



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**  
DEPARTMENT OF INTELLIGENT SYSTEMS

**DYNAMICKÉ ANALYZÁTORY PRO PLATFORMU  
SEARCHBESTIE**

DYNAMIC ANALYZERS FOR SEARCHBESTIE PLATFORM

**DIPLOMOVÁ PRÁCE**  
MASTER'S THESIS

**AUTOR PRÁCE**  
AUTHOR

**Bc. MARTIN JANOUŠEK**

**VEDOUCÍ PRÁCE**  
SUPERVISOR

**Ing. ALEŠ SMRČKA, Ph.D.**

**BRNO 2017**

## **Abstrakt**

Tato diplomová práce se zabývá problematikou testování paralelních programů. Popisuje chyby, které se mohou v paralelních programech nacházet a metody jejich testování. Podrobněji se zabývá metodou dynamické analýzy a věnuje se konkrétním dynamickým analyzátorům, jako jsou Fast-Track nebo analyzátor kontraktů. Zabývá se také popisem nástrojů, které mohou být pro testování paralelních programů použity. Popsány jsou zejména nástroje SearchBestie a RoadRunner, pro které je dále popsán návrh jednoho z dynamických analyzátorů. Jedná se o návrh analyzátoru kontraktů.

## **Abstract**

This master thesis deals with testing of parallel programs. It discusses errors, which can be founded in parallel programs, and methods for their testing. It deals with dynamic analysis and particular dynamic analyzers, such as FastTrack or Contract analyzer. It describes platforms for testing of parallel programs, such as SearchBestie and RoadRunner. The thesis also presents a design of Contract analyzer, as a new component of RoadRunner and Searchestie platform.

## **Klíčová slova**

Dynamická analýza, testování paralelních programů, analyzátor kontraktů, SearchBestie, RoadRunner

## **Keywords**

Dynamic analysis, testing of parallel programs, contract analyzer, SearchBestie, RoadRunner

## **Citace**

JANOUŠEK, Martin. *Dynamické analyzátoru pro platformu SearchBestie*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Smrčka Aleš.

# **Dynamické analyzátoru pro platformu SearchBestie**

## **Prohlášení**

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

.....  
Martin Janoušek  
9. května 2017

## **Poděkování**

Rád bych poděkoval mému vedoucímu Ing. Aleši Smrčkovi Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Testování vícevláknových programů</b>	<b>4</b>
2.1	Souběžnost a relace Happens before . . . . .	5
2.2	Logický čas . . . . .	6
2.3	Synchronizační nástroje . . . . .	6
2.4	Chyby v synchronizaci . . . . .	7
2.4.1	Uváznutí (deadlock) . . . . .	7
2.4.2	Blokování . . . . .	8
2.4.3	Stárnutí . . . . .	8
2.4.4	Časově závislá chyba nad daty . . . . .	8
2.4.5	Porušení atomicity . . . . .	9
2.4.6	Porušení kontraktu . . . . .	9
2.5	Metody testování . . . . .	9
2.5.1	Statická analýza . . . . .	9
2.5.2	Dynamická analýza . . . . .	10
2.5.3	Deterministické testování . . . . .	10
2.5.4	Zátěžové testování . . . . .	10
2.5.5	Vkládání šumu . . . . .	11
2.6	Nástroje pro testování Java programů . . . . .	11
2.6.1	RoadRunner . . . . .	11
2.6.2	SearchBestie . . . . .	13
2.6.3	Propojení SearchBestie a frameworku Roadrunner . . . . .	14
<b>3</b>	<b>Dynamické analyzátorý</b>	<b>16</b>
3.1	FastTrack . . . . .	16
3.1.1	Popis analyzátoru . . . . .	16
3.1.2	Zápis následovaný čtením nebo zápisem . . . . .	17
3.1.3	Čtení následované zápisem . . . . .	17
3.2	Analyzátor kontraktů . . . . .	17
3.2.1	Definice základního kontraktu . . . . .	17
3.2.2	Rozšíření kontraktu . . . . .	18
3.2.3	Dynamická analýza založena na LockSet algoritmu . . . . .	19
3.2.4	Dynamická analýza založena na relaci Happens before . . . . .	19

<b>4 Návrh analyzátoru kontraktů</b>	<b>21</b>
4.1 Specifikace požadavků . . . . .	21
4.2 Přehled částí systému . . . . .	22
4.3 Nástroj v Roadrunneru . . . . .	23
4.3.1 Vnitřní reprezentace . . . . .	24
4.3.2 Práce s šablonami a instancemi . . . . .	26
4.3.3 Vytvoření nové instance . . . . .	26
4.3.4 Krok advance . . . . .	28
4.3.5 Vektorový čas . . . . .	30
4.3.6 Informace o detekci porušení . . . . .	30
4.4 Podpora nástroje v SearchBestie . . . . .	30
4.4.1 Cover producer a úprava nástroje RRPlugin . . . . .	31
4.4.2 ContractRICreator a úprava SearchBestie . . . . .	31
<b>5 Implementace</b>	<b>34</b>
5.1 Implementační problémy . . . . .	34
5.1.1 Rozšíření RoadRunneru o parametry metod . . . . .	34
5.1.2 Rozšíření RoadRunneru o návratový parametr . . . . .	38
5.2 Implementační detaile . . . . .	38
<b>6 Testování</b>	<b>39</b>
6.1 Jednotkové testování . . . . .	39
6.2 Testování dynamického analyzátoru ve frameworku RoadRunner . . . . .	39
6.3 Testování SearchBestie . . . . .	68
<b>7 Závěr</b>	<b>69</b>
<b>8 SP: 1. Testování paralelních programů</b>	<b>70</b>
8.1 Paralelní program . . . . .	70
<b>Literatura</b>	<b>71</b>

# **Kapitola 1**

## **Úvod**

[TODO] S: motivace + aktualni stav + vlastni prinos [TODO obsah]: motivace, proc to vzniklo, co existuje, jaky je aktualni stav, co se deje v kazde kapitole - idealne odstavec vlastni prinos, kde napisu co jsem konkretne udelal

## Kapitola 2

# Testování vícevláknových programů

[Poznamky z konzultace] technologie, metody v dane oblasti (testovani vicevlaknovych programu), jak se delaji dyn. analyz. v java, roadrunner, searchbestie technologie, se kterymi prichazim do styku; metody v dane oblasti (testovani vicevlaknovych programu - zhruba popsat tuto oblast, kontrakty, java)

1. jak se dela dynamicka analiza v java
  2. popsat co to jsou kontrakty
- + popis roadrunneru a pouzitych technologii (searchbestie)
- Mozna by to chtelo cast nasledujiciho textu presunout do uvodu.

