsou v prvních částech rozebrány různé přístupy k testování paralelních programů. V dalších částech je popsán problém instrumentace programu a možnosti instrumentace programů napsaných v jazyce C# pro .NET Framework. Následně je vybráno několik nástrojů pro testování paralelních programů pro .NET Framework, které jsou představeny detailněji.

2.4.1 Statická analýza

Statická analýza je přístup k testování paralelních programů, který spočívá v tom, že program je zkoumán bez jeho spuštění [23]. Statická analýza poskytuje metody, které mohou zkoumat vlastnosti běhu programu bez jeho exekuce. Statická analýza většinou probíhá ještě před samotným překladem programu a využívá se pro detekci chyb, které mohou vést např. k předčasnému ukončení programu či špatně definovanému výsledku. Statická analýza se často používá například k detekci chyb jako jsou:

- přístup k prvku pole mimo jeho rozsah,
- únik paměti,
- špatná alokace paměti,
- přístup k neinicializované proměnné či ukazateli,
- nevalidní aritmetické operace,
- ...

Hlavní myšlenkou při provádění statické analýzy je abstrakce systému do systému jiného, který stále obsahuje zásadní vlastnosti původního systému, ale zjednodušuje jej tak, aby bylo snadnější ho analyzovat. Přesnost analýzy lze měřit pomocí klasifikace výstupu analýzy do tří kategorii:

- falešné alarmy nebo pozitiva – neexistující chyby nahlášené jako chyby
- falešná negativa – opravdové chyby, které ovšem nejsou analýzou detekovány
- pravdivé alarmy – opravdové chyby, které jsou analýzou detekovány

Hlavními výhodami statické analýzy je to, že může být použita bez spouštění samotného programu v raných částech vývoje, není moc nákladná a je většinou velmi rychlá. Za nevýhody lze považovat především to, že analyzuje pouze aproximaci skutečného programu a produkuje velké množství falešných alarmů. Tato vlastnost komplikuje programátorovi detekci reálných chyb mezi těmi, které chybami ve skutečnosti nejsou.

2.4.2 Dynamická analýza

Dynamická analýza (narozdíl od statické) testuje program pomocí jeho provádění a analyzování informací o programu sbíraných za jeho běhu [4]. Hlavní myšlenkou dynamické analýzy je to, že reálný program je nainstrumentován (více v kapitole 2.4.3) a jsou do něj vloženy části kódu, které poskytují z pohledu analýzy zajímavé informace o běhu programu. Analyzátor (program, který provádí analýzu nad získanými informacemi) tyto informace zpracovává a detekuje chyby nejen v aktuálním běhu programu, ale i chyby, které by mohly nastat v jiném běhu programu a jiném proložení jednotlivých operací. Analýza je většinou prováděna nad opakovanými běhy programu. Pro kompletnost analýzy je nutné specifikovat různé parametry pro jednotlivé běhy programu, aby analýza detekovala co největší počet potenciálních problémů, které mohou nastat.

Výhody dynamické analýzy spočívají především v preciznosti, jelikož je zkoumán reálný běh programu, a nejen jeho abstrakce. Další z výhod je např. možnost analyzovat jen vybranou část programu (ne program celý).

Nevýhody lze spatřovat v tom, že výstup analýzy většinou nebude kompletní, jelikož není možné nasimulovat každý z potenciálních běhů programu, kterých může být nekonečně mnoho. Tento problém lze řešit např. technikami vkládání šumu, které jsou popsány níže. Dalším problémem může být náročnost vytvoření datové sady, která by pokrývala co největší počet možných problémových běhů programu. Jako poslední lze zmínit špatné možnosti škálování, jelikož dynamická analýza vysoce komplexních programů může být velmi výpočetně a paměťově náročná. Tento problém se dá řešit omezením analýzy pouze na části systému, které jsou z pohledu analýzy nejzajímavější.

Dynamické analyzátory se dají klasifikovat dle toho, jaké chyby jsou schopné detekovat. Analyzátor lze označit jako *sound*, pokud detekuje všechny chyby, které při dané exekuci programu nastanou. *Precise (precizní)* jsou pak analyzátory, které neprodukují falešné alarmy.

Aby dynamická analýza mohla detekovat co nejvíce potenciálních problémů, je nutné (a zároveň velmi náročné) vytvořit podmínky pro běh programu takové, aby bylo možné nasimulovat co největší počet možných běhů programu. Program se totiž může za různých podmínek chovat jinak (především proložení jednotlivých vláken může být jiné), a tím pádem mohou nastat i jiné chyby. Pro docílení nasimulování co nejvíce případů, se používají techniky **vkládání šumu** či **deterministického testování**.

Metody vkládání šumu spočívají v tom, že se do programu vkládají části kódu, které zpozdí určitá vlákna a tím bylo docíleno i méně častých proložení vláken [12]. V praxi se vkládání šumu provádí tak, že je provádění vlákna po určitou dobu pozastaveno, než pokračuje ve své exekuci, a tím je vynuceno přepnutí kontextu na jiné vlákno. V jazyce C# a frameworku .NET Framework toho lze dosáhnout například:

- Prázdným cyklem.
- Voláním funkce `System.Threading.Thread.Sleep`, která přijímá jediný parametr → počet milisekund, po které je vlákno pozastaveno.
- ...

Doba, po kterou je vlákno pozastaveno, je nazývána *síla šumu* (čím delší doba, tím silnější šum). Hlavními otázkami, na které je nutno si odpovědět před použitím této metody, jsou:

- Kam v programu šum vkládat (angl. *noise placement problem*)?
- Jak silný šum na daná místa vložit (angl. *noise seeding problem*)?

Deterministické testování je testování, kdy exekuce programu je plně v režii deterministického plánovače, který se snaží otestovat co nejvíce různých proložení vláken [12]. Plánovač pracuje tak, že před každou instrukcí, která je z pohledu analýzy důležitá, uloží aktuální stav do stavového prostoru jako uzel a vypočte všechna možná rozložení vláken, která do stavového prostoru uloží jako možné přechody. Tato reprezentace umožňuje zaznamenání si každého jednotlivého běhu testovaného programu a replikaci konkrétního běhu. Samotné přechody mezi stavy probíhají tak, že pokračování v exekuci je umožněno jen některým vláknům. Ostatní vlákna jsou pozastavena (např. vložením šumu) a pokračují až po určité době.

2.4.3 Instrumentace programu

Pojem instrumentace označuje možnost monitorování programu či měření jeho výkonu pro diagnostiku chyb, které program obsahuje [20]. V praxi to znamená především:

- Sledování kódu – přijímání informačních zpráv o aktuálním běhu programu v průběhu jeho vykonávání.
- Ladění – vysledování a oprava chyb programu v průběhu vývoje.
- Výkonostní měření – sledování výkonu aplikace.
- Logování událostí – uchovávání hlavních událostí, které v průběhu programu nastaly. Událostí může být chyba v programu i jiná informace dávající v kontextu exekuce smysl (spuštění určité metody, změna dat, ...).

Instrumentace jako termín tedy pokrývá celou sadu vývojářských technik, které pomáhají programátorovi lépe porozumět tomu, co se aktuálně v prováděném programu odehrává a na základě toho poté provádět optimalizaci, opravu chyb či vizualizovat aktuální stav programu.

Existuje spousta přístupů k instrumentaci programu, které se liší tím, kdy (a na jakém kódu) je instrumentace prováděna.

Statická instrumentace je způsob instrumentace, kdy je instrumentační kód vkládán do programu při překladu před samotným spuštěním programu [14]. Druhou možností statické instrumentace je modifikace již zkompilovaných programů či knihoven.

Instrumentace zdrojového kódu (angl. source-to-source) je prováděna samotným programátorem při vývoji programu. Ten je zodpovědný za to, že program bude poskytovat informace o svém aktuálním stavu.

Dynamická analýza je prováděna tak, že instrumentační kód je vložen do programu dynamicky v průběhu provádění programu. V prostředí .NET Framework dynamická analýza spoléhá na to, že programy využívají Common Language Runtime (zkr. CLR) a Just-In-Time compiler (viz 2.5.1), který kompiluje kód právě načítaného modulu do nativního kódu. Instrumentace je provedena při tomto překladu za běhu programu.

2.5 Prostředí .NET Framework

V následující části jsou popsána určitá specifika prostředí .NET Framework z pohledu instrumentace. Je vysvětleno především to, jak je program v .NET Framework kompilován a prováděn. Dále jsou ukázány možnosti instrumentace takového programu a nástroje, které se pro testování paralelních programů v .NET Framework využívají.

2.5.1 Common Language Infrastructure

Common Language Infrastructure (zkr. CLI) je standard (ECMA 335), který popisuje rozhraní pro provádění .NET programů a formát prováděného kódu. CLI je rozhraní, pro které mohou být aplikace napsány ve více vysokoúrovňových jazycích tak, aby byly spustitelné v různých prostředích bez nutnosti přepisovat tyto aplikace pro všechna tato prostředí [10]. V následujících sekcích je dokumentováno několik hlavních součástí tohoto rozhraní.

Common Language Runtime

Common Language Runtime (zkr. CLR) je prostředí pro běh programu kompatibilní s CLI. Je to kompletní vysokoúrovňový virtuální stroj navržený pro podporu několika programovacích jazyků a spolupráci mezi nimi [19]. Mezi hlavní vlastnosti CRL patří:

- **Garbage collection (GC)** – označení pro automatické uvolňování nevyužívané paměti. GC zajišťuje to, aby objekty, na které již nadále neexistuje v programu žádný odkaz, byly odstraněny z paměti. CLR si udržuje odkazy na objekty v paměti na hromadě a čas od času uvolní paměť, která je využívána objekty, u kterých již není třeba. Všechna uvolněná paměť může být dále v programu použita. GC je nástroj, který především usnadňuje práci programátorovi a zjednodušuje psaný kód, protože není potřeba ručně volat metody pro mazání objektů.
- **Paměťová bezpečnost** – vlastnost CLR, která značí zajištění toho, aby program byl bezpečný, co se týče paměti. To znamená, že program pracuje pouze s pamětí, která byla pro daný program alokována (a neuvolněna). Kontroly při běhu programu zajišťují, aby nebyly použity ukazatele do paměti, která nebyla alokována (nebo byla již uvolněna).

- **Typová bezpečnost** – požadavek na to, aby každá alokace paměti byla přiřazena určitému typu. Každý objekt v paměti smí provádět pouze operace, které jsou validní pro typ jemu přiřazený.

Hlavním cílem CLR je tedy usnadňovat vývoj aplikací pro programátory. Především zjednodušuje zápis programu tím, že přebírá zodpovědnost za některé operace, které by jinak musel dělat programátor.

Kód psaný pro CLR je zkompilován ve formě modulů do tzv. *assembly*, které mají většinou příponu `.dll` nebo `.exe`. Takto zkompilované moduly mohou být poté načteny v průběhu provádění programu jako tzv. spravovaný (angl. managed) kód.

Spravovaný a nespravovaný kód

Spravovaný kód (angl. managed code) je kód, který poskytuje CLR dostatek informací pro to, aby CLR mohl správně vykonávat:

- Vyhledávání deklarací jednotlivých funkcí.
- Procházet zásobník.
- Zpracovávat výjimky.

Tyto informace mohou být poskytovány pomocí metadat modulu nebo pomocí sledování jeho exekuce.

Naopak kód, který takové informace neposkytuje, je nazýván **nespravovaný** (angl. unmanaged) či nativní. Tento kód nemůže být dál kontrolován a většinou se spouští ještě před inicializací samotného CLR. Nativní kód je specifický pro samotné prostředí (operační systém, architekturu procesoru), na kterém lze CLR spustit a umožňuje programátorovi přistupovat k nižším úrovním daného systému.

Common Intermediate Language

Common Intermediate Language (CIL) je instrukční sada využívána CLR. Programy napsané pomocí této instrukční sady mohou být virtuálním strojem (CLR) interpretované nebo přeložené pomocí Just-In-Time (JIT) kompilátoru do nativního kódu. V kódu 2.5 je možné vidět ukázku programu „Hello, World!“ napsaného pomocí CIL. Tento program nastaví pomocí direktivy `.maxstack` maximální velikost zásobníku a poté na tento zásobník uloží parametr typu `string` s hodnotou `Hello, World!` a zavolá funkci `WriteLine` ze jmenného prostoru `System.Console` a stejnojmenného modulu. Poté je program ukončen navrácením prázdné hodnoty z metody `Main`;

```
.method private hidebysig static void Main (string [] args)
   cil managed {
   .maxstack 8
   .entrypoint

   IL_0000: ldstr "Hello, World!"
   IL_0005: call void [System.Console]System.Console::WriteLine(string)
   IL_000a: ret
}
```

Kód 2.5: Ukázka programu Hello, world! in CIL

Just-In-Time kompilátor

Just-In-Time kompilátor (JIT) je součástí CLR a je to výchozí způsob exekuce programů napsaných pro .NET Framework. Pokaždé, když se CLR snaží načíst nový modul, pro který nenalezl nativní obraz, je tento modul přeložen JIT kompilátorem do nativního kódu.

Při prvním načtení modulu je do virtuální tabulky metod pro každou metodu inicializován záznam jako volání JIT kompilátoru. Při prvním volání metody je metoda zpracována JIT kompilátorem, který přeloží její kód do kódu nativního a je zodpovědný za správné namapování této metody (pomocí skoku) na místo v paměti, kde začíná nativní kód.

Narozdíl od Ahead-Of-Time (AOT) kompilace (kdy je nutno do nativního kódu přeložit všechny metody, jelikož nelze předpokládat, které metody budou při samotné exekuci volány) jsou metody překládány až za běhu programu. Každá metoda je kompilována maximálně jednou při prvním volání.

2.5.2 Instrumentace programů v .NET Framework

V sekci 2.4.3 jsou popsány přístupy, které se využívají při instrumentaci programů. V této sekci jsou představeny konkrétní nástroje, které lze využít pro instrumentaci programů v .NET Framework. Sekce se věnuje nástrojům, které jsou používány pro instrumentaci statickou, jelikož tyto nástroje jsou využity v praktické části práce.

PostSharp

PostSharp[2] je proprietární nástroj, který podporuje celou řadu programovacích vzorů, pomocí kterých se snaží snížit komplexitu psaného kódu a zjednodušit jeho údržbu.

Možnost instrumentace programů je jen jednou ze součástí celého systému, mezi které patří také usnadnění logování, cachování, validace parametrů funkcí, ...

Nástroj PostSharp umožňuje instrumentaci metod (přidání kódu před/za volání dané metody či přidání kódu, který se provede po tom, co je v dané funkci vyvolána výjimka), atributů (přidání kódu před/za čtení či zápis do konkrétní proměnné) a dalších prvků jazyka C#.

Samotná instrumentace je prováděna pomocí tzv. aspektů. Ty lze přiřazovat k jednotlivým třídám, metodám či atributům tříd pomocí atributů nebo pomocí reflexe[3] zvolit prvky, kterým daný aspekt náleží. Instrumentaci lze provést při samotném překladu, kdy jsou modifikované metody uloženy do binárních souborů i modifikací již zkompilovaného binárního souboru. Oba přístupy vyžadují existenci konfiguračního souboru, ve kterém lze specifikovat jednotlivé aspekty, které se mají využít, licenci a další parametry.

PostSharp je distribuován skrze NuGet balíček, což je způsob instalace knihoven třetích stran v prostředí .NET. Balíček lze instalovat do projektu pomocí správce balíků NuGet příkazem `Install-Package`[4].

Výhody: robustnost řešení, stabilně vyvíjený nástroj, variabilita použití, lze využít i pro jiné problémy než instrumentace.

Nevýhody: relativně drahá licence (ovšem s možností evaluace zdarma) pro nekomerční (vědecké) využití.

[2] https://www.postsharp.net/

[3] Objekty, které popisují jednotlivé assembly, moduly a typy.