V dnešní době nalezneme v běžných počítačích procesory, které mají 2 nebo více jader. Aby jeden paralelní program mohl využívat více těchto jader současně, musí využívat jeden z následujících způsobů. Prvním způsobem je použití více procesů. V tomto kontextu lze proces chápat jako sekvenčně prováděný samostatný program ve vlastním adresovém prostoru. [20, 1].

Druhým způsobem je použití *vícevláknového* programu, ve kterém se nachází pouze jeden proces obsahující několik vláken<sup>1</sup>. V tomto kontextu proces chápeme jako obálku sady souběžně prováděných vláken v jednom adresovém prostoru. Přestože použití vláken přináší řadu výhod, jako například vyšší rychlosť přepínání kontextu a sdílení prostředků, přináší také problémy v podobě souběžného přístupu k témto sdíleným prostředkům. Tento přístup musí být řízen pomocí tzv. *synchronizačních prostředků*, jinak může docházet k chybám způsobených paralelizací. [1].

Základním požadavkem v dnešních operačních systémech<sup>2</sup> je souběžné provádění více programů. Vzhledem k tomu, že počet procesů a vláken, které mají běžet souběžně, je většinou vyšší než počet fyzických jader počítače, musí být výpočetní čas procesoru rozprostřen mezi všechny tyto procesy a vlákna. Rozprostření výpočetního času zajišťuje *plánovač*, který plánuje přepínání kontextu procesů a vláken. Plánovač je typicky preemptivní<sup>3</sup> a zohledňuje několik vlastností procesů a vláken, jako je například priorita, čas strávený na procesu, doba čekání nebo paměťové požadavky. Díky plánovači a dalším zdrojům nedeterminismu (obsluha přerušení, blokující operace, atd.) je paralelní provádění programu také nedeterministické, tj. procesy a vlákna mohou být různě *proloženy*. Tento nedeterminismus způsobuje, že chyby vzniklé paralelním prováděním programu se v jednom běhu mohou vyskytnout, zatímco v jiném ne. Z hlediska testování paralelních programů je tedy důležité ověřit, zda se chyba nevyskytuje v žádném možném proložení [17].

<sup>1</sup>V jazyce Java je program chápán jako proces s vlákny, přestože lze vytvořit i programy, které budou obsahovat více procesů. Nicméně každý proces obsahuje minimálně jedno vlákno [16].

<sup>2</sup>Dnešními operačními systémy jsou myšleny multiprogramové operační systémy, kde uživatel požaduje souběžný běh více programů a tedy souběžný běh více procesů a vláken.

<sup>3</sup>Při preemptivním plánování může být procesor procesu odebrán bez jeho přičinění (ukončení procesu, zahájení čekání, atd.).

## 2.1 Souběžnost a relace Happens before

V paralelním programu je důležité určit pořadí jednotlivých událostí napříč procesy (vlákny). Protože výpočet jednotlivých procesů a vláken probíhá asynchronně, musí být zaveden způsob, jak pořadí těchto událostí určit. Vztah mezi dvěma událostmi, které se provedly ve stejném nebo různém procesu (vlákně) se nazývá *relace kauzálního uspořádání* nebo taky relace *Happens before* [13]. Relace Happens before byla představena Leslieem Lamportem pro prostředí distribuovaných systémů, a proto zde bude uvedena v tomto obecnějším kontextu. Nicméně tato relace platí jak pro uspořádání událostí napříč vlákny, tak i procesy. V této kapitole tedy budou uvažovány procesy a vlákna za totožné. Následující vysvětlení této relace vychází ze zdrojů [13, 10].

Paralelní program se skládá z množiny  $n$  asynchronních procesů  $p_1, p_2, \dots, p_n$ . Procesy spolu navzájem mohou komunikovat pouze pomocí zasílání zpráv<sup>4</sup>. Dále platí, že procesy nemohou sdílet svůj globální čas<sup>5</sup> a výpočet procesů, včetně zasílání zpráv, probíhá asynchronně (předpokládejme, že každý proces běží na vlastním. Zprávu zaslanou procesem  $p_i$  procesu  $p_j$  označme jako  $m_{ij}$ . Vykonávání procesu se skládá ze sekvence atomických událostí. Události mohou být tří typů [10]:

- interní akce – mění vnitřní stav procesu,
- odeslání zprávy  $m$  ( $send(m)$ ) – mění vnitřní stav odesílajícího procesu,
- přijetí zprávy  $m$  ( $recv(m)$ ) – mění vnitřní stav přijímajícího procesu.

Dále nechť  $e_i^x$  označuje  $x$ -tou událost procesu  $i$ . Pak platí, že události v procesu jsou lineárně uspořádány podle indexu  $x$ . Toto lineární uspořádání událostí procesu  $i$  označme jako relaci  $\rightarrow_i$  a množinu všech akcí procesu  $p_i$  označme jako  $h_i$ . Relace  $\rightarrow_i$  značí kauzální závislost nad procesem  $p_i$ , tj. zápis  $e_i^x \rightarrow_i e_i^y$  značí, že událost  $e_i^x$  se stala před událostí  $e_i^y$  v procesu  $p_i$ . Dále definujme relaci  $\rightarrow_{msg}$ , která značí kauzální závislost mezi dvojicí událostí  $send(m)$  a  $recv(m)$ .

Nechť  $H = \bigcup_{i \in n} h_i$  značí množinu všech událostí vykonalých v paralelním programu. Pak *relace kauzálního uspořádání* (tj. **relace Happens Before**)  $\rightarrow$  je definována následovně [10]:

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

Výše definovaná relace *Happens before* umožňuje definovat uspořádání mezi dvěma událostmi. Pokud pro dvě události  $e_i$  a  $e_j$  platí  $e_i \rightarrow e_j$ , pak můžeme říct, že událost  $e_i$  předchází událost  $e_j$ . To znamená, že v systému nemůže nastat situace, kdy by se událost  $e_j$  stala před událostí  $e_i$ . Pokud pro stejné dvě události platí  $e_i \not\rightarrow e_j$ , pak událost  $e_i$  nepředchází událost  $e_j$ . Pro výše definované vztahy platí následující pravidla [10]:

$$\begin{aligned} e_i \not\rightarrow e_j &\Rightarrow e_j \not\rightarrow e_i \\ e_i \rightarrow e_j &\Rightarrow e_j \not\rightarrow e_i \end{aligned}$$

---

<sup>4</sup>Pod zasíláním zpráv je možné si představit také komunikaci pomocí sdílené paměti, přístupu k zámku, atd. Pro jednoduchost bude veškerá tato komunikace zobrazena jako zasílání zpráv mezi procesy.

<sup>5</sup>Globální čas procesu je pohled daného procesu na celkový čas (tj. čas všech procesů) v paralelním programu. Každý proces může mít ve stejnou chvíli jiný pohled na celkový čas.

Posledním možným vztahem dvou událostí je *souběžnost* (concurrency). Dvě události jsou souběžné, pokud platí [10]:

$$e_i \not\rightarrow e_j \wedge e_j \not\rightarrow e_i.$$

Tento vztah se značí jako  $e_i \parallel e_j$ . Z hlediska odhalování chyb v paralelních programech je tento vztah nejdůležitější. Pokud jsou dvě události souběžné (tj. konkurentní), pak nelze říci, která událost proběhne dříve. Může se stát, že při jednom spuštění bude nejprve provedena událost  $e_i$  a poté událost  $e_j$ , ale při jiném spuštění bude jejich pořadí opačné. Pokud budeme uvažovat reálný příklad, pak je tento problém například v souběžném zápisu více vláken do jedné sdílené proměnné (viz kapitola 2.4).

## 2.2 Logický čas

V předchozí kapitole bylo definováno uspořádání událostí v jednotlivých procesech. Každé takové události je přiřazeno časové razítko, které značí čas procesu v době provádění této události. Nejedná se ovšem o běžný fyzický čas, ale o čas *logický* [13]. Logický čas je dostačující k určení uspořádání událostí a není nijak závislý na fyzickém čase.

Systém logického času se skládá z časové domény  $T$  a logických hodin  $C$ , přičemž na množině  $T$  je definována relace částečného uspořádání  $<$ , která není ničím jiným, než výše definovanou relací Happens before. Logické hodiny jsou funkce, která mapuje události  $e$  z množiny všech událostí  $H$  na prvky z množiny  $T$  ( $C : H \rightarrow T$ ). Pomocí této funkce je možné určit relaci Happens before mezi dvěma událostmi  $e_i$  a  $e_j$  následovně:

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

Logický čas může být implementován následujícími způsoby [10]:

- *Skalární čas* (Scalar time) – proces si uchovává čas jako číslo  $d$ , které se po každé atomické události inkrementuje. Pokud proces odešle zprávu  $m$ , pak do zprávy vloží hodnotu čísla  $d$ . Při příjetí zprávy  $m$  proces přečte hodnotu času vloženého do zprávy  $d_{msg}$  a svoji hodnotu času  $d$  upraví následovně:  $d = \max(d, d_{msg})$ .
- *Vektorový čas* (Vector time) – každý proces uchovává vektor  $v$ , který má  $n$  položek ( $n$  je počet procesů). Při provedení interní události proces  $p_i$  inkrementuje hodnotu na pozici  $v_i$ , která označuje čas procesu  $p_i$ . Pokud proces odesílá zprávu, pak stejně jako u skalárního času přidává do zprávy právě tento svůj čas. Pokud proces přijme zprávu od procesu  $p_j$ , pak aktualizuje hodnotu  $v_j$ , která značí povědomí procesu  $p_i$  o lokálním času procesu  $p_j$ .
- *Maticový čas* (Matrix time) – rozšíření vektorového času, kde každý proces uchovává čtvercovou matici řádu  $n$ . Matice obsahuje povědomí o vektorovém čase každého procesu.

## 2.3 Synchronizační nástroje

Při provádění paralelního programu nastávají situace, kdy je nutné procesy nebo vlákna vzájemně řídit (synchronizovat). Pro řízení souběžného přístupu procesů nebo vláken lze použít následující synchronizační prostředky<sup>6</sup>:

---

<sup>6</sup>Pokud není uvedeno jinak, pak lze synchronizační prostředky použít jak pro procesy, tak pro vlákna.

- *Semafor* – může být *binární*, nebo *obecný*. Binární semafor obsahuje metody `init(value)`, `lock(value)` a `unlock()`. Proces čeká při zavolání metody `lock()`, dokud hodnota semaforu není 0 a pak jej nastaví na 1. Metoda `unlock()` nastaví hodnotu semaforu na 0 a od blokuje procesy čekající na metodě `lock()` stejného semaforu. Obecný semafor má oproti binárnímu kapacitu, která značí kolik jednotek zdroje chráněného semaforem je k dispozici. Binární semafor je obecný semafor s kapacitou 1. V Javě je obecný semafor implementován třídou `java.util.concurrent.Semaphore` [19, 15].
- *Mutex* – binární semafor určený pro vzájemné vyloučení. Při zamknutí mutexu je uložen jeho vlastník a pouze tento vlastník jej může odemknout. V Java SE neexistuje třída reprezentující mutex, nicméně lze pro tento účel upravit třídu `java.util.concurrent.Semaphore` [19, 15].
- *Bariéra* – umožňuje sadě procesů počkat na všechny ostatní. V Javě je implementována třídou `java.util.concurrent.CyclicBarrier` [15].
- *Zámek* – má podobný princip jako semafor, ale jedná se pouze o sdílenou proměnnou a tudíž může řídit přístup pouze mezi vlákny, nikoli procesy [17].
- *Monitor* – abstraktní datový typ, ve kterém jsou sdílené proměnné dostupné pouze přes operace monitoru (včetně jejich inicializace). Tyto operace monitoru jsou vzájemně vyloučené. Monitor lze implementovat v Javě například pomocí tříd `java.util.concurrent.locks.Lock` a `java.util.concurrent.locks.Condition` a `java.util.concurrent.locks.Lock`. Stejně tak lze použít klíčové slovo `synchronized` na libovolný objekt (`java.lang.Object`). Tento objekt obsahuje metody `wait()` (čekání) a `notify()` (upozornění ostatních, že již nemusí dále čekat) [19, 15].

Výše uvedené synchronizační prostředky představují pouze základní sadu synchronizačních nástrojů. Nicméně další takové nástroje jsou odvozeny právě z těchto nástrojů. Jak bylo zmíněno, synchronizační nástroje slouží k řízení souběžného přístupu. V další sekci budou ukázány základní chyby, které mohou nastat, pokud není synchronizace provedena správně, nebo dokonce vůbec.

## 2.4 Chyby v synchronizaci

Jak již bylo zmíněno v kapitole 8.1, chyby v paralelních programech mohou v jednom běhu nastat, zatímco v jiném nemusí. Je to dáno proložením běhu jednotlivých vláken nebo procesů. V následujícím textu budou prezentovány některé ze základních chyb tohoto typu včetně příkladů<sup>7</sup>, na kterých bude tento problém demonstrován. Příklady budou vysvětlovány nad vlákny, nicméně pro procesy je situace stejná.

### 2.4.1 Uváznutí (deadlock)

Prvním chybou, která je zde představena je *uváznutí*. Uváznutí je situace kdy vlákna čekají na stav, který by mohl nastat, pokud by jedno z těchto vláken mohlo pokračovat. Vlákna jsou takto blokovány navždy [17, 19].