[4] https://www.nuget.org/

Mono.Cecil

Mono.Cecil[5] je open-source projekt, který umožňuje především analýzu zkompilovaných binárních souborů pro prostředí .NET Framework a/nebo jejich modifikaci. Knihovna Mono.Cecil poskytuje objektovou reprezentaci .NET programů a je využívána v mnoha projektech, ve kterých je zapotřebí zkoumat, modifikovat či generovat kód v CIL.

Za hlavní výhody této knihovny se dá považovat především vyzrálost projektu, jelikož knihovna je vyvíjena již od roku 2004 a vytvořila se kolem ní silná komunita, která usnadňuje práci novým programátorům množstvím příkladů či tutoriálů. Výhodou je i možnost modifikace programu, který byl napsán v jakémkoliv z jazyků, které lze do CLI zkompilovat.

Nevýhodou je zejména to, že neumožňuje instrumentaci nativních knihoven operačního systému a poměrně složitý vývoj co se týče ladění jednotlivých funkcionalit.

dnlib

dnlib[6] je open-source knihovna zaměřující se na práci s již zkompilovanými programy. Usnadňuje čtení/zápis modulů ve formátu `.dll` souborů. Pomocí abstrakce nad metodami reflexe jazyka C# umožňuje jednoduše načíst typy, metody, atributy, ... modulu/třídy a všechny tyto elementy poté měnit.

Knihovna umožňuje programátorovi sestoupit až na úroveň instrukční sady CIL a upravovat kód modulu pomocí této instrukční sady. To je dle mého názoru největší výhodou i nevýhodou této knihovny. Pomocí této instrukční sady má programátor naprostou kontrolu nad tím, jak cílový modul modifikuje, ale samozřejmě je tím na něj kladen nárok na perfektní znalost této instrukční sady. Instrumentace již zkompilovaného modulu pomocí této knihovny tedy může být z implementačního hlediska v jazyku C# náročnějším úkolem než za pomoci jiných knihoven.

Harmony

Harmony[7] je knihovna, která umožňuje jednoduše poskytovat jinou funkcionalitu pro metody programu napsaném v jazyce C#. Základním funkčním blokem této knihovny je tzv. patch, který přidává/mění funkcionalitu konkrétní metody. Umožňuje přidat kód před/za volání konkrétní funkce nebo měnit CIL kód funkce pomocí IL procesoru. Při využití této knihovny je možné zachovat zdrojový kód všech souborů netknutý nebo přiřadit konkrétní metodě více patchů s rozdílnou funkcionalitou.

Výhody: umožňuje implementaci pouze pomocí jazyka C#, jednoduché rozhraní, lze využít ladění při vývoji funkcionality.

Nevýhody: podporuje pouze instrumentaci metod (ne přístupu k proměnným), nelze instrumentovat již přeložené programy (`.dll` či `.exe`).

Další nástroje

Mezi dalšími nástroji, které lze využít při instrumentaci programů v .NET Framework, stojí za zmínku uvést například **Easy Hook** či **Castle Core**. Tyto knihovny umožňují jednoduše nahrazovat implementace metod implementacemi alternativními, avšak neobsahují kompletní požadovanou funkcionalitu, která je zapotřebí v rámci této práce (více v kapitole 3.1).

[5] https://github.com/jbevain/cecil
[6] https://github.com/0xd4d/dnlib
[7] https://github.com/pardeike/Harmony

2.5.3 Nástroje pro testování paralelních programů v .NET Framework

Existuje spousta nástrojů pro testování paralelních programů napsaných v jazyce C# pro platformu .NET Framework. V následující kapitole jsou představeny ty, které používají pro nalezení chyb dynamickou analýzu. Každý z nástrojů se zaměřuje na jiný druh chyb, a proto je vždy nutné zvážit individuálně pro každý projekt, jaký nástroj (nebo kombinaci nástrojů) je vhodné použít.

CHESS

CHESS je nástroj z dílny americké firmy Microsoft z roku 2007. CHESS jako nástroj používá přístup k testování paralelních programů zvaný „testování na základě konkurenčních scénářů“ (angl. concurrency scenario testing). Tento přístup byl představen v článku *CHESS: A Systematic Testing Tool for Concurrent Software* [21]. Spočívá v tom, že pro zajímavé konkurentní scénáře (scénáře, které byly vývojářem systému vytipovány jako důležité či potenciálně problematické) se pomocí techniky *model checking* vygenerují takové běhy programu, aby pokryly všechna možná proložení vláken v daném scénáři. Model checking je technika, která pro model systému s konečným počtem stavů provádí kontrolu jeho správnosti na základě určených požadavků.

Pro vícevláknové programy je tato technika často implementována pomocí plánovače, který je zodpovědný za to, která vlákna aktuálně v systému běží a plánuje běh vláken tak, aby jednotlivé běhy pokryly všechna možná proložení vláken.

Mezi jednu z hlavních nevýhod této techniky je tzv. *exploze stavového prostoru* (angl. state space explosion), kdy počet stavů modelu může růst exponenciálně už pro velmi malé programy. Tento problém řeší nástroj CHESS tak, že limituje počet přepnutí vláken mezi sebou. Speciální plánovač je navržen tak, že vlákno může provést jakýkoliv počet kroků, než je kontext přepnut na jiné vlákno. Přístup je založen na předpokladu, že pro detekci většiny chyb v synchronizaci stačí použít malý počet systematicky zvolených přepnutí vláken.

Pro instrumentaci programu využívá rozhraní Win32 API[8]. Nástroj CHESS monitoruje volání synchronizačních funkcí tohoto rozhraní a záměrně zavádí do těchto volání nedeterminismus.

Pokud některý z testovacích scénářů skončí chybou, nástroj je schopen tento běh reprodukovat pomocí stejného proložení vláken. Nástroj lze využít pro nalezení klasických chyb v paralelních programech (např. těch, které jsou popsané v kapitole 2.3).

The Intel Thread Checker

The Intel Thread Checker je nástroj, který se zaměřuje na detekci uváznutí (i těch potenciálních), časově závislých chyb nad daty a nesprávného užívání nativního rozhraní Windows pro synchronizaci Win32 API [26]. Thread Checker dokáže provést instrumentaci zdrojového kódu i již zkompilovaných binárních souborů. Instrumentovaný program udělá z každého přístupu do paměti či volání synchronizačního API pozorovatelnou událost a při jejím provedení poskytne analyzátoru takové informace, aby byl schopen provést analýzu správnosti programu.

Časově závislé chyby nad daty jsou detekovány pomocí algoritmů založených na relaci *Hapens-before*. Algoritmy pro detekci časově závislých chyb nad daty pracující nad relací Happens-before hledají přístupy do paměti, které nejsou deterministicky uspořádané podle této relace a tyto přístupy poté označí jako potenciální chyby (více v sekci 2.6.2).

[8] https://docs.microsoft.com/en-us/windows/win32/sync/synchronization

SharpDetect

SharpDetect je nástroj vyvinutý na MFF UK v Praze v rámci diplomové práce SharpDetect: Dynamic Analysis Framework for C#/.NET Programs [7]. Autor práce Bc. Andrej Čižmárik se ve své práci zabýval detekcí časově závislých chyb nad daty. Instrumentace je v nástroji SharpDetect prováděna za pomocí knihovny dnlib popsané výše. Nástroj implementuje algoritmy `FastTrack` a `Eraser` pro detekci data race chyb. Nástroj byl testován nad knihovnou NetMQ[9], ve které odhalil jednu reálnou chybu. Nástroj sice splňuje některé z požadavků popsaných v kapitole 3.1, ale mezi jeho hlavní nevýhody patří především nemožnost použití tohoto nástroje na platformě .NET Framework a absenci kompletnější dokumentace.

Parallel Checker

Parallel Checker je oproti ostatním nástrojům popsaným v této kapitole analyzátorem statickým. Důvodem, proč je v této části zmiňován, je to, že spojení statické a dynamické analýzy může vést v testování paralelních programů k zajímavým výsledkům. Dynamická analýza je pro komplexní programy velice náročnou úlohou, a právě statická analýza může s touto náročností pomoci. Statická analýza (i přes větší množství falešných alarmů) může vyselektovat místa v programu s potenciálními chybami. Dynamická analýza poté nemusí být prováděna nad celým komplexním systémem, ale může se zaměřit pouze na místa, která byla statickou analýzou označena jako potenciálně problematická.

Parallel Checker je distribuován v podobě rozšíření do vývojového prostředí Visual Studio, které je hojně využíváno vývojáři aplikací v jazyce C#. Parallel Checker dokáže provádět statickou analýzu přímo při psaní kódu a varovat vývojáře před potenciální chybou.

2.6 Algoritmy pro detekci chyb v synchronizaci

V následující sekci jsou rozebrány algoritmy pro detekci chyb v paralelních programech. Dynamický analyzátor je program, který analyzuje exekuci jiného programu a snaží se detekovat chyby v testovaném programu za jeho běhu. V této sekci jsou popsány analyzátory, které se využívají pro detekci uváznutí i analyzátory užívané pro detekci časově závislých chyb nad daty. Mezi algoritmy pro detekci uváznutí je vysvětlen princip algoritmu Goodlock a mezi algoritmy pro detekci data race chyb algoritmy FastTrack (založený na vektorovém času a relaci Happens-before, viz 2.1), AtomRace vyvinut na FIT VUT v Brně a Eraser (zástupce skupiny LockSet algoritmů).

Mezi nejdůležitější parametry algoritmů pro testování paralelních programů patří to, jak ovlivňují výkon (angl. performance) testovaného systému a poté to, jaká hlášení obecně algoritmus poskytuje. Při dynamické analýze se počítá s tím, že analyzátor svou činností zpomalí vykonávání samotného programu, který analyzuje. Různé analyzátory ale produkují jinak velké zpomalení. Tím je do značné míry ovlivněna množina potenciálních využití daného analyzátoru. Při volbě analyzátoru je také důležité se zamyslet nad výstupy, které konkrétní analyzátor poskytuje. Především to, jaké procento opravdových chyb (pravdivých alarmů) detekuje a kolik při tom podá falešných hlášení (falešných alarmů).

Obecně lze algoritmy používané pro detekci synchronizačních chyb v paralelních program rozdělit do následujících skupin:

[9] https://netmq.readthedocs.io/en/latest/

- Precizní – neprodukují falešná hlášení, nahlásí chybu pouze v případě, že k ní opravdu v programu došlo.
- Neprecizní – snižují složitost algoritmů a poskytují větší pokrytí programu oproti precizním, ale produkují více falešných hlášení.

Pro potřeby této kapitoly jsou zavedeny notace pro zápis určitých událostí v běžícím paralelním programu, které jsou důležité pro následnou dynamickou analýzu běhu tohoto programu. Tyto notace jsou v každém z algoritmů rozšiřovány pro jeho konkrétní účely.

Paralelní program se skládá z množiny T souběžně běžících vláken, kde každé vlákno má vlastní unikátní identifikátor (značí se t_i, kde $i > 0$). Každé vlákno může manipulovat s proměnnými, které jsou značeny písmeny x, y, z. $Rd(t, x)$ a $Wr(t, x)$ značí událost čtení, resp. zápisu, vlákna t z/do proměnné x. Každé vlákno t může žádat o získání ($acq(t, l)$) či uvolnění ($rel(t, l)$) zámku $l \in L$, kde L je množina všech zámků. V případě více zámků je použit zápis l_i značící i-tý zámek množiny L. Z vlákna t_i může vzniknout operací $fork(t_i, t_j)$ vlákno t_j a vlákna t_i a t_j lze naopak operací $join(t_i, t_j)$ spojit do jednoho.

2.6.1 Algoritmy pro detekci uváznutí

Algoritmus Goodlock

Saddek Bensalem and Klaus Havelund ve svém článku z roku 2005 [5] představili algoritmus pro detekci uváznutí v programu zvaný Goodlock. Algoritmus pracuje nad instrumentovaným programem, který při své exekuci monitoruje metody *zamknutí (lock)* a *uvolnění (unlock)* zámků. Z těchto událostí je poté tvořen graf zámků s hranami symbolizujícími pořadí zamknutí jednotlivých zámků a v tomto grafu poté hledány cyklické závislosti značící uváznutí ve skupině vláken.

Pro detailnější popis algoritmu je nutné nadefinovat základní objekty, se kterými se dále v popisu pracuje. $\sigma = e_1, e_2, \ldots, e_n$ je sekvence událostí lock/unlock (angl. execution trace) vygenerovaných instrumentovaným programem, prvek na pozici i v sekvenci σ značíme $\sigma[i]$. Orientovaný graf G je dvojice (S, R) množin, kde S je množina vrcholů a $R \subseteq S \times S$ je množina hran. Cestou p rozumíme neprázdný graf $G = (S, R)$ ve formě $S = \{x_1, x_2, \ldots, x_k\}$ a $R = \{(x_1, x_2), (x_2, x_3), \ldots, (x_{k1}, x_k)\}$, kde x_i jsou všechna odlišná až na případ, kdy $x_k = x_1$, pak tuto cestu nazýváme cyklem. V případě, že jsou hrany grafu ohodnoceny prvky z množiny W, je graf G trojice (S, R, W) a je nazýván grafem ohodnoceným, kde $R \subseteq S \times W \times S$. Kontext zamknutí značíme $C_L(\sigma, i)$ a tento kontext definuje mapování z vlákna na množinu zámku, které mělo vlákno v držení v čase i. Formálně: $\forall t \in T_\sigma : C_L(\sigma, i)(t) = \{o | \exists j : j < i \wedge \sigma[j] = lock(t, o) \wedge \neg \exists k \ : j < k \ \geq i \wedge \sigma[k] = unlock(t, o)\}$

Graf zámků ze sekvence událostí σ vypočteme jako minimální orientovaný graf $G_L = (L, R)$ takový, že L je množina všech zámku L_σ a $R \subseteq L \times L$ je definováno jako $(l_1, l_2) \in R$, pokud existuje vlákno $t \in T_\sigma$ a čas $i \geq 2$ v σ taková, že $\sigma[i] = lock(t, l_2)$ a $l_1 \in C_L(\sigma, i-1)(t)$.

Množina cyklů v G_L ($cycles(G_L)$) značí množinu potenciálních uváznutí v programu, který vygeneroval sekvenci σ. V článku [5] jsou popsána rozšíření tohoto základního algoritmu, která snižují počet falešných hlášení.

Prvním rozšířením je řešena situace, kdy je cyklus vytvořen jedním vláknem, nebo kdy je cyklus chráněn zámkem obdrženým výše v programu všemi vlákny tvořícími cyklus. Hlavní myšlenkou detekce právě popsaných situací je rozšíření hran grafu zámků o informace o tom, které vlákno způsobilo přidání hrany a jaké zámky v tu dobu mělo ve svém držení. Detekce *validních cyklů* pomocí této informace filtruje falešná hlášení. Rozšířený graf lze pro sekvenci

událostí σ definovat jako minimální orientovaný ohodnocený graf $G_L = (L, W, R)$ takový, že L je množina všech zámků L_σ, $W \subseteq T_\sigma \times 2^L$ je množina ohodnocení hran obsahující identifikátor vlákna a množinu zámků a $R \subseteq L \times W \times L$ je definována jako: $(l_1, (t, g), l_2) \in R$, pokud $\exists t \in t_\sigma \wedge \exists i \geq 2$ takové, že: $\sigma[i] = l(t, l_2) \wedge l_1 \in C_L(\sigma, i-1)(t) \wedge g = C_L(\sigma, i-1)(t)$. Pro graf G_L a cyklus $c = (L, W, R) \in cycles(G_L)$, říkáme, že:

- vlákna c jsou validní, když: $\forall e, e' \in R\, e \neq e' \Rightarrow thread(e) \neq thread(e')$,
- množiny zámků c jsou validní, když: $\forall e, e' \in R\, e \neq e' \Rightarrow guards(e) \cap guards(e') = \emptyset$,

kde pro každou $e \in R$, $threads(e)$, resp. $guards(e)$, značí první, resp. druhou složku ohodnocení hrany e. Cykly, které algoritmus vyhodnotí jako validní (potenciální chyba), jsou ty, pro které platí, že jejich vlákna i množiny zámků jsou validní (tzn. vlákna v cyklu se liší a množiny zámků se nepřekrývají). Takto zůstanou pouze cykly, které nebyly tvořeny jedním vláknem a nebyly chráněny výše obdrženým zámkem.