Uváznutí může nastat v příkladu 2.1, pokud 1. vlákno zamkne zámek A na řádku 6 a ve stejnou chvíli 2. vlákno uzamkne zámek B na tomtéž řádku. V dalším kroku chce 1. vlákno uzamknout zámek B, ale ten je vlastněn 2. vláknem. Současně chce 2. vlákno uzamknout zámek A, ale ten je vlastněn 1. vláknem. Vlákna tedy čekají, než se zámky uvolní, což nemůže nikdy nastat. Pokud

---

<sup>7</sup>Příklady budou psané v pseudokódu vycházejícího z jazyka Java.

ovšem nastane takový průběh, že 1. vlákno získá oba zámky (řádky 6 a 7) a až poté chce 2. vlákno uzamknout zámek B (řádek 6), pak k uváznutí nedojde, protože vlákno A dokončí práci a poté uvolní oba zámky. 2. vlákno pak může oba zámky získat. V tomto příkladu byly popsány 2 průběhy. V jednom k chybě došlo a ve druhém ne. V dalších příkladech už budou demonstrovány jen takové proložení, ve kterých k chybě dojde.

Listing 2.1: Pseudokód příkladu uváznutí dvou vláken.

```

1 Lock A = ...;
2 Lock B = ...;
3
4 // 1. vlakno                                // 2. vlakno
5 void run() {                               void run() {
6     A.lock();                           B.lock();
7     B.lock();                           A.lock();
8     B.unlock();                         A.lock();
9     A.unlock();                         B.lock();
10 }                                         }

```

## 2.4.2 Blokování

Další chybou je *blokování*, které nastává, pokud vlákno čeká na stav, který generuje jiné vlákno, a toto čekání není nutné z hlediska synchronizace. K blokování také dochází pokud vlákno čeká na stav, který nemůže nikdy nastat [19].

V příkladu 2.2 může nastat situace, kdy jedno vlákno získá zámek A a už nikdy jej neuvolní. Druhé vlákno tak bude pořád blokováno.

Listing 2.2: Pseudokód příkladu blokování dvou vláken.

```

1 Lock A = ...;
2
3 // 1. vlakno                                // 2. vlakno
4 void run() {                               void run() {
5     while(true)                           while(true)
6     A.lock();                           A.lock();
7 }                                         }

```

## 2.4.3 Stárnutí

*Stárnutí* je podobný problém jako blokování, avšak čekání vlákna není shora omezeno. Vlákno čeká na splnění podmínky, která nemusí být nikdy platná v okamžiku testování, ale může nastat situace, kdy platná bude. Pak je vlákno uvolněno a může pokračovat. Pokud by k uvolnění nemohlo nikdy dojít, tak by se jednalo o blokování [19].

## 2.4.4 Časově závislá chyba nad daty

K časově závislé chybě nad daty může dojít, pokud dochází k souběžnému přístupu více vláken ke sdílené proměnné a alespoň jeden z přístupů je zápis [19].

V příkladu 2.3 se nachází sdílená proměnná `value`, která reprezentuje aktuální hodnotu. Dále se zde nachází dvě vlákna, kde 1. vlákno zapisuje aktuální hodnoty a 2. vlákno vypisuje aktuální hodnoty na výstup. Pokud jsou obě vlákna spuštěny současně, pak může dojít ke dvěma výsledkům. Na výstupu se objeví hodnota 5 nebo 8 podle toho, zda bude nejdříve aktuální hodnota vypsána, nebo aktualizována.

Listing 2.3: Pseudokód příkladu časově závislé chyby nad daty.

```
1 int value = 5;
2
3 // 1. vlakno                                // 2. vlakno
4 void run() {                               void run() {
5     value = 8;                           print(value);
6 }                                         }
```

## 2.4.5 Porušení atomicity

K porušení atomicity dojde, pokud vlákno získá stav nějaké sdílené proměnné, další výpočet vlákna závisí na tomto stavu a zároveň během tohoto výpočtu k této proměnné může přistoupit jiné vlákno a její stav změnit [19].

Následující příklad vychází z předchozího příkladu. K porušení atomicity například dojde, pokud 2. vlákno přečeť stav proměnné `value` a uloží si jej do proměnné `tmp`, následně 1. vlákno aktualizuje hodnotu `value` na hodnotu 8 a nakonec 1. vlákno provede také aktualizaci této proměnné, ale již na základě neplatného stavu uloženého v proměnné `tmp`. Na výstupu se tak může objevit některá z množiny hodnot: 6, 8, 9.

Listing 2.4: Pseudokód porušení atomicity.

```
1 volatile int value = 5;
2
3 // 1. vlakno                                // 2. vlakno
4 void run() {                               void run() {
5     value = 8;                           int tmp = value;
6 }                                         value = tmp + 1;
7                                         print(value);
8 }
```

## 2.4.6 Porušení kontraktu

Poslední zde uvedenou chybou je *porušení kontraktu* [14]. Kontrakt je sekvence veřejných metod objektu, která musí být vykonána atomicky, s ohledem na ostatní veřejné metody stejného objektu. Problematika kontraktů je dále rozebírána v kapitole 3.2.1.

... včetne popisu kontraktu ... více popsat

## 2.5 Metody testování

V předchozí kapitole byla vysvětlena problematika paralelních programů a chyb, které se v nich mohou vyskytovat. V této kapitole proto budou popsány metody testování, pomocí kterých lze chyby v paralelních programech odhalit. Největší důraz zde bude kladen na *dynamickou analýzu* a metodu *vkládání šumu*, neboť tyto dvě metody souvisí s dalšími částmi této práce.

### 2.5.1 Statická analýza

*Statická analýza* zkoumá software bez jeho spouštění. Tato metoda je například používána pro hledání syntaktických chyb a je typicky spouštěna před překladem. Nicméně i tuto metodu je možné využít pro testování paralelních programů. Nevýhodou této analýzy je produkování velkého množství *false alarmů*<sup>8</sup>, neboť statická analýza nemá k dispozici informace o konkrétních instancích

<sup>8</sup>*False alarm* je varování, že se může vyskytnout chyba, přestože ve skutečnosti nikdy vzniknout nemůže.

objektů. Existují ovšem metody založené na provádění testovaného softwaru, které se tento problém snaží odstranit. Jednou z nich je následující metoda *dynamické analýzy* [6].