Druhým rozšířením Goodlock algoritmu je rozšíření, které zamezuje detekci cyklů z částí programu (segmentů), které nemůžou běžet konkurentně. Instrumentovaný kód musí poskytovat informaci o událostech vytvoření vlákna (start) a ukončení vlákna (join). Analyzátor poté ze sekvence událostí vytvoří nový orientovaný graf, který značí to, jaké segmenty kódu byly vykonány před jinými. Ohodnocení hran v grafu zámků je rozšířeno o informaci, ve kterém segmentu byly zámky zamknuty. Vyhodnocení validních cyklů poté obsahuje kontrolu, jestli zamknutí zámků opravdu pochází ze dvou paralelně prováděných segmentů.

2.6.2 Algoritmy pro detekci časově závislých chyb nad daty

Algoritmus FastTrack

Algoritmus FastTrack patří do skupiny precizních algoritmů pracující s vektorovou reprezentací času a relací Happens-before [13]. Návrh algoritmu obsahuje tzv. *epochy*, což jsou dvojice čas/vlákno s označením $c@t$, kde c je aktuální hodnota času vlákna t. Epochy lze reprezentovat jednou 32-bitovou hodnotou typu `integer`, kde prvních 8 bitů značí identifikátor vlákna t a dalších 24 bitů uchovává aktuální hodnotu c vlákna t. Podle relace Happens-before (dále pouze $\preceq$):

$$c@t \preceq V \;\; \Leftrightarrow c \leq V(t),$$

kde V je vektorový čas.

Aktuální stav analyzátoru definujme jako čtveřici (C, L, R, W), kde:

- C je množina vektorových časů jednotlivých vláken (C_t značí vektorový čas vlákna t).
- L je množina vektorových časů všech zámků (L_m značí vektorový čas posledního odemknutí zámku m).
- R_x označuje buď epochu posledního zápisu do proměnné x (pokud všechna ostatní čtení $R'_x \preceq R_X$) nebo vektorový čas všech čtení proměnné x.
- W_x značí epochu posledního zápisu do proměnné x.

Tento stav je aktualizován při výskytu událostí $Rd(t, x)$, $Wr(t, x)$, $acq(t, l)$, $rel(t, l)$, $fork(t_i, t_j)$ a $join(t_i, t_j)$ a zároveň s přesunem do následujícího stavu dochází ke kontrole, jestli aktuální událost nevede k data race chybě.

Ve výpisu 2.6 je ukázána zjednodušená verze algoritmu s funkcemi `read` a `write`, které popisují změny vnitřního stavu analyzátoru při čtení/zápisu z/do konkrétní proměnné.

```
class ThreadState {
    int threadId;
    int C[];
    int epoch; // epoch == C[threadId]
}

class VarState {
    int W, R; // epoch of last write/read operation
    int Rvc[]; // used if R == READ_SHARED
}

class LockState {
    int L[];
}

void read(VarState x, ThreadState t) {}
    if (x.R == t.epoch) return; // Same Epoch
    // Check for RAW data race error
    if (x.W > t.C[ThreadId(x.W)]) error;
    // update read state
    if (x.R == READ_SHARED) { // Shared -> update vector clock
        x.Rvc[t.threadId] = t.epoch;
    } else {
        if (x.R <= t.C[ThreadId(x.R)]) { // Exclusive read
            x.R = t.epoch;
        } else { // Share read
            if (x.Rvc == null)
                x.Rvc = newClockVector(); // init vector clock
            x.Rvc[ThreadId(x.R)] = x.R;
            x.Rvc[t.threadId] = t.epoch;
            x.R = READ_SHARED;
        }
    }
}

void write(VarState x, ThreadState t) {
    if (x.W == t.epoch) return; // Same Epoch

    // Check for WAW data race error
    if (x.W > t.C[ThreadId(x.W)]) error;

    // Check for WAR data race error
    if (x.R != READ_SHARED) { // Shared
        if (x.R > t.C[ThreadId(x.R)]) error;
    } else { // Exclusive
        // Compare vector clocks
```

```
            if (x.Rvc[u] > t.C[u] for any u) error;
        }
        x.W = t.epoch; // update write state
    }
```

Kód 2.6: Pseudokód algoritmu FastTrack

Data race typu WAW (Write after Write) algoritmus detekuje tak, že ke každé sdílené proměnné si pamatuje epochu W_x posledního zápisu do ní a při následujícím zápisu je tato epocha porovnána s vektorovým časem C_i, vlákna t_i, které zapisuje do proměnné. Pokud $W_x \preceq C_i$, tak k data race chybě nedošlo. V opačném případě se jedná o chybu a ta je analyzátorem zaznamenána.

Data race typu RAW (Read after Write) jsou detekovány stejným způsobem jako chyby typu WAW.

Data race typu WAR (Write after Read) je komplikovanější z toho důvodu, že Read operace nemusí být uspořádány dle relace Happens-before ani v programu, který chybu neobsahuje. Zápis do proměnné by tak mohl být v kolizi se čtením proměnné z jakéhokoliv vlákna (ne pouze tím posledním). Proto si nestačí pamatovat pouze epochu R_x posledního čtení proměnné, ale vektorový čas všech posledních čtení ze všech vláken. Tento vektorový čas ovšem není nutné uchovávat pro všechny proměnné. K velkému množství proměnných je přistupováno buď:

- Pouze z jednoho vlákna → čtení z nich je úplně uspořádáno podle relace Happens-before.
- Pouze v kritické sekci strážené zámkem L → čtení z nich je uspořádáno podle relace Happens-before pomocí synchronizace nad tímto zámkem.

V těchto případech algoritmus FastTrack při čtení z proměnné x pouze aktualizuje hodnotu epochy R_x a pokračuje v exekuci programu. Pokud ovšem nastane situace, že dvě po sobě jdoucí čtení z proměnné x nejsou podle relace Happens-before uspořádány (neplatí $R_x \preceq C_i$), začne se uchovávat celý vektorový čas pro všechna vlákna, které k proměnné x přistupují. K těmto situacím ovšem dle [13] dochází v praxi minimálně.

Algoritmus tedy oproti jiným algoritmům (např. D^+_{JIT} [25]) neuchovává vektorový čas pro všechna vlákna a všechny proměnné, a tím snižuje paměťové i časové nároky na analýzu. Porovnání epochy s vektorovým časem totiž proběhne v čase $\mathcal{O}(1)$ oproti porovnání dvou vektorů, které probíhá se složitostí $\mathcal{O}(n)$.

Algoritmus AtomRace

Algoritmus AtomRace je algoritmus vyvinutý na FIT VUT v Brně. Algoritmus je založen *přímo* na nízkoúrovňové definici data race chyby. Ta říká, že data race je chyba vyskytující se v případě, že dvě nebo více vláken přistupují ke sdílené proměnné, alespoň jeden z přístupů je zápis a neexistuje žádná synchronizace, která by těmto přístupům bránila v tom, aby běžely současně [17]. Detekce data race chyb spočívá v nalezení situace, kdy takové přístupy nastanou.

Při instrumentaci programu, který má být tímto algoritmem analyzován, je nutné všechna čtení/zápisy z/do proměnné nahradit sekvencí instrukcí *BeforeAccessEvent, i, AfterAccessEvent*, kde i je instrukce samotného čteni/zápisu a *BeforeAccessEvent/AfterAccessEvent* jsou speciální instrukce přidány před/za i. Tyto instrukce musí obsahovat:

1. Identifikátor vlákna, kterému přístup patří.
2. Informace o místu v paměti, ke kterému se v rámci dané instrukce přistupuje.
3. Typ přístupu (čtení/zápis).
4. Místo v aplikaci, kde k přístupu došlo.

Při analýze programů následně hledáme takové kolizní atomické sekce, z nichž alespoň jedna je zápis. V praxi je ovšem vykonáni sekvence instrukcí *BeforeAccessEvent, i, AfterAccessEvent* většinou velice rychlé, a tudíž pravděpodobnost zpozorování chyby (kolizních atomických sekcí) je nízká. Proto se využívají techniky *vkládání šumu* ke zvýšení této pravděpodobnosti. V kapitole 2.4.2 byla popsána základní myšlenka těchto technik a způsob její možné realizace v jazyce C#.

V kódu 2.7 lze vidět pseudokód algoritmu AtomRace. Každé sledované proměnné `v` se při inicializaci nastaví hodnota `Access(v)` na `null`. Před každým přístupem k proměnné je hodnota `Access(v)` aktualizována na aktuální vlákno a lokaci. Pokud ovšem tato hodnota není prázdná (rovna `null`), je zkontrolována podmínka současného přístupu a vyhodnocena přítomnost data race chyby.

```
Inicializace:
∀v ∈ SharedVariables : Access(v) = null;

Aktualizace:
switch (AtomRaceEvent) {
    case: beforeAccessEvent(v, loc)
        if (Access(v) == null) then
            Access(v) = (t_current, loc);
        else if ((getMode(Access(v).loc) == write) ||
            (getMode(loc) == write)) then
            DATA RACE DETECTED
    case: afterAccessEvent(v, loc)
        if (Access(v).t == t_current) then
            Access(v) = null;
}
```

Kód 2.7: Pseudokód algoritmu AtomRace

Algoritmus Eraser

Algoritmus Eraser patří do skupiny neprecizních algoritmů. Je oblíbený pro svou jednoduchost a nízkou časovou i paměťovou složitost. Algoritmus kontroluje to, jestli všechny přístupy ke sdíleným prostředkům dodržují předem definovaná pravidla ohledně použití zámků (tzv. *locking discipline*). V nejjednodušší podobě může pravidlem být to, že každý přístup ke sdílené proměnné je chráněn nějakým (jakýmkoliv zámkem).

Pro každou sdílenou proměnnou x si algoritmus uchovává množinu $C(x)$ kandidátních zámků. Zámek $l \in C(x)$ právě tehdy, když každé vlákno t při přístupu k proměnné x mělo ve svém držení zámek l. Po inicializaci nové proměnné je $C(x)$ rovno množině všech možných zámků. Při každém přístupu vlákna t k proměnné x je hodnota $C(x)$ aktualizována jako průnik $C(x)$ a zámků držených vláknem t v době přístupu k proměnné. Pokud je po

aktualizaci $C(x)$ rovno $\emptyset$, tak proměnná x není konzistentně chráněna žádným zámkem a analyzátor produkuje varování o potenciální data race chybě v programu. Podmínka pokaždé držet zámek při přístupu k proměnné není vždy nutná. Vynálezci tohoto algoritmu identifikovali tři případy, kdy nemusí být tato podmínka splněna, a přesto není porušena atomicita programu.

1. Inicializace proměnné – sdílené proměnné jsou často inicializovány bez toho, aby byly chráněny zámkem.
2. Readonly proměnné – mnoho proměnných je inicilizováno, ale později je z nich pouze čteno (readonly), tyto proměnné nemusí být chráněny zámkem.
3. Read-Write zámky – tento druh zámku umožňuje zápis pouze jednomu vláknu (není nutné, aby ostatní vlákna držely ten stejný zámek, protože z proměnné vždy pouze čtou).

Existuje mnoho rozšíření tohoto algoritmu, které vylepšují jeho určité vlastnosti jako např. algoritmus GoldiLocks [11] či RaceTrack, který kombinuje oba přístupy LockSet-base algoritmů s těmi využívající vektorový čas [29].

Kapitola 3

Návrh systému pro detekci chyb uváznutí a časově závislých chyb nad daty v paralelních programech

V této kapitole jsou popsány požadavky na instrumentační framework a dynamický analyzátor. Kapitola obsahuje popis jednotlivých prvků systému a komunikační rozhraní mezi nimi. Dále je popsán návrh instrumentačního frameworku za pomocí nástroje PostSharp a možnosti provedení instrumentace pomocí tohoto frameworku. Na závěr kapitoly je popsán návrh dynamických analyzátorů pro detekci uváznutí a časově závislých chyb nad daty dle kapitoly 2.6.

3.1 Specifikace požadavků

Specifikace požadavků se skládá ze tří částí (obecné požadavky napříč systémem, požadavky na instrumentační framework a požadavky na dynamické analyzátory pro detekci chyb).

1. Obecné požadavky na všechny části systému:
   1.1. Systém musí být spustitelný na platformě .NET Framework.
   1.2. Lze jednoduše nakonfigurovat typ analýzy (detekce uváznutí či data race).
   1.3. Systém poskytuje užitečné informace programátorovi o probíhané analýze.
   1.4. Funkcionalita musí být ověřena pomocí automatických testů.
2. Funkční požadavky na instrumentační framework:
   2.1. Instrumentační framework umožňuje snadnou konfiguraci instrumentace programů pro testování.
   2.2. Instrumentační framework umožňuje instrumentovat konkrétní funkci, především pak:
      - lze rozeznat vlákno, ze kterého je volána,
      - je umožněna práce s atributy dané funkce,
      - lze přidat kód před volání konkrétní funkce,
      - lze přidat kód po navrácení z funkce.

2.3. Instrumentační framework umožňuje instrumentovat čtení/zápis z/do konkrétní proměnné, především pak:
   - lze rozeznat vlákno, ze kterého je k proměnné přistupováno,
   - lze jednoznačně identifikovat místo v programu, kde se proměnná nachází,
   - lze získat její aktuální hodnotu,
   - lze přidat kód před přístup k paměti,
   - lze přidat kód po dokončení přístupu k paměti.

2.4. Instrumentační framework provádí instrumentaci s žádnou či pouze minimální změnou zdrojového kódu.

3. Funkční požadavky na dynamické analyzátory:

   3.1. Jednoduchou konfigurací lze zvolit konkrétní analyzátor pro daný běh programu.

   3.2. Analyzátor dokáže zpracovávat události přijímané z instrumentovaného programu.

   3.3. Analyzátor dokáže ve sledovaném programu detekovat druh chyb, pro který je určen.

   3.4. Analyzátor dokáže poskytnout výstup s výsledky analýzy, především detekované chyby a popis událostí, které k chybě vedly.

3.2 Prvky systému a jejich komunikace

Hlavními prvky systému by měly být dvě na sobě nezávislé části, které mezi sebou komunikují pomocí předem definovaných zpráv. Jedna část (instrumentační framework) by se měla starat o korektní instrumentaci programu pro potřeby dynamických analyzátorů. Instrumentační framework je zodpovědný za to, že jím nainstrumentovaný program zasílá zprávy o důležitých událostech (z pohledu analýzy), které za jeho běhu nastanou. Druhá část systému (dynamický analyzátor) by měla v systému fungovat jako konzument již zmíněných zpráv a jeho zodpovědností by mělo být jejich korektní zpracování a analýza. Výsledky analýzy by měl poskytnout jako výstup po skončení analýzy.

3.2.1 Komunikační rozhraní mezi prvky systému

Níže je sepsán seznam hlavních událostí, u kterých je nutné, aby instrumentovaný program zasílal zprávy o jejich výskytu (zodpovědnost instrumentačního frameworku).

Vytvoření nového vlákna – událost vyvolána při vzniku nového vlákna. Zpráva musí obsahovat identifikátor nového vlákna. Reprezentována třídou `ThreadCreatedEvent`.

Zánik existujícího vlákna – událost vyvolána při zániku (ukončení) existujícího vlákna. Zpráva musí obsahovat identifikátor vlákna. Reprezentována třídou `ThreadCancelledEvent`.

Zamknutí zámku – událost vyvolána při pokusu některého z vláken obdržet konkrétní zámek. Zpráva musí obsahovat identifikátor vlákna, které chce operaci provést a identifikátor zámku, který chce obdržet. Zpráva musí být odeslána před pokusem o obdržení zámku i po dokončení této operace (s případnou informací, jestli operace proběhla úspěšně). Reprezentována třídami `(Before/After)LockAcquiredEvent`.

Uvolnění zámku – událost vyvolána při tom, co jedno z běžících vláken uvolní některý ze zámků, které má ve svém držení. Zpráva musí obsahovat identifikátor vlákna, které

chce operaci provést a identifikátor zámku, který chce uvolnit. Zpráva musí být odeslána před pokusem o obdržení zámku i po dokončení této operace (s případnou informací, jestli operace proběhla úspěšně). Reprezentována třídami `(Before/After)LockReleasedEvent`.

Čtení z proměnné – událost vyvolána při pokusu jakéhokoliv vlákna číst instrumentovanou proměnnou. Zpráva musí obsahovat identifikátor vlákna, které chce operaci provést a identifikátor proměnné, ze které chce vlákno číst. Zpráva musí být odeslána před pokusem o čtení z proměnné i po dokončení čtení. Reprezentována třídami `(Before/After)MemoryReadEvent`.

Zápis do proměnné – událost vyvolána při pokusu jakéhokoliv vlákna zapisovat do instrumentované proměnné. Zpráva musí obsahovat identifikátor vlákna, které chce operaci provést a identifikátor proměnné, do které chce vlákno zapisovat. Zpráva musí být odeslána před pokusem o zápis do proměnné i po dokončení zápisu. Reprezentována třídami `(Before/After)MemoryWriteEvent`.

3.3 Návrh instrumentačního frameworku

V sekcích 3.1 a 3.2 jsou rozebrány požadavky na instrumentaci programu, který bude testován. Dle těchto požadavků byl navrhnut samotný instrumentační framework, který bude požadavky splňovat. Instrumentační framework bude implementován jako samostatná knihovna a bude využívat deklarací zpráv z komunikačního rozhraní pro inicializaci objektů zpráv.

Při rozhodování o tom, jaký přístup k samotné instrumentaci zvolit, bylo důležité brát v potaz:

- stabilitu a jednoduchost výsledného instrumentačního nástroje,
- náročnost budoucí údržby a rozšiřitelnosti,
- škálovatelnost daného frameworku pro velké projekty.

Především z těchto důvodů bylo v průběhu návrhu systému rozhodnuto, že při implementaci frameworku bude využit nástroj PostSharp[1], jehož základní popis byl uveden již v kapitole 2.5.2. Ostatní nástroje popsané v této kapitole byly sice zajímavými alternativami, ale buď nesplňovaly veškeré požadavky pro instrumentaci, kterou je třeba v rámci této práce provést (např. uměly instrumentovat pouze metody, a ne přístupy do paměti), nebo již přestaly být vyvíjeny, a tak jejich využití sice mohlo splnit účely této práce, ale pro budoucí vývoj nástroje by volba takových nástrojů mohla být spíše komplikací a instrumentační framework by musel být brzy refaktorován.

Dalším možným přístupem k instrumentaci v .NET Framework by byla implementace knihovny, která by přímo modifikovala zkompilovaný kód v CIL (viz 2.5.1). Tento přístup nakonec nebyl zvolen z toho důvodu, že implementační náročnost této metody by byla značně vyšší než jiné přístupy. Možnosti ladění při vývoji jsou taktéž značně omezeny. Tento přístup vyžaduje důkladnou znalost CIL, což by mohl být problém při údržbě nástroje v budoucnu dalšími vývojáři, kteří by nejprve museli nastudovat psaní kódu v CIL, aby mohli framework rozšířit o další funkce. Existují sice knihovny (např. Mono.Cecil či dnlib[2]) , které práci s CIL usnadňují, i přesto ale bylo vyhodnoceno, že tento přístup není ideální volbou v rámci této práce.

[1] https://www.postsharp.net/
[2] https://github.com/0xd4d/dnlib

Nástroj PostSharp je komplexní nástroj sloužící ke zjednodušování psaní kódu v jazyce C#. Nástroj je distribuován jako NuGet balíček a v rámci této práce je využita jeho verze 6.8. Jeho robustnost a maturita (vyvíjen více než 10 let) předesílá stabilitu řešení postaveném na tomto nástroji. Je předpokladem, že nástroj bude pokračovat ve svém zlepšování, bude reagovat na vývoj technologií v .NET ekosystému, a aplikace implementované s jeho pomocí mohou být jednodušeji udržitelné.

PostSharp podporuje tvorbu tzv. aspektů. Právě tyto aspekty budou základním stavebním kamenem instrumentačního frameworku implementovaném v rámci této práce. Aspekt si lze představit jako kus kódu, který lze vložit na jiné (předem určené) místo v programu. V nástroji PostSharp existuje několik druhů aspektů. Každý z nich je představován abstraktní třídou ve jmenném prostoru `PostSharp.Aspects`. Mezi základní typy aspektů, které budou využívány v instrumentačním frameworku, patří následující aspekty (jejich název je shodný s názvem abstraktní třídy).

`OnMethodBoundaryAspect`[3] – aspekt, díky kterému je možné instrumentovat volání metody, na kterou je aplikován. Instrumentace se provádí přetížením metod `OnEntry` (kód provedený před voláním funkce), `OnSuccess` (provedený před návratem z funkce), popř. `OnException` (kód je zavolán, pokud ve funkci nastane výjimka). Metody `OnEntry` či `OnSuccess` přijímají jako parametr objekt, který obsahuje informace o konkrétním volání instrumentované metody (hodnoty argumentů, instanci objektu, ze kterého je volána, návratovou hodnotu, ...). Tato třída bude sloužit jako bázová třída pro všechny implementované aspekty, které budou mít za úkol instrumentovat konkrétní metodu.

`LocationInterceptionAspect`[4] – aspekt, díky kterému je možné instrumentovat přístup k atributu či vlastnosti nějaké třídy. Instrumentace se provádí přetížením metod `OnGetValue`, resp. `OnSetValue`. Metody `OnGetValue` či `OnSetValue` přijímají jako parametr objekt, který obsahuje informace o lokaci, ke které je přistupováno (identifikátor, instance objektu, kterému atribut či vlastnost náleží, ...). Tato třída bude sloužit jako bázová třída pro všechny implementované aspekty, které budou mít za úkol instrumentovat přístup k atributům a vlastnostem tříd.

Každý z aspektů bude reprezentován svou vlastní třídou, která bude dědit jednu z výše popsaných tříd a přetěžovat určité její metody. Každý z aspektů bude zodpovědný za zasílaní některé ze zpráv popsaných v sekci 3.2.1. V tabulce 3.1 lze vidět návrh jednotlivých aspektů. Především to, z jaké bázové třídy budou dědit a za zasílání jakých zpráv bude daný aspekt zodpovědný.

[3] https://doc.postsharp.net/t_postsharp_aspects_onmethodboundaryaspect
[4] https://doc.postsharp.net/t_postsharp_aspects_locationinterceptionaspect

Název aspektu	Bázová třída	Přetížená metoda	Generovaná událost
LockAcquireAspect	OnMethodBoundaryAspect	OnEntry	BeforeLockAcquiredEvent
LockAcquireAspect	OnMethodBoundaryAspect	OnSuccess	AfterLockAcquiredEvent
LockReleaseAspect	OnMethodBoundaryAspect	OnEntry	BeforeLockReleasedEvent
LockReleaseAspect	OnMethodBoundaryAspect	OnSuccess	AfterLockReleasedEvent
MemoryAccessAspect	LocationInterceptionAspect	OnGetValue	BeforeMemoryReadEvent AfterMemoryReadEvent
MemoryAccessAspect	LocationInterceptionAspect	OnSetValue	BeforeMemoryWriteEvent AfterMemoryWriteEvent
MAWithNoiseAspect	LocationInterceptionAspect	OnGetValue	BeforeMemoryReadEvent AfterMemoryReadEvent
MAWithNoiseAspect	LocationInterceptionAspect	OnSetValue	BeforeMemoryWriteEvent AfterMemoryWriteEvent
ThreadCancelAspect	OnMethodBoundaryAspect	OnSuccess	ThreadCreatedEvent
ThreadCreateAspect	OnMethodBoundaryAspect	OnSuccess	ThreadCancelledEvent

Tabulka 3.1: Návrh jednotlivých aspektů pro instrumentaci.

Přiřazení aspektů ke konkrétním metodám/vlastnostem/atributům je možné provést vícero způsoby. Jeden ze způsobů je ten, že aspekt přiřadíme k metodě pomocí atributu dané metody. Příklad takového přiřazení je možné vidět v kódu 3.1. Tento způsob přiřazení nevyhovuje specifikovaným požadavkům ze dvou důvodů.

1. Pro instrumentaci by byl vyžadován velký zásah do zdrojového kódu, protože ke všem metodám, které by bylo nutné instrumentovat, by bylo potřeba přiřadit konkrétní atribut.
2. Nelze takto instrumentovat funkce z již zkompilovaných knihoven (např systémových). To by znemožňovalo např. instrumentaci spousty základních synchronizačních primitiv.

```
[OnFooMethodBoundaryAspect]
static void Foo(string[] args)
{
    // Do Foo
}
```

Kód 3.1: Přiřazení aspektu k metodě Foo pomocí atributu

Druhým možným způsobem, jak přiřadit aspekty ke konkrétním metodám/vlastnostem/atributům, je implementovat rozhraní `PostSharp.Aspects.IAspectProvider` a jeho metodu `ProvideAspects`. Uvnitř této metody lze pomocí reflexe jazyka C# vyhledat konkrétní deklarace metod/atributů a přiřadit k nim zvolený aspekt. Metoda vrací kolekci takto vytvořených aspektů. Danou implementaci rozhraní `IAspectProvider` pak stačí zmínit v souboru `AssemblyInfo.cs` modulu, který chceme instrumentovat či v konfiguračním souboru `postsharp.config` (více v kapitole 4). Tímto způsobem bude prováděna instrumentace programu v instrumentačním frameworku. Bude implementována jedna implementace rozhraní `IAspectProvider`, která bude instrumentovat program (přiřazovat aspekty) dle konfiguračního souboru.

Samotná instrumentace je nástrojem PostSharp provedena buď při překladu programu nebo z příkazové řádky programem `postsharp-cli.exe`, který přijímá jako první parametr cestu k již zkompilovanému souboru (`.dll` či `.exe`) a jako další parametr přijímá konkrétní implementaci rozhraní `IAspectProvider`, podle kterého je instrumentace binárního souboru provedena.

3.4 Návrh dynamických analyzátorů pro detekci chyb

Dynamický analyzátor pro detekci chyb uváznutí je část systému, která je zodpovědná za korektní příjem událostí z nainstrumentovaného programu a nad těmito událostmi exekucí algoritmu pro detekci chyb. Tato část systému bude implementována, podobně jako instrumentační část, jako samostatná knihovna obsahující analyzátory pro detekce jednotlivých druhů chyb.

Všechny analyzátory ve výsledném systému budou dědit z abstraktní třídy `AnalyserBase`, která obsahuje virtuální metody pro příjem zpráv z nainstrumentovaného programu. Konkrétní analyzátor pak implementuje ty metody, jejichž zprávy potřebuje pro svou analýzu. V kódu 3.2 lze vidět deklaraci jedné virtuální metody v bázové třídě pro dynamické analyzátory.

```
public abstract class AnalyserBase
{
    ...
    public virtual void OnThreadCreated(ThreadCreatedEvent eventInfo)
    {
    }
    ...
}
```

Kód 3.2: Virtuální metoda pro příjem zprávy o události vytvoření nového vlákna

Pro výběr analyzátoru/ů pro konkrétní běh programu bude implementováno rozhraní `IAnalysersProvider`, s metodou `GetProviders()`, která bude vracet kolekci objektů třídy `AnalyserBase`. Těmto analyzátorům budou zasílány zprávy z instrumentačního frameworku.

Dynamický analyzátor pro detekci chyb uváznutí

Dynamický analyzátor pro detekci chyb uváznutí implementuje algoritmus Goodlock popsaný v kapitole 2.6.1. Pro fungování tohoto algoritmu musí analyzátor přijímat zprávy: BeforeLockAcquiredEvent, BeforeLockReleasedEvent a ThreadCreatedEvent. Zpracováním těchto zpráv vytváří graf zámků a nad tímto grafem následně spouští běh algoritmu Goodlock pro hledání cyklů v grafu. Výstup analyzátoru slouží především jako informace pro programátora o chybách (i těch potenciálních) ve sledovaném systému.

Dynamický analyzátor pro detekci časově závislých chyb nad daty

V systému budou fungovat dva analyzátory pro detekci časově závislých chyb nad daty. Jeden z nich bude implementovat algoritmus FastTrack a druhý algoritmu AtomRace (oba popsány v kapitole 2.6.2).

Jak se liší oba algoritmy, tak se liší i události, o kterých potřebují být oba analyzátory informovány. Zatímco analyzátoru implementujícímu algoritmus AtomRace stačí pro

jeho práci přijímat zprávy o událostech (Before/After)MemoryReadEvent a (Before/After)MemoryWriteEvent, algoritmus FastTrack vyžaduje příjem zpráv o událostech AfterMemoryWriteEvent, AfterMemoryReadEvent, AfterLockAcquiredEvent, BeforeLockReleasedEvent a ThreadCreatedEvent. Analyzátory mohou běžet paralelně i odděleně. Jejich výstupy by měly být podobné a měly by programátorovi říct, v jaké situaci v programu došlo (či může dojít) k časově závislé chybě nad daty.

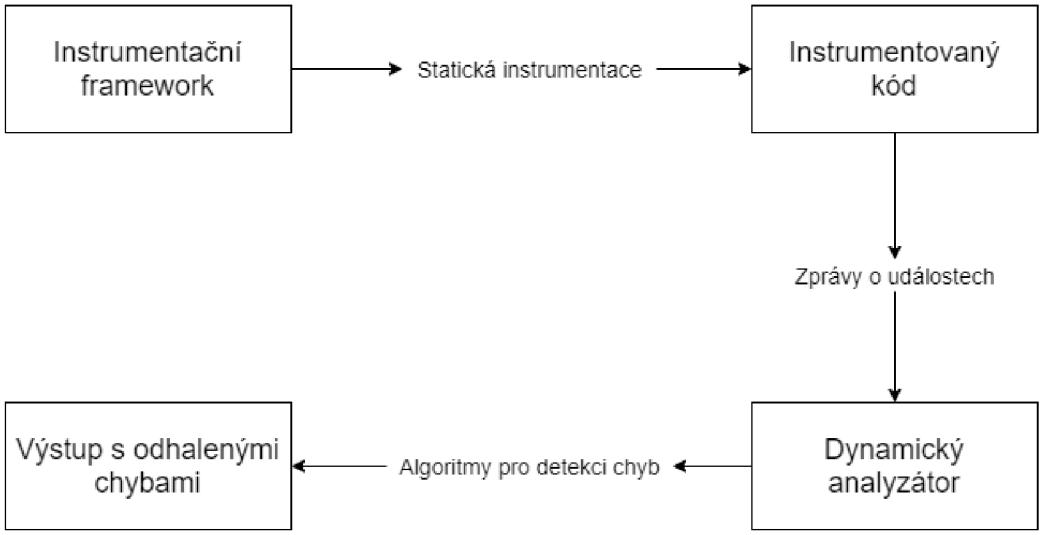
3.5 Shrnutí návrhu

Systém vyvíjený v rámci této práce se bude skládat ze dvou hlavních částí – instrumentačního frameworku a knihovny dynamických analyzátorů. Každá z částí bude implementována jako samostatná knihovna v jazyce C#. Části budou mezi sebou komunikovat pomocí zasílání zpráv. Zprávy samotné budou definovat jednotné komunikační rozhraní.