### 2.5.2 Dynamická analýza

*Dynamická analýza* zkoumá software na základě jeho provádění, během kterého shromažďuje informace o jeho běhu (události a jejich uspořádání, stavy zámků, vláken, paměťových míst, atd.). Kromě hlášení chyb, které v daném běhu nastaly se dynamická analýza snaží extrapolovat nasbírané informace a odhalit chyby, které nenastaly. Přesto dynamická analýza nedokáže odhalit všechny chyby, ale pouze ty, které lze odvodit z běhů, kterých byla svědkem. Z tohoto důvodu bývá tato analýza spouštěna opakovaně a kombinována například s metodou *stochastického vkládání šumu* nebo *deterministického testování*. Díky témtu metodám je zvyšován počet testovaných proložení programu, a tím i pravděpodobnost nalezení konkurenčních chyb. Samotná analýza zatěžuje systém, což má podobný efekt jako vkládání šumu. Tento jev se označuje jako *noise effect* a je třeba s ním při analýze počítat. Dynamická analýza je většinou zaměřena pouze na určitý typ chyb a v takovém případě sbírá pouze informace související s tímto typem. U různých typů dynamické analýzy se rozlišují dvě vlastnosti *sound* a *precise*. Pokud dynamická analýza splňuje první vlastnost, pak nemůže přehlédnout chybu. Pokud splňuje druhou vlastnost, pak neprodukuje false alarmy. Nicméně platí, že dynamická analýza nemusí splňovat ani jednu z nich, tj. může přehlížet chyby a zároveň může produkovat false alarmy. Programy provádějící dynamickou analýzu se nazývají *dynamické analyzátory* (viz kapitola 3) a jejich příkladem jsou: *Eraser*, *GoldiLocks*, *FastTrack* nebo *DJIT+* [6, 18, 8, 9].

### 2.5.3 Deterministické testování

Deterministické testování je metoda založená na opakovém provádění testovaného softwaru, přičemž má plnou kontrolu nad jeho prováděním. K tomuto účelu je použitý *deterministický plánovač*, který je implementován například pomocí vkládání silného šumu. Cílem je otestovat co nejvíce možných proložení vláken. V každém kroku provádění softwaru analyzátor zkoumá, jaké možnosti v plánování mohou nastat a ty ukládá do stavového prostoru testovacích scénářů. Při dalších běžích se provádí další scénáře z tohoto stavového prostoru, které se opět dále větví. Z tohoto důvodu jsou zavedeny některé heuristiky, které omezují velikost stavového prostoru. Stavový prostor všech možných scénářů je příliš velký a u větších programů by testování trvalo příliš dlouho. Často se tak využívá například testování, kdy plánovač nechává běžet vždy pouze jedno vlákno a ostatní nechává pozastavené, nebo je například omezen maximální počet přepnutí kontextu. Případ, kde se testují všechny proložení se nazývá *full model checking* [6].

### 2.5.4 Zátěžové testování

*Zátěžové testování* je založeno na principu vytvoření nejhoršího možného prostředí, ve kterém aplikace může běžet. V souvislosti s hledáním konkurenčních chyb je to vytvoření velkého množství vláken, které budou navzájem soupeřit o sdílené zdroje. Tímto přístupem mohou být odhaleny některé chyby, ale s největší pravděpodobností se bude jednat o chyby častěji se vyskytující. Vzhledem tomu, že proložení prováděné testovaným softwarem jsou náhodné, může toto testování způsobovat opakované provádění již prozkoumaných proložení, přestože existují jiné proložení, které otestovány nebyly. Další nevýhodou je značné zatížení jak testovaného softwaru, tak prostředí, kde testování běží [6].

## 2.5.5 Vkládání šumu

Tento typ testování vkládá šumu (*noise*) [12] do prováděného kódu, který opožďuje vlákna a tak může dojít k proložení, které by jinak nastalo pouze velmi výjimečně. Stejně tak mohou být odhaleny chyby souběžného přístupu dvou instrukcí, které jsou v kódu dostatečně daleko od sebe a za normálních okolností by nebylo prakticky možné, aby se vykonaly souběžně. Tato chyba může být odhalena vložením dostatečné *silného*<sup>9</sup> šumu do jednoho z vláken. Pomocí metody vkládání šumu může být prozkoumáno velké množství scénářů v relativně krátké době. Náhodné vkládání šumu nemusí být příliš efektivní, protože může docházet k vkládání na místa, které nijak neovlivňují proložení vláken a šum by tak pouze zatěžoval systém. Lepších výsledků je dosaženo v případě použití některé heuristiky, která určuje vkládání šumu pouze na specifická místa v kódu (*noise seeding problem*) [6]. Stejně důležitým faktorem, jako je umístění šumu, je i vkládání vhodného typu šumu (*noise seeding problem*). Typ je dán sílou šumu a operací, která šum představuje [6].

V programovacím jazyce Java lze použít například funkce [15]:

- `yield()` – způsobí přepnutí kontextu (síla udává počet zavolání této funkce, než může vlákno pokračovat),
- `sleep()` – blokuje (uspí) vlákno po zadanou dobu (síla šumu),
- `wait()` – jako `sleep()` s rozdílem, že vlákno čeká na objektu *monitoru*.

## 2.6 Nástroje pro testování Java programů

Pro testování paralelních programů v jazyce Java existuje celá řada programů, a proto zde budou představeny především ty, jež souvisejí s dynamickou analýzou, která je předmětem této práce. Jedním z nejznámějších je *Java Pathfinder* (JPF) [2], který byl vyvinut v *NASA Ames Research Center*. JPF dokáže vykonávat testovaný program a ukládat, porovnávat a obnovovat stavy tohoto programu. Díky těmto vlastnostem je používaný jako tzv. *model checker*, ale lze v něm také na definovat dynamické analyzátoru. Podobné nástroje jako JPF jsou například *Bandera* nebo *CBMC*, který byl původně vyvinut pro C/C++, ale rozšířen o podporu jazyka Java.

Dalším nástrojem je projekt *IBM ConTest* [3] sloužící pro instrumentaci a dynamickou analýzu Java programů. Tento nástroj je velmi podobný dalšímu nástroji *RoadRuner* [9] (viz kapitola 2.6.1), který lze rovněž využít pro instrumentaci a dynamickou analýzu. V neposlední řadě existuje nástroj *Java Race Detector & Healer*<sup>10</sup> vyvinutý výzkumnou skupinou VeriFIT, který slouží pro detekci časově závislých chyb. Tento projekt využívá pro instrumentaci programu zmíněný IBM ConTest.

Jak bylo vysvětleno v kapitole 2.5.2, tak dynamická analýza nedokáže odhalit všechny chyby během jednoho běhu, a proto je nutné ji spouštět opakováně. Pro tento účel slouží nástroj *SearchBestie* [11], který byl opět vyvinut výzkumnou skupinou VeriFIT. SearchBestie společně s nástrojem RoadRunner jsou využívány v další části této práce a proto budou nyní popsány detailněji.