Instrumentační framework bude zodpovědný za korektní instrumentaci testovaného programu tak, aby byl instrumentovaný program schopen zasílat zprávy o událostech důležitých z hlediska analýzy pomocí definovaného komunikačního rozhraní.

Dynamický analyzátor zodpovídá za korektní příjem zpráv z nainstrumentovaného programu, zpracování těchto zpráv a následnou analýzu běhu programu. Hlavní úlohou a cílem analyzátoru je detekce chyb (i potenciálních) v testovaném programu a poskytnutí programátorovi co nejpodrobnější popis chyby pro její snadnou opravu.

Na obrázku 3.1 lze vidět zjednodušený diagram částí systému, hlavní úlohy jednotlivých částí a směr komunikace mezi nimi.



Obrázek 3.1: Zjednodušený diagram návrhu systému

Kapitola 4

Implementace instrumentace a dynamických analyzátorů

V následující kapitole je popsána implementační část celé práce. Detailně je rozebrán způsob instrumentace testovaného programu, způsob komunikace jednotlivých částí systému i práce jednotlivých analyzátorů. Vysvětlen je také způsob konfigurace nástroje při reálném použití a technologie, které byly při vývoji využity.

4.1 Implementace komunikace mezi částmi systému

Komunikace mezi jednotlivými částmi systému je prováděna tak, že instrumentovaný kód posílá události o tom, k jaké situaci v programu došlo a tyto události jsou zpracovány příslušným zpracovatelem události.

Každá událost je reprezentována třídou dědící z bázové abstraktní třídy `EventBase`. Všechny třídy reprezentující události lze nalézt v projektu `Taipan.Core` ve složce `Events`. Strukturu tříd lze vidět v následujícím stromě.

- `EventBase` – bázová třída pro všechny události
  - `ThreadEventBase` – obsahuje identifikátor vlákna
    - `LockEventBase` – obsahuje identifikátor zámku
      - `AfterLockAcquiredEvent` – zaslaná po zamknutí zámku vláknem
      - `BeforeLockAcquiredEvent` – zaslaná před zamknutím zámku vláknem
      - `BeforeLockReleasedEvent` – zaslaná před uvolněním zámku vláknem
    - `MemoryAccessEventBase` – obsahuje informace o místu v paměti
      - `AfterMemoryReadEvent` – zaslaná po čtení z paměti
      - `AfterMemoryWriteEvent` – zaslaná po zápisu do paměti
      - `BeforeMemoryReadEvent` – zaslaná před čtením z paměti
      - `BeforeMemoryWriteEvent` – zaslaná před zápisem do paměti

Obrázek 4.1: Stromová struktura tříd reprezentující události

Zpracování událostí přijímaných z nainstrumentovaného programu probíhá pomocí rozhraní `IEventHandler`, které deklaruje jedinou metodu `void HandleEvent(TEvent eventInfo)`, kde `TEvent` je generický parametr rozhraní, jehož typ musí dědit ze třídy `EventBase`. Všechny třídy zodpovědné za zpracování událostí jsou vytvořeny ve jmenném prostoru `Taipan.Analysers.Handler`. Zpracování událostí proběhne tak, že pomocí instance rozhraní `IAnalysersProvider` je získána kolekce všech aktivních analyzátorů a těm je daná událost přeposlána.

Po spuštění programu je volána metoda `Initialize` třídy `TaipanServiceLocator`. Třída plní roli IoC (Inversion of Control) kontejneru. V metodě `Initialize` jsou zaregistrovány veškeré závislosti, které jsou pro správný běh nástroje Taipan potřeba, především se jedná o:

- Implementace rozhraní `IAnalysersProvider` (viz kapitola 3.4).
- Registrace implementací rozhraní `IEventHandler` pro jednotlivé typy zpráv.

Metoda `GetService` třídy `TaipanServiceLocator` umožňuje získat zaregistrovanou implementací pro typ, který obdrží jako parametr. Toho je využito především v metodě `SendEvent` třídy `EventsHelper`, která přijímá jako parametr instanci třídy `EventBase` a vyhledá příslušného zpracovatele zprávy, kterému zprávu pošle ke zpracování.

4.2 Instrumentace programu

V kapitole 3.3 je odůvodněn výběr nástroje PostSharp pro implementaci instrumentace v rámci práce. Na tuto kapitolu je v následující sekci navázáno. Veškeré aspekty pro nástroj PostSharp, které byly v rámce práce vyvinuty, lze nalézt ve složce `/src/Taipan/Aspects`. Pro instrumentaci byl zvolen přístup využití rozhraní `IAspectProvider`. Jedinou implementací tohoto rozhraní je v nástroji Taipan třída `TaipanAspectProvider`. *Tato třída je vstupním bodem veškeré instrumentace v nástroji Taipan.*

Třída `TaipanAspectProvider` dědí z bázové třídy `AssemblyLevelAspect`[1]. To znamená, že tento aspekt je uplatňován na celý modul (assembly). Konfigurace samotné instrumentace a přiřazení tohoto aspektu k assembly je popsáno v sekci 4.4. Ve třídě `TaipanAspectProvider` je implementována metoda `ProvideAspects`, která přijímá jako parametr assembly, pro které má být instrumentace provedena a vrací kolekci instancí tříd `AspectInstance`, kde každá z instancí značí instrumentaci jedné metody/atributu. Po přečtení konfigurace ze souboru `postsharp.config` vytváří výslednou kolekci aspektů voláním pomocných metod ze statické třídy `AspectInstancesHelper`.

4.2.1 Instrumentace vláken

Instrumentace metod třídy `System.Threading.Thread` probíhá voláním metody `AspectInstancesHelper.GetThreadAspectInstances`, která provede instrumentaci metod zodpovědných za vytvoření vlákna (metoda `Start`) a zánik vlákna (metody `Abort` a `Join`).

Metodě `Start` je přiřazen aspekt `ThreadCreateAspect`, jehož funkcí je to, že po úspěšném volání této funkce vytvoří událost `ThreadCreateEvent`.

Metodám `Abort` a `Join` je přiřazen aspekt `ThreadCancelledAspect`, jehož funkcí je to, že po úspěšném volání této funkce vytvoří událost `ThreadCancelledEvent`.

[1] https://doc.postsharp.net/t_postsharp_aspects_assemblylevelaspect

4.2.2 Instrumentace synchronizačních primitiv

Instrumentace synchronizačních primitiv jazyka C# probíhá voláním metody `AspectInstancesHelper.GetSynchronizationAspectInstances`, která provede instrumentaci metod nejčastěji používaných pro synchronizaci. Instrumentovány jsou metody tříd `System.Threading.Monitor` a `System.Threading.Mutex`.

V tabulce 4.1 lze vidět instrumentované metody synchronizačních primitiv a jim přiřazené aspekty, resp. jejich názvy třídy. Instrumentace metod synchronizačních primitiv spočívá v odeslání událostí `BeforeAcquireLockEvent`, `BeforeReleaseLockEvent` a `AfterAcquireLockEvent`.

Název třídy	Název metody	Přiřazený aspekt
Monitor	Enter	MonitorEnterAspect
Monitor	TryEnter	MonitorTryEnterAspect
Monitor	Wait	MonitorWaitAspect
Monitor	Pulse	LockReleaseAspect
Monitor	PulseAll	LockReleaseAspect
Monitor	Exit	LockReleaseAspect
Mutex	WaitOne	MutexWaitAspect
Mutex	ReleaseMutex	LockReleaseAspect

Tabulka 4.1: Přiřazení aspektů metodám synchronizačních primitiv.

4.2.3 Instrumentace přístupů do paměti

Instrumentace přístupů do paměti probíhá voláním metody `AspectInstancesHelper.GetLocationSetAspectInstances`, která provede instrumentaci atributů třídy, která je metodě předána jako parametr `type`. Metoda provádí instrumentaci pro všechny atributy (privátní i veřejné), které třída deklaruje. Instrumentace se neprovádí pro atributy bázových tříd. Pro výběr atributů třídy se používají metody `GetFields` a `GetProperties` třídy `System.Type`.

Aspekty sloužící pro instrumentaci přístupu k atributům jsou reprezentovány třídami `MemoryAccesAspect` a `MemoryAccesWithNoiseAspect`.

Implementace techniky vkládání šumu

Metoda `AspectInstancesHelper.GetLocationSetAspectInstances` přijímá i nepovinný parametr `withNoise`, což je hodnota typu `bool`, která udává to, zdali má být na atributy třídy uplatněn aspekt, který přidává šum ke všem přístupům k atributu (třída `MemoryAccesWithNoiseAspect`).

Tato třída v přetížených metodách `OnGetValue` a `OnSetValue` mimo vytvoření událostí a její poslání příslušnému zpracovateli přidává šum, který v různých bězích programu zajišťuje různá proložení vláken, a tudíž zvyšuje pravděpodobnost nalezení chyby.

Metody vkládání šumu byly popsány v kapitole 2.4.2. Při implementaci nástroje Taipan byla zvolena technika vkládání náhodného šumu. Výhodou této techniky je její implementační a výpočetní nenáročnost (skvělý poměr mezi cenou a výkonem). Šum je vkládán v náhodné síle na náhodná místa voláním metody `ApplyRandomNoise` třídy `MemoryAccesWithNoiseAspect`, jejíž kód lze vidět v ukázce 4.1.

```
private void ApplyRandomNoise()
{
    var random = new Random();
    if (random.Next(10) < 5)
    {
        var sleepTime = random.Next(100, 300);
        Thread.Sleep(sleepTime);
    }
}
```

Kód 4.1: Metoda pro vložení náhodného šumu

4.3 Implementace dynamických analyzátorů

Dynamické analyzátory nástroje Taipan jsou zodpovědné za korektní zpracování událostí z nainstrumentovaného kódu a následnou analýzu těchto událostí. V této sekci je vysvětlena implementace analyzátorů, struktura tříd a formát výstupů, který jednotlivé analyzátory poskytují.

Všechny dynamické analyzátory lze najít v projektu `Taipan.Analysers`. Každý analyzátor je reprezentován třídou, která dědí z abstraktní třídy `AnalyserBase`. Tato třída obsahuje virtuální metody, přijímající události z nainstrumentovaného programu. Příklad metod lze vidět v kódu 4.2. Tento přístup byl při návrhu zvolen z toho důvodu, aby bylo zajištěno jednotné zpracování události. Ve třídách analyzátorů dědící z této třídy je nutno přetížit metody obsluhující události, které ke své práci analyzátor potřebuje a zprávy mu budou za běhu programu zasílány.

```
public virtual void OnThreadCreated(ThreadCreatedEvent eventInfo)
{
}

public virtual void OnAfterLockAcquired(AfterLockAcquiredEvent eventInfo)
{
}
```

Kód 4.2: Ukázka virtuálních metod třídy `AnalyserBase`

4.3.1 Detekce uváznutí

Pro detekci uváznutí byl zvolen algoritmus Goodlock (detailně představen v sekci 2.6.1). Analyzátor implementující tento algoritmus, je reprezentován třídou `GoodlockDeadlockAnalyser`. Analyzátor pro svou práci potřebuje z testovaného programu přijímat události `ThreadCreatedEvent`, `BeforeLockAcquiredEvent`, `BeforeLockReleasedEvent` a `RunAnalysisEvent`.

Analyzátor zpracováním prvních tří událostí vytváří graf zámků. Pro práci s grafem byla využita knihovna `QuickGraph`[2], která usnadňuje základní operace s grafem a poskytuje metody pro jeho průchod.

Po přijmutí události `RunAnalysisEvent`, která je z programu posílána periodicky, analyzátor spustí metodu `FindCycles` rozhraní `ICyclesDetector`, která v grafu zámku, jenž je

[2] https://www.nuget.org/packages/QuickGraph.NETStandard/

předán metodě jako parametr, vyhledá všechny validní cykly. Prohledávání grafu je provedeno pomocí prohledávání do hloubky (Depth First Search, DFS). Pro každý uzel v grafu jsou prohledány všechny výchozí hrany z tohoto uzlu a v nich jsou rekurzivně hledány validní cykly.

Pokud je nějaký validní cyklus v grafu nalezen, je vyvolána výjimka `DeadlockException`, ve které jsou informace o vláknech a zámcích, které cyklus tvořily.

Ukázky použití detekce uváznutí lze nalézt v sekci 5.3.

4.3.2 Detekce časově závislých chyb nad daty

Pro detekci časově závislých chyb nad daty byly zvoleny dva algoritmy popsány v sekci 2.6.2. Oba lze nalézt ve složce `/src/Taipan.Analysers/DataRaces` a v této sekci jsou sepsány jejich implementační detaily. Ukázky použití detekce časově závislých chyb nad daty lze nalézt v sekci 5.3.

Algoritmus AtomRace

Analyzátor implementující algoritmus AtomRace je reprezentován třídou `AtomRaceDataRaceAnalyser`.

Analyzátor pro svou práci potřebuje z testovaného programu přijímat události `BeforeMemoryReadEvent`, `BeforeMemoryWriteEvent`, `AfterMemoryReadEvent` a `AfterMemoryWriteEvent`.

Třída obsahuje privátní atribut `_currentAccesses` typu slovník (`System.Collection.Generic.Dictionary`), kde klíčem je instance objektu `LocationInfo`, která označuje místo v paměti, ke kterému je přistupováno a hodnotou je instance třídy `MemoryAccess` uchovávající informace o konkrétním přístupu (identifikátor vlákna, čtení/-zápis). Analyzátor po každé události `BeforeMemoryReadEvent` a `BeforeMemoryWriteEvent` vyhledá ve slovníku hodnotu s klíčem rovným objektu, ke kterému je aktuálně přistupováno. Pokud slovník hodnotu s tímto klíčem neobsahuje, je do něj hodnota přidána. Pokud ji obsahuje a alespoň jeden z přístupů (přístup ze slovníku či aktuální přístup) je zápis, je vyvolána výjimka `DataRaceException`, která obsahuje informace o obou konkurentních přístupech k objektu.

Algoritmus FastTrack

Analyzátor implementující algoritmus FastTrack (popsán v sekci 2.6.2) je reprezentován třídou `FastTrackDataRaceAnalyser`. Analyzátor pracuje s následujícími třídami:

- `Epoch` – reprezentuje jednu epochu (dvojici čas, vlákno)
- `LockState` – reprezentuje stav jednoho zámku (stav uzamknutí, vektorový čas)
- `ThreadState` – reprezentuje stav jednoho vlákna (identifikátor, epocha, vektorový čas)
- `VariableState` – reprezentuje stav jedné proměnné (poslední epocha zápisu/čtení, vektorový čas)

Analyzátor pro svou práci potřebuje z testovaného programu přijímat události `ThreadCreatedEvent`, `AfterMemoryReadEvent`, `AfterMemoryWriteEvent`, `AfterLockAcquiredEvent` a `BeforeLockReleasedEvent`.

V metodě obsluhují událost `ThreadCreatedEvent` se vytvoří reprezentace vektorového času zkopírováním vektorového času vlákna, ze kterého bylo původní vlákno vytvořeno.

Metoda obsluhující událost `AfterLockAcquiredEvent` má na starost aktualizaci vektorového času vlákna porovnáním s vektorovým časem zámku, který byl vláknem zamknut.

Metoda obsluhující událost `BeforeLockReleasedEvent` naopak aktualizuje vektorový čas zámku, který bude uvolněn.

Při zpracování událostí `AfterMemoryReadEvent` a `AfterMemoryWriteEvent` analyzátor nejprve ze svého vnitřního stavu získá instance tříd `ThreadState` a `VariableState` příslušného vlákna a proměnné, jichž se operace týká. Následně je volána metoda `TryRead` (pro čtení), resp. `TryWrite` (pro zápis) třídy `VariableState`, která přijímá jako parametr stav vlákna a v případě, že porovnáním tohoto stavu s posledními přístupy k proměnné je detekována chyba, je vyvolána výjimka `DataRaceException` s informacemi o konkurentních přístupech. Pokud chyba detekována není, je aktualizována hodnota posledního čtení či zápisu ve stavu konkrétní proměnné a program pokračuje ve své exekuci.