### 2.6.1 RoadRunner

*RoadRunner* [9] je framework navržený pro dynamickou analýzu vícevláknových Java programů, který je také napsán v jazyce Java. *RoadRunner* vkládá instrumentační kód do bytekódu testovaného programu, což umožňuje testovat programy i bez znalosti zdrojových kódů. Vložený instrumentační kód generuje *tok událostí*, které nastávají v testovaném programu. Těmito událostmi jsou například:

<sup>9</sup>Silným šum pozastaví provádění vlákna na delší dobu.

<sup>10</sup>Nástroj dostupný na adrese <http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/>

- události synchronizace na zámcích,
- přístup k proměnným,
- vytvoření/ukončení vláken,
- vstup/výstup do/z metod a další.

Tok události je zpracováván pomocí *nástrojů*, které mohou filtrovat události a je možné je skládat do tzv. *řetězce nástrojů*. Tímto způsobem lze skládat složitější dynamické analyzátoru z jednodušších kroků, kde každý krok je reprezentován jednodušším nástrojem.

Nástroje jsou tedy stavební kameny všech analyzátorů implementovaných v tomto frameworku. Při vytváření nástrojů je nutné rozšířit třídu obecného nástroje, tedy třídu `roadrunner.simple.Tool`. Tato třída definuje metody pro zpracování všech událostí (tzv. *handlery událostí*) z toku událostí. Každá tato metoda má jako parametr objekt reprezentující zachycenou událost. Pokud určitý nástroj nezpracovává události určitého typu, pak tyto události musí přeposlat dalšímu nástroji v řetězci nástrojů. Tabulka 2.1 obsahuje vybrané nástroje, které již jsou v tomto frameworku implementovány.

Název nástroje	Popis
ThreadLocal	Filtuje přístup k lokálním datům.
ReadOnly	Filtuje přístup k datům určeným pouze pro čtení.
ProtectingLock	Filtuje operace nad zámky, které jsou chráněny dalšími zámky.
LockSet	Detekuje časově závislé chyby nad daty s použitím LockSet algoritmu.
EraserWithBarrier	Detekuje časově závislé chyby nad daty s použitím LockSet algoritmu a analýzy bariér.
HappensBefore	Detekuje časově závislé chyby nad daty s použitím algoritmu VectorClock.
DJIT+	Detekuje časově závislé chyby nad daty s použitím optimalizovaného algoritmu VectorClock.
MultiRace	Detekuje časově závislé chyby nad daty s použitím hybridní LockSet/VectorClock analýzy.
Goldilocks	Detekuje časově závislé chyby nad daty s použitím rozšířeného LockSet algoritmu.
FastTrack	Detekuje časově závislé chyby nad daty s použitím FastTrack algoritmu.

Tabulka 2.1: Tabulka obsahující vybrané implementované nástroje v projektu RoadRunner [11]

Pro každý objekt *Thread*, který využívá JVM pro reprezentaci vláken, vytváří *Roadrunner* objekty typu *ShadowThread*, které jím přiřadí<sup>11</sup>. Stejným způsobem vytváří také objekty *ShadowLock* pro každý objekt použitý jako zámek a objekt typu *ShadowVar* pro každou lokaci v paměti. Poslední jmenovaný objekt je uložen v tzv. *shadow location*, která koresponduje s každou lokací v paměti. Zmíněné objekty pak mohou být využity pro uložení specifických informací důležitých pro konkrétní nástroj. Každý nástroj implementovaný v *RoadRunneru* by měl splňovat následující podmínky [9]:

1. Každý handler události musí vyvolat stejný handler v dalším nástroji v řetězci.
2. Pro identifikaci, který nástroj vlastní *shadow location*, musí nástroj obsahovat *shadow object* typu *T* a dále může ukládat do *shadow location* pouze objekty tohoto typu *T*.

---

<sup>11</sup>Objekt *ShadowThread* obsahuje referenci na originální objekt *Thread*

3. Metoda `makeShadowVar`<sup>12</sup> každého nástroje musí vracet objekt typu  $T$ .
4. Pokud je vyvolán `access` handler, tj. handler přístupu do paměti a *shadow location* je vlastněna tímto nástrojem, tj. *shadow location* obsahuje objekt typu  $T$ , pak nástroj musí provést jednu z následujících možností:
  - Udržet vlastnictví paměťového místa tím, že uloží/ponechá objekt typu  $T$  v *shadow location*.
  - Vzdá se vlastnictví paměťového místa pomocí metody `advance`, která nahradí *shadow location* objektem typu  $T_1$ , kde  $T_1$  značí typ *shadow* objektu následujícího nástroje. *Shadow location* už nikdy nesmí obsahovat objekt typu  $T$ .
5. Pokud je vyvolán `access` handler a paměťové místo není vlastněno tímto nástrojem, pak musí nástroj zavolat `access` handler následujícího nástroje.

Body 4. a 5. říkají, že *shadow location* každého paměťového místa může modifikovat pouze nástroj, který je jeho aktuálním vlastníkem.

### 2.6.2 SearchBestie

Platforma *SearchBestie* [11] je platforma určená pro hledání optimálních testů a jejich spouštění napísaná v jazyce Java. Využívá techniky prohledávání stavového prostoru, jež tvoří kombinace parametrů, se kterými jsou jednotlivé testy spouštěny. Lze ji mimo jiné použít pro nalezení optimálních testů při testování vícevláknových aplikací, čehož bude využito v této práci. *SearchBestie* nejprve využívala pro instrumentaci programů nástroj IBMContest, který byl později nahrazen nástrojem RoadRunner (viz 2.6.1). Jejím úkolem je hledat optimální parametry a s nimi spouštět testování programu právě prostřednictvím nástroje RoadRunner. *SearchBestie* se skládá z následujících modulů:

- *Manager* – řídí celý proces a nabízí pomocné funkcionality.
- *Search* – modul vybírá kombinaci parametrů a test, který má být v příštím kroku vykonán.
- *Executor* – vykonává testy s vybranými parametry a sbírá výsledky.
- *Storage* – ukládá výsledky testů.
- *Analysis* – analyzuje výsledky uložené modulem *Storage*.

*SearchBestie* na vstupu přijímá konfigurační soubor, ve kterém jsou definovány mimo jiné parametry ovlivňující vkládání šumu. Kombinace těchto parametrů vytváří stavový prostor možných konfigurací testů, který *SearchBestie* postupně prochází. Tyto parametry jsou:

- *NoiseFrequency* – udávající sílu šumu.
- *NoiseStrength* – udávající frekvenci šumu.
- *NoiseType* – udávající typ vkládaného šumu.

---

<sup>12</sup>Metoda `makeShadowVar` je zavolána při prvním přístupu do paměti na danou lokaci. Tj. při prvním přístupu k proměnné.