4.4 Konfigurace

Konifgurace nástroje Taipan je prováděna pomocí rozšířené konfigurace nástroje Postsharp. V článku [24] jsou popsány všechny možnosti konfigurace nástroje Postsharp. V této sekci budou ukázány pouze ty, které jsou potřebné pro správný běh nástroje Taipan.

Konfigurace instrumentace a dynamické analýzy je provedena pomocí vlastní sekce nástroje Taipan. XML definici schématu této sekce lze nalézt v souboru `TaipanConfigurationXMLSchema.xsd`. V praxi je v souboru `postsharp.config` nutné:

1. přidat definici jmenného prostoru pomocí atributu `xmlns:Taipan="https://pajda.fit.vutbr.cz/testos/taipan/-/blob/master/TaipanConfigurationXMLSchema.xsd"`
2. přidat element `<SectionType LocalName="Taipan"Namespace="Taipan"/>`
3. přidat sekci `Taipan` (viz kód 4.3)
4. přidat element `Multicast` (viz kód 4.4)

V konfigurační sekci Taipan lze nastavit hodnoty:

- `LogDirectory` – složka určena pro logování při využití `LockLoggingAnalyser`
- `Analysers` – čárkou oddělené názvy analyzátorů (povolené hodnoty jsou: `LockLoggingAnalyser`, `AtomRaceDataRaceAnalyser`, `FastTrackDataRaceAnalyser` a `GoodlockDeadlockAnalyser`)
- `DataRaceAnalysisClasses` – kolekce elementů `DataRaceAnalysisClass`, kde každý z elementů určuje jeden regulární výraz, podle kterého jsou v instrumentovaném assembly vyhledávány třídy pro instrumentaci (většinou název jmenného prostoru tříd, které chci instrumentovat)

V kódu 4.3 lze spatřit ukázku konfigurace z ukázkového programu `DataRaceDetectionSample`. Instrumentovány budou všechny atributy všech tříd jmenného prostoru `DataRaceDetectionSample` a bude použit analyzátor `AtomRaceDataRaceAnalyser`.

```
<Taipan:Taipan>
    <Taipan:LogDirectory>C:\Temp\Logs</Taipan:LogDirectory>
    <Taipan:Analysers>AtomRaceDataRaceAnalyser</Taipan:Analysers>
    <Taipan:DataRaceAnalysisClasses>
        <Taipan:DataRaceAnalysisClass>
            DataRaceDetectionSample
        </Taipan:DataRaceAnalysisClass>
    </Taipan:DataRaceAnalysisClasses>
</Taipan:Taipan>
```

Kód 4.3: Konfigurace nástroje Taipan pro detekci časově závislých chyb nad daty pomocí analyzátoru AtomRaceDataRaceAnalyser ve všech třídách jmenného prostoru DataRaceDetectionSample

Konečně pro výběr assemblies, pro která se má instrumentace provést slouží v konfiguračním souboru element `Multicast`[3]. V kódu 4.4 lze vidět konfigurace instrumentace pro assembly s názvem DataRaceDetectionSample.

```
<Multicast>
    <TaipanAspectProvider xmlns="clr-namespace:Taipan.Aspects;assembly:
    Taipan" AttributeTargetTypes="assembly:DataRaceDetectionSample" />
</Multicast>
```

Kód 4.4: Instrumentace assembly DataRaceDetectionSample

4.5 Správa kódu a CI/CD

Celý projekt byl vyvíjen pomocí vývojového prostředí Visual Studio 2019. Hlavním souborem pro toto prostředí je soubor `/Taipan.sln`, který sdružuje veškeré ostatní projekty a soubory do jednoho souboru.

Pro správu externích knihoven byl využít správce balíků nuget.org, jenž je celosvětově největším podobným správcem a v .NET komunitě již nepochybně standardem.

Veškerý kód napsaný v rámci projektu byl spravován pomocí nástroje Git. Veřejný repozitář projektu lze procházet na adrese https://pajda.fit.vutbr.cz/testos/taipan. I přes to, že byla celá práce vyvíjena pouze jedním člověkem, byl tento nástroj nezastupitelný ve verzování projektu, udržování historie a sledování vývoje, kterým řešení procházelo.

V práci bylo využito prostředí MS Azure, a především jeho možnost vytváření pipelines, které po změně zdrojových souborů provedou automatické akce nad změněným kódem. Pipeline využita v projektu lze vidět na adrese https://dev.azure.com/DavidLingos/Taipan/_build včetně všech jejích historických běhů.
V rámci pipeline byly automaticky prováděny následující akce:

1. build – překlad celého řešení pomocí příkazu `dotnet build`
2. test – spuštění všech testů obsažených v repozitáři pomocí příkazu `dotnet test`
3. pack – zabalení řešení do `.nupkg` souboru pomocí příkazu `dotnet pack`
4. push – publikace knihovny na server nuget.org pomocí příkazu `nuget push`

[3] https://doc.postsharp.net/configuration-schema#multicast

Kapitola 5

Testování a validace výsledků

Ověření správnosti funkcionality probíhalo pomocí jednotkového testování v průběhu implementace. Dále byla navržena sada integračních testů, kterými byla ověřena funkcionalita instrumentace a dynamických analyzátorů. V poslední řadě byly vytvořeny ukázkové programy obsahující chyby v paralelismu, na kterých bylo prezentováno celkové využití nástroje včetně konfigurace a výstupu.

5.1 Jednotkové testy

K jednotkovému testování byl využit nástroj `xUnit.net`[1], především jeho třída `Assert`, která obsahuje pomocné metody pro ověřování podmínek v rámci jednotlivých testů a prostředí, které poskytuje pro běh testů a přehledné zobrazení výsledků v prostředí Visual Studia. Všechny jednotkové testy lze nalézt v projektu `Taipan.UnitTests`.
Jednotkové testy byly využity k testování základních prvků nástroje Taipan, především:

- Testování správného načtení konfiguračního souboru.
- Testování implementace rozhraní `IAnalysersProvider` poskytující metody pro registraci analyzátorů.
- Testování detekce cyklů v grafu zámků.

5.1.1 Testování správného načtení konfiguračního souboru

Korektní načtení konfigurace z konfiguračního souboru je jedním z prvních úkonů po startu programu. Nekorektní konfigurace či její nesprávné načtení by vedlo k běhu programu, jehož chování by nemuselo být žádoucí. Otestování této části systému se tak zdá být skoro až nutností.

Testovací sada (třída `ConfigurationTests`) obsahuje 5 různých konfigurací, které může soubor `postsharp.config` obsahovat. Dvě z nich nelze považovat za platné (z důvodu chybějících či neplatných hodnot). Ostatní by měly být načteny do programu korektně.

V jednotlivých testech je ověřeno, zda dojde ke správnému zpracování vstupu (načtení hodnot do instance třídy `TaipanConfiguration`), či ke správnému ukončení načítání konfigurace vyvoláním výjimky (v případě nevalidní konfigurace).

[1] https://xunit.net/

5.1.2 Testování implementace rozhraní `IAnalysersProvider`

Třída implementující rozhraní `IAnalysersProvider` slouží pro práci s instancemi analyzátorů, které jsou použity za běhu programu. Třída `DefaultAnalysersProvider` je jednou z nejdůležitějších tříd celého nástroje. V rámci jednotkových testů (třída `AnalysersProviderTests`) byla ověřena funkčnost metod `SetAnalysers` (registrace analyzátorů pro běh programu), `AddAnalyser` (registrace nového analyzátoru do kolekce analyzátorů) a `GetAnalysers` pro získání všech zaregistrovaných analyzátorů.

5.1.3 Testování detekce cyklů v grafu zámků

Testovací sada pro testování detekce cyklů (třída `CylesDetectionTests`) obsahuje 4 scénáře skládající se z různého počtu uzlů a hran. Testování spočívá v sestrojení grafu pomocí třídy `AdjacencyGraph` z knihovny `QuickGraph`, zavolání metody `FindCycles` z instance třídy `CyclesDetector` implementující rozhraní `ICyclesDetector` a evaluace výsledků tohoto volání.

5.2 Integrační testy

Integrační testy byly využity pro testování korektní instrumentace programu a zasílání událostí z instrumentovaného kódu a pro testování funkčnosti jednotlivých analyzátorů. Obě části vyžadovaly rozdílné přístupy k testování, které jsou v následujících částech popsány. Všechny integrační testy lze nalézt v projektu `Taipan.IntegrationTests`.

5.2.1 Testování instrumentace programu

Pro integrační testy instrumentace bylo třeba vyvinout testovací instrumentační aspekty pro nástroj Postsharp (viz 2.5.2), které byly přiřazeny jednotlivým třídám. Aspekty využívají metody třídy `AspectInstancesHelper` (viz 4.2) pro instrumentaci jednotlivých testů. Testovací aspekty jsou implementovány ve třídách:

- `MemoryAccessInstrumentationTestsAspectProvider` – aspekt přiřazen třídě obsahující testy instrumentace přístupu do paměti.
- `MemoryAccessWithNoiseInstrumentationTestsAspectProvider` – aspekt přiřazen třídě obsahující testy instrumentace přístupu do paměti s vkládáním šumu.
- `SynchronizationInstrumentationTestsAspectProvider` – aspekt přiřazen třídě obsahující testy instrumentace synchronizačních primitiv.
- `ThreadInstrumentationTestsAspectProvider` – aspekt přiřazen třídě obsahující testy instrumentace vzniku a zániku vláken.

Pro testování korektní instrumentace programu byl implementován testovací dynamický analyzátor. Třída implementující tento analyzátor (`TestAnalyser`) dědí jako ostatní dynamické analyzátory z třídy `AnalyserBase`. Práce tohoto analyzátoru tkví v tom, že po přijmutí jakékoliv události z nainstrumentovaného kódu analyzátor inkrementuje čítač pro zprávy daného typu. Všechny testy instrumentace pomocí tohoto analyzátoru mohly ověřit počet událostí konkrétního typu vyvolaných v průběhu testu a porovnat jej s očekávanými hodnotami.

Jednotlivé testy instrumentace probíhaly následovně:

1. Vynulování čítačů testovacího analyzátoru (fáze setup).
2. Provedení instrukcí testu (fáze exercise).
3. Kontrola počtu událostí vyvolaných provedenými instrukcemi (fáze verify).

Na následujícím příkladu lze vidět ukázka jednoho testu instrumentace, konkrétně instrumentace čtení z paměti. Při čtení proměnné by mělo dojít k vyvolání události `BeforeMemoryReadEvent` a poté `AfterMemoryReadEvent`. Tato skutečnost je tímto testem ověřena.

```
public void ReadFieldTest()
{
    // Setup - reset counters
    InstrumentationTestHelpers.ResetTestAnalyser();

    // Exercise - read variable _field
    if(_field == 5)
    {
        return;
    }


    // Verify - verify events count by type
    var testAnalyser = InstrumentationTestHelpers
        .GetInstrumentationTestAnalyser();
    Assert.True(testAnalyser
        .GetEventsCount(typeof(BeforeMemoryReadEvent)) == 1);
    Assert.True(testAnalyser
        .GetEventsCount(typeof(BeforeMemoryWriteEvent)) == 0);
    Assert.True(testAnalyser
        .GetEventsCount(typeof(AfterMemoryReadEvent)) == 1);
    Assert.True(testAnalyser
        .GetEventsCount(typeof(AfterMemoryWriteEvent)) == 0);
}
```

Kód 5.1: Ukázka testu instrumentace

5.2.2 Testování funkčnosti analyzátorů

Testování funkčnosti analyzátorů bylo provedeno tak, že pro každý analyzátor byla definována testovací sada, ve které každý z testů byl tvořen sekvencí událostí, která by mohla nastat v reálném programu. Tyto události byly zaslány testovanému analyzátoru, byl kontrolován výsledek analýzy (zdali byla či nebyla chyba detekována) a tento výsledek byl porovnán s očekávaným chováním.

Konfigurace událostí, které by měly být v testu vykonány, byly reprezentovány třídami dědícími z abstraktní třídy `TestEventConfigurationBase`. Konfigurační třídy byly následující:

- `LockEventTestConfiguration` – konfigurace události vyvolané před/po zamknutí/uvolnění zámku.

- `MemoryAccessEventTestConfiguration` – konfigurace událostí vyvolané před/po přístupu do paměti.
- `RunAnalysisEventTestConfiguration` – konfigurace události pro běh analýzy.
- `ThreadEventTestConfiguration` – konfigurace události vyvolané po vytvoření či zániku vlákna.

Konfigurace jednoho testu je ztvárněna třídou `AnalyserTestConfiguration`. Tato třída obsahuje následující atributy:

- `Events` – kolekce instancí třídy `TestEventConfigurationBase` popsané výše. Reprezentuje posloupnost událostí, které mají být v rámci testu zaslány analyzátoru.
- `DataRaceShouldBeFound` – pravdivostní hodnota značící zdali by analyzátor měl nalézt časově závislou chybu nad daty.
- `DeadlockShouldBeFound` – pravdivostní hodnota značící, zdali by analyzátor měl nalézt chybu uváznutí.

Provedení testu probíhá voláním statické metody `RunTest` třídy `AnalysersTestRunner`. Tato metoda přijímá jako parametry analyzátor, který je testován a instanci třídy `AnalyserTestConfiguration`. V metodě jsou analyzátoru zaslány všechny události z konfigurace a odchytávány výjimky, které analyzátor vyvolá. Na konci testu jsou porovnány hodnoty v atributech `DataRaceShouldBeFound` a `DeadlockShouldBeFound` s výsledkem analýzy a navrácena pravdivostní hodnota, která značí (ne)úspěšný výsledek testu.

V následujícím příkladu lze vidět konfiguraci jednoho testu analyzatorů pro detekci časově závislých chyb nad daty. Konfigurace obsahuje 4 události.

1. Vlákno s číslem 1 začíná číst z atributu `location`.
2. Vlákno s číslem 2 začíná zapisovat do atributu `location`.
3. Vlákno s číslem 1 ukončuje čtení z atributu `location`.
4. Vlákno s číslem 2 ukončuje zápis do atributu `location`.

Tato posloupnost události by měla vést k vyvolání výjimky značící detekci časově závislé chyby nad daty, ale ne k vyvolání výjimky značící detekci chyby uváznutí (pravdivostní hodnoty `true, false` na posledním řádku kódu 5.2).

```
// Incorrectly synchronized WAR - Data race
new AnalyserTestConfiguration(new MemoryAccessEventTestConfiguration[]
{
        new MemoryAccessEventTestConfiguration(
            1, location, MemoryAccessType.Read, true),
        new MemoryAccessEventTestConfiguration(
            2, location, MemoryAccessType.Write, true),
        new MemoryAccessEventTestConfiguration(
            1, location, MemoryAccessType.Read, false),
        new MemoryAccessEventTestConfiguration(
            2, location, MemoryAccessType.Write, false)
```

```
}, true, false)
```

Kód 5.2: Ukázka konfigurace testu dynamického analyzátoru pro detekujícího časově závislé chyby nad daty

5.3 Demonstrační programy

Pro kompletní demonstraci využití nástroje Taipan byly vytvořeny dva ukázkové programy, na kterých je ukázáno to, jak nakonfigurovat instrumentaci a analyzátory použité pro běh programu. Jeden program obsahuje časově závislou chybu nad daty a druhý chybu uváznutí. Oba tyto programy lze spustit a sledovat výstup, který analýza poskytne poté, co je chyba detekována. Oba demonstrační programy lze nalézt ve složce `samples`.

5.3.1 Program demonstrující analýzu časově závislých chyb nad daty

Program `DataRaceDetectionSample` implementuje teoretický příklad bankovního systému popsaný v kapitole 2.3.4. V souboru `postsharp.config` lze vidět konfigurace nutná k tomu, aby byl program korektně nainstrumentován a chyba byla detekována. Jako analyzátor byl vybrán `AtomRaceDataRaceAnalyser`, instrumentovány byly veškeré atributy všech tříd v jmenném prostoru `DataRaceDetectionSample`. Po spuštění programu dojde k vyvolání výjimky `DataRaceException` mimojiné s následujícími informacemi.

```
Taipan.Analysers.DataRaces.Exceptions.DataRaceException
Message=Data race detected. The following synchronization elements formed
concurrent accesses:
------- Stack trace for thread 11 -------
...
at DataRaceDetectionSample.Program.set__amount(Int32 value) in :line 0
at DataRaceDetectionSample.Program.Thread1Start() in ...\Program.cs:line 14
...

------- Stack trace for thread 12 -------
...
at DataRaceDetectionSample.Program.set__amount(Int32 value) in :line 0
at DataRaceDetectionSample.Program.Thread2Start() in ...\Program.cs:line 25
...
```

Kód 5.3: Výstup analýzy časově závislých chyb nad daty

Z výpisu lze tedy vyčíst, že k chybě došlo, když vlákno s číslem 11 zapisovalo do proměnné amount (v kódu na řádku 14) a zároveň vlákno s číslem 12 zapisovalo do stejné proměnné (v kódu na řádku 25).

5.3.2 Program demonstrující analýzu chyb uváznutí

V programu `DeadlockDetectionSample` je implementován teoretický příklad převodu mezi dvěma bankovními účty. Po spuštění programu dojde k vytvoření dvou vláken, z nichž první simuluje převod z bankovního účtu A na účet B a druhé totožný převod jen v opačném pořadí. Pro změnu zůstatku na účtu (proměnné `account1Balance`/`account2Balance`) je potřeba získat zámek náležící danému účtu

(proměnné `account1BalanceLock`/`account2BalanceLock`). V programu dojde k chybě uváznutí stejné, jaká je ukázána v příkladu v kapitole 2.3.1. V souboru `postsharp.config` lze vidět konfigurace nutná k tomu, aby byl program korektně nainstrumentován a chyba byla detekována. Jako analyzátor je nutné zvolit `GoodlockDeadlockAnalyser`, instrumentovány byly veškeré metody sloužící pro zamknutí/odemknutí zámku a vzniku/zániku vlákna. Po spuštění programu dojde k vyvolání výjimky `DeadlockException` mimojiné s následujícími informacemi.

```
Taipan.Analysers.Deadlocks.Exceptions.DeadlockException
Message=Deadlock detected. The following synchronization elements form
a cycle:
------- Stack trace for thread 3 -------
...
at Taipan.Aspects.MonitorEnterAspect.OnEntry(MethodExecutionArgs args)
at PostSharp.ImplementationDetails._9d24223a.<>z__Aspects.
<System.Threading.Monitor.Enter>b__(Object obj, Boolean& lockTaken) in
:line 0
at DeadlockDetectionSample.Program.TransferFromAccount1ToAccount2
(Int32 amount) in ...\Program.cs:line 20
at DeadlockDetectionSample.Program.<>c.<Main>b__6_0() in ...\Program.cs
:line 41
...

------- Stack trace for thread 4 -------
...
at Taipan.Aspects.MonitorEnterAspect.OnEntry(MethodExecutionArgs args)
at PostSharp.ImplementationDetails._9d24223a.<>z__Aspects.
<System.Threading.Monitor.Enter>b__(Object obj, Boolean& lockTaken) in
:line 0
at DeadlockDetectionSample.Program.TransferFromAccount2ToAccount1
(Int32 amount) in ...\Program.cs:line 33
at DeadlockDetectionSample.Program.<>c.<Main>b__6_1() in ...\Program.cs
:line 42
...
```

Kód 5.4: Výstup analýzy chyb uváznutí

Z výpisu lze tedy vyčíst, že k chybě došlo, když vlákno s číslem 3 žádalo o zámek na řádku 20 a zároveň vlákno s číslem 4 žádalo o ten stejný zámek na řádku 33.

Kapitola 6

Nasazení nástroje v systému MES PHARIS

V průběhu návrhu i implementace nástroje Taipan bylo výsledné řešení konzultováno s firmou UNIS a.s. za účelem toho, aby byl vyvinutý nástroj využitelný pro otestování systému MES PHARIS této firmy. V následující závěrečné kapitole je ve stručnosti představen systém MES PHARIS a je popsán způsob testování tohoto systému za použití kombinace statické a dynamické analýzy. V závěru kapitoly jsou ukázány dva testovací scénáře pro systém MES PHARIS, kde v jednom se potvrdila chyba ze statické analýzy, v druhém z nich chyba odhalena nebyla.

MES PHARIS (MES – Manufacturing Execution System) je plnohodnotný modulární výrobní informační systém firmy UNIS a.s. pro kapacitní plánování a rozvrhování výroby, řízení výroby, údržby, výrobní logistiky a sběr dat ze strojů a technologií ve výrobě. MES PHARIS pokrývá potřeby výroby od okamžiku vystavení výrobního příkazu až po zaskladnění výsledného produktu. Zajišťuje kontinuální sledování, řízení a vyhodnocování výroby v reálném čase, sběr dat z technologických zařízení (vstřikovací lisy, CNC stroje, montážní linky, přídavná zařízení – sušící zařízení, temperační přístroje, kontrolní stanice, ...) a dlouhodobé ukládání všech průvodních informací. Tato data jsou následně využívána pro analýzu výroby za účelem kontinuálního zvyšování efektivity výroby a rentability výrobních podniků.

MES PHARIS je vysoce paralelní systém. Ve všech jeho částech můžou najednou běžet stovky vláken současně. Z toho pramení potřeba mít systém pokryt testy, které případné problémy spojené s paralelismem a synchronizací mezi jednotlivými vlákny pomůžou odhalit dříve než v produkčním prostředí (v konkrétním výrobním závodu).

6.1 Způsob testování

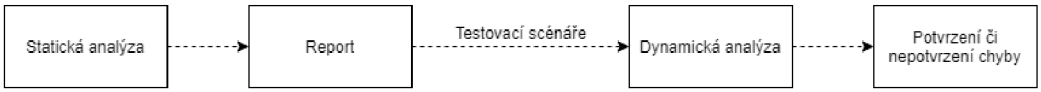
Z důvodů vysoké komplexity systému MES PHARIS by bylo spuštění dynamické analýzy nástrojem Taipan vysoce náročnou operací. Instrumentace veškerých přístupů do paměti a práce se zámky by několikanásobně zvýšila časové i paměťové nároky aplikace. Nasazení nástroje Taipan v produkčním prostředí tedy bylo vyloučeno, jelikož by způsobilo více než značné zpomalení systému, které by pro koncového zákazníka bylo nepřípustné.

V disertační práci „Combining static and dynamic analysis to find multi-threading faults beyond data races“ [2] byly sepsány možnosti testování paralelních programů za využití

kombinace statické a dynamické analýzy a představen nástroj JNuke, který právě na základě kombinace obou přístupů provádí analýzu programů napsaných v jazyce Java [3].

Kombinace statické a dynamické analýzy může vést k potlačení negativních vlastností jednotlivých přístupů, a naopak posílení vlastností dobrých. Statická analýza, jakožto většinou méně přesný ale rychlý nástroj, může pomoci s odhalením potenciálně chybných míst v programu, na která se poté může testování pomocí dynamické analýzy zaměřit. Není tak třeba instrumentovat celý program, ale pouze ty části, které byly statickou analýzou vyhodnoceny jako potenciálně nevalidní. Tím lze snížit výpočetní nároky dynamické analýzy, zkombinovat výstupy obou analýz a získat kvalitnější výsledky.

Na obrázku 6.1 je ukázán navržený způsob testování systému MES PHARIS, kdy nejprve bylo využito statických analyzátorů (viz 6.1.1) pro detekci potenciálně chybných míst. Výstup statické analýzy je poté zpracován a jsou podle něj navrženy testovací scénáře, nad kterými je následně spuštěna dynamická analýza, která může chybu potvrdit.



Obrázek 6.1: Diagram způsobu testování systému MES PHARIS

6.1.1 Použité statické analyzátory

Pro statickou analýzu byly využity dva různé analyzátory z dílny švýcarské výzkumné skupiny OST Concurrency Lab[1]. Oba analyzátory jsou jinak navrženy a generují rozdílné výstupy. Kombinací výstupů z obou analyzátorů vznikl kompletní report, ve kterém byla následně vytipována místa (třídy, moduly, ...), která byla podrobena dynamické analýze.

Parallel Checker

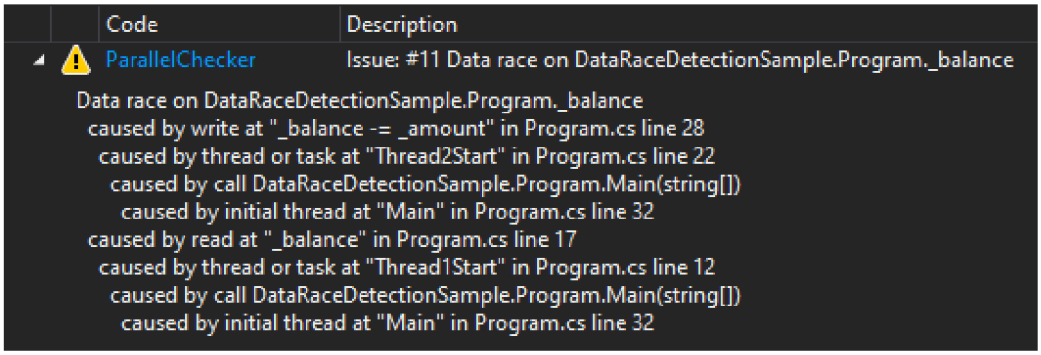
Parallel Checker je nástroj, který se zaměřuje na detekci běžných chyb v paralelismu jako jsou chyby uváznutí a časově závislé chyby nad daty [6]. Nástroj se zaměřuje na analýzu, která by měla mít následující nezbytné vlastnosti:

- Statická – nástroj analyzuje zdrojový kód bez nutnosti spouštět program.
- Rychlá – nástroj se snaží co nejrychleji analyzovat nově psaný kód a programátora včas varovat o potenciálních problémech.
- Precizní – analýza je prováděna způsobem, aby byl co nejvíce redukován počet falešných hlášení.

Nástroj je distribuován jako rozšíření do vývojového prostředí Visual Studio, kde analyzuje aktuální projekt a poskytuje programátorovi okamžitou zpětnou vazbu v podobě varování. Na obrázku 6.2 lze vidět ukázku výstupu, který nástroj vygeneruje po detekci potenciální chyby.

Mezi hlavní výhody nástroje patří především rychlost analýzy a to, že dokáže detekovat problémy přímo při psaní zdrojového kódu. Mezi nevýhody patří to, že analýza obsahuje více falešných negativ, kdy analyzátor nedokáže odhalit problémy komplexnějšího rázu.

[1] https://concurrency.ch/



Obrázek 6.2: Ukázka výstupů statického analyzátoru Parallel Checker

Parallel Helper

Parallel Helper je podobně jako nástroj Parallel Checker poskytován jako rozšíření do vývojového prostředí Visual Studio [8].

Narozdíl od nástroje Parallel Checker se Parallel Helper zaměřuje i na místa v kódu, ve kterých svou analýzou chybu nenalezne, ale jsou svým zápisem podezřelá z toho, že by v nich chyba mohla nastat nebo na programátorské konstrukce, které nedodržují doporučované praktiky pro psaní paralelního kódu v jazyce C#. Parallel Helper výstupy z analýzy řadí do tří kategorií:

- Best Practices (Nejlepší praktiky) – doporučované postupy pro psaní robustnějšího kódu.
- Smells (Podezření) – místa v kódu, která porušují doporučené praktiky a mohou vést k neočekávanému chování při užívání rozhraní třetími stranami.
- Bugs (Chyby) – nejzávažnější z nahlášených problémů. Tento typ označuje místa v kódu, kde se s vysokou pravděpodobností nachází problémy s paralelismem.

Seznam všech analýz, které nástroj provádí, lze vidět na adrese `https://github.com/Concurrency-Lab/ParallelHelper/tree/master/doc/analyzers`. Analýzy s názvem začínajícím na `PH_B` generují hlášení typu Bugs, `PH_P` hlášení typu Best Practices a `PH_S` hlášení typu Smells.

Hlavní výhodou tohoto nástroje je to, že generuje více (obecnějších) varování, a tudíž umí poukázat na více potenciálních problémů. Část výstupu je sice tvořena falešnými hlášeními o chybách, které ve skutečnosti chybami nejsou, ale mohou poukázat na slabá místa v kódu, na které by měla být zaměřena vývojářova pozornost či následná dynamická analýza.

6.2 Nalezené chyby v systému MES PHARIS

Systém MES PHARIS se skládá z více než 200 C# projektů (knihoven, klientských aplikací, testovacích projektů, ...), pro které byla spuštěna statická analýza. Report z výstupů statické analýzy vedl k identifikaci míst, na kterých by bylo vhodné otestovat funkčnost nástroje Taipan a pro tato místa byly následně vytvářeny automatické testy za použití příslušného dynamického analyzátoru.

Pro účely této práce byly vybrány dva testovací scénáře pro systém MES PHARIS. S testováním paralelismu v dalších částech systému bude v budoucnu i za pomocí nástroje Taipan pokračováno.

Výstupem analýzy pomocí Parallel Checkeru bylo 219 varování v 15 projektech. 56 z těchto problémů byly problémy typu „Thread-unsafe calls“, poukazující na nebezpečné volání funkcí nad jedním objektem (většinou kolekcí). Ostatní problémy značily potenciální časově závislé chyby nad daty. Nutno zde podotknout, že některé z problémů byly ve výsledcích analýzy duplikovány, protože byly analýzou detekovány vícero cestami.

Výstupem analýzy pomocí Parallel Helperu bylo 1270 varování v 93 projektech. 504 z nich bylo z kategorie Bugs, 681 z kategorie Best Practices a zbytek z kategorie Smells. Většina z varování typu Bugs byla spojená s potenciální časově závislou chybou nad daty, kdy k jednomu atributu bylo v programu přistupováno jak z místa chráněného zámkem, tak z místa, kde zámek použit nebyl. Další častou chybou byla manipulace s kolekcemi, kdy alespoň jeden z přístupů byl chráněn zámkem a alespoň jeden chráněn nebyl.
Mezi méně závažnými chybami se objevovaly chyby typu:

- Asynchronní metoda vracející typ `void`.
- Chybějící klíčové slovo `await`.
- Vytváření nových vláken v konstruktorech tříd.
- ...

Jako ukázky testů, které ověřovaly potenciální chyby detekované statickou analýzou, byly vybrány dvě varování statického analyzátoru Parallel Checker, z nichž jedna chyba byla dynamickou analýzou nástrojem Taipan potvrzena, druhá potvrzena nebyla.

První případ byl detekován statickým analyzátorem Parallel Checker jako potenciální data race nad atributy `lastComputedRemoval` a `recentOrderOdUncomputed` třídy `ManAndMachineTimeService`. Konkrétně výpis analyzátoru vypadal následovně:

```
Issue: #8 Data race on Phoenix.PharMach.Prod.Production.ManAndMachineTime.
ManAndMachineTimeService.recentOrderOdUncomputed
Issue: #5 Data race on Phoenix.PharMach.Prod.Production.ManAndMachineTime.
ManAndMachineTimeService.lastComputedRemoval
```

Kód 6.1: Reálný výstup nástroje Parallel Checker

K těmto atributům se přistupuje ve veřejné metodě `GetAllUncomputedClosedRemovals` a z ní volané privátní metodě `GetNextOrderOfUncomputed`. Obě metody obsahují zápis i čtení z obou atributů. Metoda je v programu volána ze dvou různých míst a tato volání byla v testu použita.