Na základě jednotlivých konfigurací (tj. stavů) ve stavovém prostoru vkládá RoadRunner šum do testovaného programu a tím ovlivňuje možné proložení vláken.

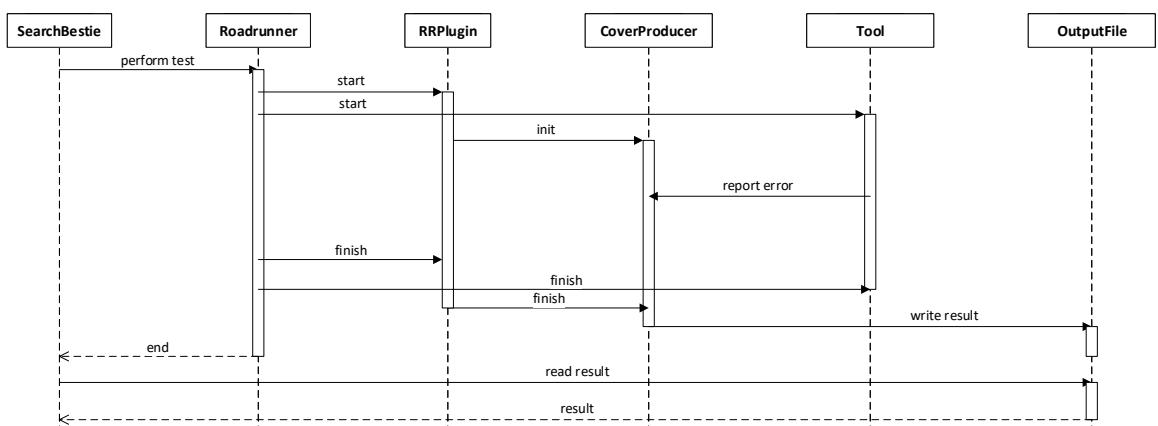
Ve vstupním souboru lze také definovat další parametry, mezi které patří:

- *Searcheng* – definuje, jakým způsobem se bude prohledávat stavový prostor.
- *Executor* – v tomto elementu se nachází nastavení pro modul *Executor*. Je zde možné například nastavit maximální počet spuštění testů.
- *Fitness* – definuje výpočet fitness funkce.
- *Javatest* – tento element definuje jaký typ testu bude spuštěn a nastavení tohoto testu. Vzhledem k použití RoadRunneru se jedná o *RoadRunnerTest* a nastavení všech parametrů, které jsou potřebné pro jeho spuštění.

### 2.6.3 Propojení SearchBestie a frameworku Roadrunner

Jak bylo zmíněno v předchozí kapitole, SearchBestie je nástroj určený pro hledání optimálních parametrů testů a jejich spuštění. Samotné testy jsou tedy prováděny samostatným nástrojem, kterým je v tomto případě RoadRunner. SearchBestie následně sbírá výsledky z těchto testů a na jejich základě vytváří stavový prostor. Pro propojení je důležitá zejména třída *RRPlugin* v RoadRunneru, která musí být použita jako první nástroj v řetězci nástrojů. Tato třída obsahuje tzv. *CoverProucery*, které zaznamenávají informace o chybách, vyskytujících se v testovaném programu. Při ukončení testu CoverProducery zapíší všechny nasbírané informace o chybách do výstupních souborů, které jsou následně načteny v Searchbestie a jejich obsah je dále zpracován. Každý takový CoverProducer může zaznamenávat různé informace a ty musejí být zpracovány různým způsobem. Z toho důvodu musí existovat odpovídající tzv. *ResultItemCreator*, který dokáže tyto výsledky zpracovat.

V diagramu refkomunikace-obecna je znázorněna zjednodušená vzájemná komunikace mezi těmito komponentami. SearchBestie nejprve spustí RoadRunner s patřičnými parametry pomocí metody *performTest*. Následně RoadRunner spustí RRPlugin a další nástroje (včetně nástroje *Tool*), zadané v parametrech. RRPlugin pro každý další nástroj vytvoří patřičný CoverProducer. Jakmile dojde k chybě, která byla detekována některým nástrojem (*Tool*), pak tento nástroj reportuje chybu odpovídajícímu CoverProduceru. Na konci testování jsou ukončovány jednotlivé nástroje a v této chvíli dá RRPlugin pokyn všem CoverProducerům, aby zapsaly všechny nasbírané informace do výstupního souboru. Jakmile SearchBestie dostane informaci o ukončení RoadRunneru, načte a zpracuje informace z výstupních souborů jednotlivých CoverProducerů.



Obrázek 2.1: Komunikace v systému.

## Kapitola 3

# Dynamické analyzátory

Dynamické analyzátorы jsou programy provádějící dynamickou analýzu, přičemž každý z nich je zaměřen na hledání určitého typu chyb. Typickým problémem je hledání časově závislých chyb nad daty. Analyzátorů, které hledají tento typ chyb je hned celá řada. Nejjednodušším z nich je algoritmus *Eraser*, který zkoumá program s pomocí množin zámeků držených jednotlivými vlákny, přičemž princip tohoto algoritmu dále rozšiřuje další analyzátor *Goldilocks*. Dalšími analyzátorы jsou například *DJIT+* nebo *FastTrack*, které hledají chyby za pomoci vektorového času. Příkladem dynamického analyzátoru, který hledá jiný typ chyby může být analyzátor kontraktů. Právě tento analyzátor společně s analyzátem *FastTrack* bude dále více popsán [8, 18, 7].

### 3.1 FastTrack

#### 3.1.1 Popis analyzátoru

*FastTrack* [8] analyzátor patří do skupiny *precizních*<sup>1</sup> analyzátorů a je určen pro odhalování časově závislých chyb nad daty. Využívá adaptivní reprezentaci relace Happens before, díky které dokáže významně zvýšit svou rychlosť oproti jiným precizním analyzátorům, jako jsou například BasicVC nebo DJIT+. Běžné precizní dynamické analyzátorы ukládají pro každé vlákno vektorový čas posledního zápisu do každé proměnné  $x$ . Vektorový čas je  $n$ -tice čísel, kde  $n$  je počet vláken (viz kapitola 2.2). Všechny operace nad vektorovým časem mají tudíž složitost  $O(n)$ . Oproti tomu FastTrack může ukládat pouze informaci o posledním zápisu do každé proměnné  $x$  napříč všemi vlákny, a to v případě, že všechny zápisy do proměnné  $x$  jsou uspořádané relací Happens before. Tato informace, nazývaná *epocha*, se skládá z času posledního zápisu a identifikátoru vlákna, ve kterém zápis proběhl. Veškeré operace nad touto epochou je následně možné provést v konstantním čase  $O(1)$ . Stejně jako pro zápis ukládá FastTrack informaci pouze o posledním čtení každé proměnné  $x$ , pokud jsou opět všechny předchozí čtení uspořádány pomocí relace Happens before. Pokud nastane případ, kdy některé čtení nebo zápisy nejsou touto relací uspořádány, FastTrack ukládá informaci o časech vykonání těchto operací pomocí vektorových hodin [8].