Ve zjednodušené verzi by testovací scénář spočíval ve spuštění dvou vláken a v nich metody `Start`. Metoda Start ve svém těle volá 100× metodu `GetAllUncomputedClosedRemovals`. V tomto běhu programu byla prováděna dynamická analýza algoritmy AtomRace i FastTrack.

```
void Start() {
    for(i=0;i<100;i++) {
        GetAllUncomputedClosedRemovals();
    }
```

```
}

var thread1 = new Thread(Start).Start();
var thread2 = new Thread(Start).Start();

thread1.Join();
thread2.Join();
```

Kód 6.2: Pseudokód testu pro detekci časově závislé chyby nad daty ve třídě `ManAndMAchineTimeService`

V uvedeném případě Taipan chybu potvrdil u obou atributů. V kódech 6.3 a 6.4 lze vidět ukázky konkurentních přístupů k atributu `recentOrderOdUncomputed` způsobující časově závislou chybu nad tímto atributem.

```
public IEnumerable<CRemovalOfWork> GetAllUncomputedClosedRemovals()
{
        ...
        if (recentOrderOdUncomputed == EOrderOfUncomputed.TheOldest)
        {
            ...
        }
    ...
}
```

Kód 6.3: Přístup k atributu recentOrderOdUncomputed v metodě GetAllUncomputedClosedRemovals

```
private static string GetNextOrderOfUncomputed()
{
        ...
        recentOrderOdUncomputed++;
        ...
}
```

Kód 6.4: Přístup k atributu recentOrderOdUncomputed v metodě GetNextOrderOfUncomputed

V druhém případě statický analyzátor Parallel Checker ve svém výstupu obsahoval mimojiné řádek:
Issue: 18 Data race on Phoenix.PharMach.Prod.InfoTable.CInformationTable.Inner.instance značící potenciální časově závislou chybu nad parametrem `instance` třídy `CInformationTable.Inner`.

Pro tento případ byl navržen test, který spustí 100 vláken a v každém z nich přistoupí k objektu `instance` a zavolá na něm metodu `CheckStates` provádějící dotaz do databáze.

```
void Start()
{
    var instance = CInformationTable.Instance;
    instance.CheckStates();
}
// Create 100 threads calling method Start
```

```
var threads = new List<Thread>();
for(var i = 0; i < 100; i++)
{
    threads.Add(new Thread(Start));
}
...
// Wait until all threads finish their work
```

Kód 6.5: Pseudokód testu atributu `instance` třídy `CInformationTable.Inner`

V tomto konkrétním případě nebyla časově závislá chyba nad daty nástrojem Taipan detekována především z toho důvodu, že nástroj Taipan při instrumentaci atributů tříd neinstrumentuje atributy, které jsou označeny klíčovým slovem `readonly`. Nástroj Parallel Checker naopak toto klíčové slovo při své analýze neuvažuje.

Kapitola 7

Závěr

Cílem této práce bylo navázat na výzkumné projekty skupiny VeriFIT, navrhnout a implementovat dynamický analyzátor pro programy napsané v jazyce C# se zaměřením na paralelní programy a chyby v paralelismu (především chyby uváznutí a časově závislé chyby nad daty). Jedním ze základních požadavků byla kompatibilita výsledného nástroje s platformou .NET Framework, jelikož funkcionalita byla ověřená mimo jiné na systému MES PHARIS od firmy UNIS a.s., který je provozován nad touto platformou.

Výstupem celé práce je nástroj Taipan, který využívá pro instrumentaci proprietární nástroj Postsharp. Pro detekci chyb uváznutí byl implementován algoritmus Goodlock a pro analýzu časově závislých chyb nad daty algoritmy FastTrack a AtomRace. Funkcionalita nástroje Taipan byla ověřena jednotkovými a integračními automatickými testy, které pokrývají instrumentační část i funkčnost jednotlivých analyzátorů.

Nástroj Taipan byl využit při testování systému MES PHARIS, které probíhalo kombinací statické a dynamické analýzy. Pomocí statické analýzy byla detekována místa v programu s potenciálními problémy. Výstup statické analýzy poté posloužil jako vstupní informace pro dynamickou analýzu. Vytipovaná místa byla pokryta automatickými testy, byla provedena instrumentace a pomocí nástroje Taipan byly potvrzovány či naopak vyvraceny hypotézy statické analýzy. Dynamickou analýzou bylo potvrzeno několik reálných chyb v systému MES PHARIS. V testování dalších částí systému MES PHARIS bude pokračováno.

Budoucí vývoj nástroje Taipan by mohl vést směrem optimalizace dynamické analýzy tak, aby bylo analýzu možné spustit i nad většími systémy a ne jen na částech komplexních systémů. Nástroje Taipan lze jednoduše rozšířit o další dynamické analyzátory, a bylo by jej možné rozšířit např. o analyzátor paralelních volání funkcí nad LINQ abstrakcemi, které jsou v programovacím jazyku C# hojně využívány či jiné dynamické analyzátory.

-

Literatura

[1] Andrews, G. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000. ISBN 9780201357523. Dostupné z: https://books.google.cz/books?id=npRQAAAAMAAJ.

[2] Artho, C. Combining static and dynamic analysis to find multi-threading faults beyond data races. In:. Hartung-Gorre, April 2005. ISBN 3896499971.

[3] Artho, C. a Biere, A. Combined Static and Dynamic Analysis. *Electronic Notes in Theoretical Computer Science*. 2005, sv. 131, s. 3–14. DOI: https://doi.org/10.1016/j.entcs.2005.01.018. ISSN 1571-0661. Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005). Dostupné z: https://www.sciencedirect.com/science/article/pii/S1571066105002537.

[4] Ball, T. The Concept of Dynamic Analysis. In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Springer-Verlag, 1999, s. 216–234. ESEC/FSE-7. ISBN 3540665382.

[5] Bensalem, S. a Havelund, K. Dynamic Deadlock Analysis of Multi-Threaded Programs. In: *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*. Berlin, Heidelberg: Springer-Verlag, 2005, s. 208–223. HVC'05. DOI: 10.1007/11678779_15. ISBN 3540326049. Dostupné z: https://doi.org/10.1007/11678779_15.

[6] Bläser, L. Practical Detection of Concurrency Issues at Coding Time. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2018, s. 221–231. ISSTA 2018. DOI: 10.1145/3213846.3213853. ISBN 9781450356992. Dostupné z: https://doi.org/10.1145/3213846.3213853.

[7] Čižmárik, A. a Parízek, P. SharpDetect: Dynamic Analysis Framework for C#/.NET Programs. In: *Runtime Verification*. Cham: Springer International Publishing, 2020, s. 298–309. ISBN 978-3-030-60508-7.

[8] ConcurrencyLab. *Parallel Helper*. GitHub, 2021. Dostupné z: https://github.com/Concurrency-Lab/ParallelHelper.

[9] Downey, A. B. *The Little Book of Semaphores*. Green Tea Press, March 2016. Dostupné z: https://open.umn.edu/opentextbooks/textbooks/83.

[10] EcmaInternational. *Common Language Infrastructure (CLI) the 6-th edition (June 2012)*. Ecma international, 2021. Dostupné z: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[11] Elmas, T., Qadeer, S. a Tasiran, S. *Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets*. MSR-TR-2006-163. November 2006. 18 s. Dostupné z: https://www.microsoft.com/en-us/research/publication/goldilocks-efficiently-computing-the-happens-before-relation-using-locksets/.

[12] Fiedor, J., Hrubá, V., Křena, B., Letko, Z., Ur, S. et al. Advances in Noise-Based Testing of Concurrent Software. *Softw. Test. Verif. Reliab.* GBR: John Wiley and Sons Ltd. květen 2015, sv. 25, č. 3, s. 272–309. DOI: 10.1002/stvr.1546. ISSN 0960-0833. Dostupné z: https://doi.org/10.1002/stvr.1546.

[13] Flanagan, C. a Freund, S. N. FastTrack: Efficient and Precise Dynamic Race Detection. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2009, s. 121–133. PLDI '09. DOI: 10.1145/1542476.1542490. ISBN 9781605583921. Dostupné z: https://doi.org/10.1145/1542476.1542490.

[14] Kempf, T., Karuri, K. a Gao, L. Software Instrumentation. In: *Wiley Encyclopedia of Computer Science and Engineering*. American Cancer Society, 2008, s. 1–11. DOI: https://doi.org/10.1002/9780470050118.ecse386. ISBN 9780470050118. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386.

[15] Kshemkalyani, A. D. a Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[16] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. červenec 1978, sv. 21, č. 7, s. 558–565. DOI: 10.1145/359545.359563. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/359545.359563.

[17] Letko, Z., Vojnar, T. a Křena, B. AtomRace: Data Race and Atomicity Violation Detector and Healer. In: *PADTAD '08*. Association for Computing Machinery, 2008, s. 1–10. Proceedings of the 6th workshop on Parallel and distributed systems. ISBN 978-1-60558-052-4. Dostupné z: https://www.fit.vut.cz/research/publication/8685.

[18] Lucia, B., Devietti, J., Strauss, K. a Ceze, L. Atom-Aid: Detecting and Surviving Atomicity Violations. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. USA: IEEE Computer Society, 2008, s. 277–288. ISCA '08. DOI: 10.1109/ISCA.2008.4. ISBN 9780769531748. Dostupné z: https://doi.org/10.1109/ISCA.2008.4.

[19] MicrosoftCorporation. *The Book of the Runtime*. GitHub, 2021. Dostupné z: https://github.com/dotnet/coreclr/tree/master/Documentation/botr.

[20] MicrosoftCorporation. *Microsoft Documentation*. Microsoft Corporation, 2021. Dostupné z: https://docs.microsoft.com/cs-cz/?view=netframework-4.6.1.

[21] Musuvathi, M., Qadeer, S. a Ball, T. *CHESS: A systematic testing tool for concurrent software*. MSR-TR-2007-149. November 2007. 16 s. Dostupné z:

https://www.microsoft.com/en-us/research/publication/chess-a-systematic-testing-tool-for-concurrent-software/.

[22] Netzer, R. a Miller, B. What are Race Conditions? - Some Issues and Formalizations. *ACM letters on programming languages and systems*. Září 1992, sv. 1. DOI: 10.1145/130616.130623.

[23] Nielson, F., Nielson, H. R. a Hankin, C. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. ISBN 3642084745.

[24] PostSharpTechnologies. Configuring Projects Using postsharp.config. In:. PostSharp Technologies, 2021. Dostupné z: https://doc.postsharp.net/configuration-system.

[25] Pozniansky, E. a Schuster, A. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery. červen 2003, sv. 38, č. 10, s. 179–190. DOI: 10.1145/966049.781529. ISSN 0362-1340. Dostupné z: https://doi.org/10.1145/966049.781529.

[26] Rahul V. Patil, B. G. Tools And Techniques To Identify Concurrency Issues. In:. June 2008. Dostupné z: https://docs.microsoft.com/en-us/archive/msdn-magazine/2008/june/tools-and-techniques-to-identify-concurrency-issues.

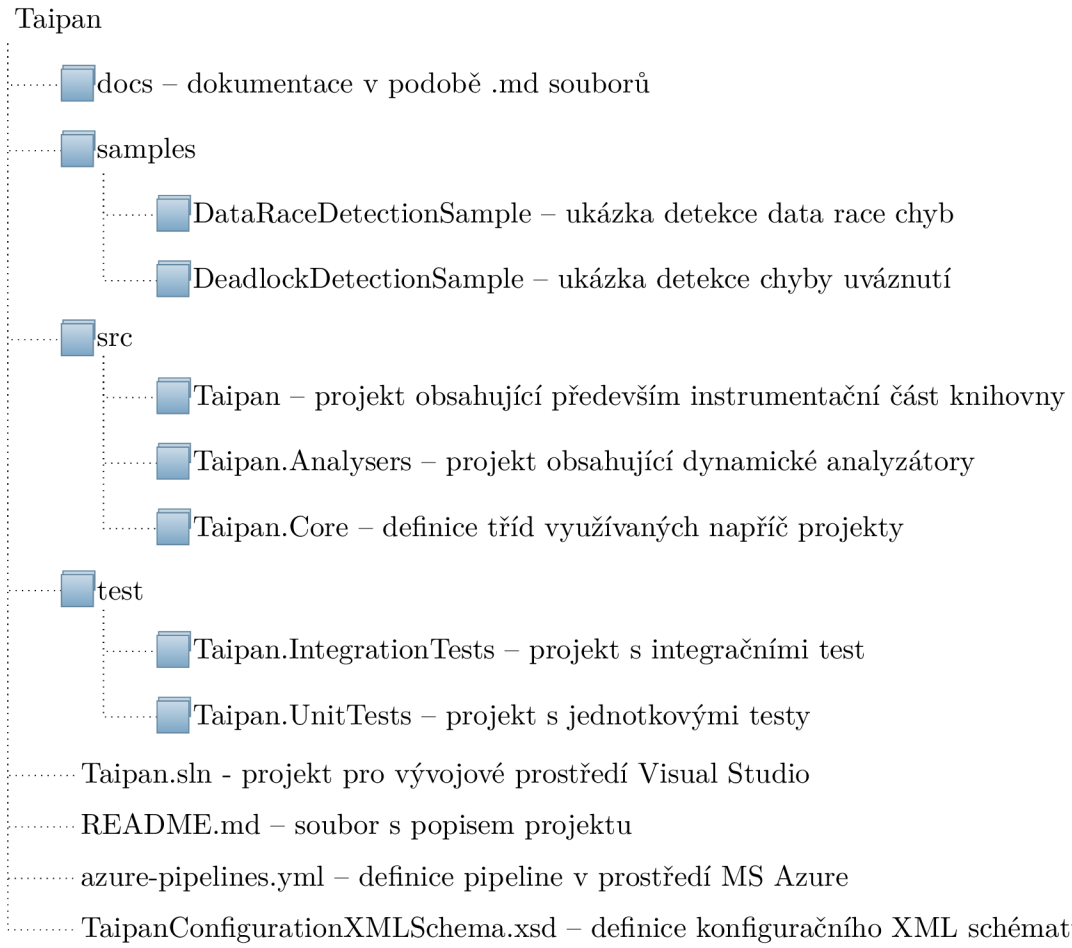
[27] Silberschatz, A., Galvin, P. B. a Gagne, G. *Operating System Concepts*. 9th. Wiley Publishing, 2012. ISBN 1118063333.

[28] Stallings, W. *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011. 1-788 s. ISBN 978-0-273-75150-2.

[29] Yu, Y., Rodeheffer, T. a Chen, W. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.* New York, NY, USA: Association for Computing Machinery. říjen 2005, sv. 39, č. 5, s. 221–234. DOI: 10.1145/1095809.1095832. ISSN 0163-5980. Dostupné z: https://doi.org/10.1145/1095809.1095832.

Příloha A

Obsah přiloženého paměťového média



diplomova_prace.pdf – soubor obsahující tuto diplomovou práci
diplomova_prace.zip – soubor obsahující zdrojové soubory k této diplomové práci
README.md -– soubor obsahující základní informace a popis obsahu pamětového média

Příloha B

Instalace a spuštění nástroje

Zdrojové soubory lze stáhnout:

1. z DVD v adresáři `Taipan`
2. z Github adresáře https://pajda.fit.vutbr.cz/testos/taipan

Překlad všech projektů lze provést příkazem `dotnet build`.

Spuštění všech testů lze provést příkazem `dotnet test`.

Ukázkový program pro detekci chyb uváznutí je po překladu spustitelný pomocí `.exe` souboru `samples/DeadlockDetectionSample/bin/Debug/DeadlockDetectionSample.exe`.

Ukázkový program pro detekci časově závislých chyb nad daty je po překladu spustitelný pomocí `.exe` souboru `samples/DataRaceDetectionSample/bin/Debug/DataRaceDetectionSample.exe`.

Všechny výše uvedené operace lze také provézt z vývojového prostředí Visual Studio po otevření souboru `Taipan.sln`

Nástroj je také dostupný volně ke stažení jako NuGet balíček na adrese https://www.nuget.org/packages/Taipan/.