Protože nutnost ukládat plný vektorový čas je pouze v minimálním množství situací, dokáže FastTrack díky použití epoch snížit režii spojenou s analýzou z  $O(n)$  až na  $O(1)$ . Jak bylo popsáno výše, FastTrack analyzátor ukládá informaci o posledním zápisu pomocí epochy. Vlákna ovšem pro reprezentaci logického času používají vektorový čas a proto je nutné tyto dvě formy umět porovnat. *Epocha* je definována jako dvojice  $c@t$ , kde  $c$  je logický čas vlákna  $t$  a platí, že tato epocha před-

<sup>1</sup>Dynamický analyzátor je precizní, pokud splňuje vlastnost *precise* (viz kapitola 2.5.2)

chází vektorový čas  $V$  (označováno jako  $c@t \leq V$ ) v relaci Happens before, jestliže platí  $c \leq V(t)$  [8].

Analyzátor rozlišuje následující 3 typy situací, které mohou nastat při přístupu ke sdílené proměnné [8]:

- čtení s následným zápisem,
- zápis s následným čtením,
- zápis následující dalším zápisem.

### 3.1.2 Zápis následovaný čtením nebo zápisem

Detekování dvou konkurenčních zápisů je pomocí porovnání epochy s vektorovým časem jednoduché. Pokud označíme epochu posledního zápisu do proměnné  $x$  jako  $W_x$ , pak ve chvíli následujícího zápisu do této proměnné v čase  $V$  stačí porovnat tento vektorový čas s epochou  $W_x$ . Stejným způsobem lze také rozhodnout, zda mohlo dojít k časově závislé chybě nad daty při zápisu následovaném čtením. Ve chvíli čtení proměnné  $x$  v čase  $V$ , se opět porovná tento vektorový čas s epochou posledního zápisu  $W_x$ , tedy zda platí:  $W_x \leq V$ . Pokud tato rovnost platí, pak nedošlo k časově závislé chybě nad daty [8].

### 3.1.3 Čtení následované zápisem

Detekování možného současněho přístupu při čtení s následované zápisem není v analyzátoru Fast-Track tak jednoduché, jako předchozí dva případy. Důvodem je to, že jednotlivá čtení libovolné proměnné  $x$  nemusejí být totálně uspořádána, jako tomu je u zápisu<sup>2</sup>. Pokud nastane případ, že některá čtení nejsou totálně uspořádána, pak je nutné uchovat celý vektorový čas tohoto čtení. Tento případ může nastat pouze v případě, že je proměnná  $x$  sdílena a čtení není chráněno zámkem. V ostatních případech postačí uchování epochy. To zda došlo k současnemu přístupu lze opět určit podobně jako v předchozích případech, ale může nastat stav, kdy budou porovnávány dva vektorové časy, konkrétně  $R_x$  (čas posledního čtení proměnné  $x$ ) a aktuální čas zápisu  $V$ . V tomto případě bude porovnání časově nejnáročnější, nicméně k této situaci dochází pouze velmi zřídka [8].

## 3.2 Analyzátor kontraktů

### 3.2.1 Definice základního kontraktu

*Kontrakt* [14] byl původně definován jako sekvence příkazů s definovanými podmínkami. Pokud je sekvence vykonána bez splnění těchto podmínek, dochází k porušení tohoto kontraktu. *Kontrakt pro současnost* [4] je oproti tomu protokol přístupu veřejných služeb modulu (tj. veřejných metod). Každý modul může mít definován vlastní kontrakt, který obsahuje množinu sekvencí služeb (metod). Podmínkou splnění kontraktu je atomické vykonání těchto sekvencí, pokud jsou vykonávány nad stejným objektem.

Pro další popis je nutné zavést následující značení, které vychází z článků [7] a [5]. Nechť je množina jmen veřejných metod modulu označena jako  $\Sigma_M$ , dále kontrakt jako množina klauzulí  $R$ , kde každá klauzule  $\varrho \in R$  je regulární výraz nad  $\Sigma_M$ . K porušení kontraktu dojde, pokud sekvence reprezentovaná kontraktem není provedena atomicky nad stejným objektem  $o$ , tj. sekvence kontraktu je proložena alespoň jednou metodou z množiny  $\Sigma_M$ .

---

<sup>2</sup>Pokud dojde k porušení totálního uspořádání u zápisu, pak byla nalezena časově závislá chyba nad daty.

Příkladem kontraktu (označme jej  $\varrho_1$ ) může být sekvence metod `indefOf` a `get` nad objektem typu *seznam*. Pokud by tato sekvence byla proložena například metodou `add`, která by vložila prvek na první místo seznamu, pak by index hledaného prvku vrácený metodou `indexOf` již nebyl platný a následné použití metody `get` by vrátilo jiný prvek seznamu. Tento kontrakt lze zapsat jako:

$$\varrho_1 : \text{indefOf } \text{get}.$$

### 3.2.2 Rozšíření kontraktu

Rozšíření publikované v [5] vychází z předchozí definice kontraktů, přičemž spojuje kontrolu atomicity sekvencí kontraktů s definicí metod, které tuto sekvenci nesmí porušit. Znamená to tedy, že sekvence metod musí být atomická pouze s ohledem na proložení pouze určenou množinu metod. Dalším rozšířením je přidáním parametrů do jednotlivých metod.

Uvažujme nejprve *rozšíření kontraktů o parametry*. Pokud se znovu zaměříme na příklad kontraktu  $\varrho_1$ , pak je viditelné, že porušení kontraktu způsobí chybu pouze v případě, kdy je návratová hodnota metody `indefOf` použita jako parametr metody `get`. Tento kontrakt s rozšířením o parametry lze zapsat jako:

$$\varrho'_1 : X = \text{indefOf}(\_) \text{ get}(X).$$

Dále uvažujme *rozšíření o kontextové informace* (tj. rozšíření o metody, které nesmí sekvenci kontraktu porušit). Uvažujme opět příklad kontraktu  $\varrho_1$  a základní definicí kontraktu. Tato definice říká, že sekvence kontraktu nesmí být proložena žádnou metodou z množiny veřejných metod modulu  $\Sigma_M$ . Uvažujme tedy například metody `indexOf` a `remove` z množiny  $\Sigma_M$ . Pokud bude sekvence kontraktu  $\varrho_1$  proložena voláním metody `indexOf`, pak k žádné chybě nedojde. Pokud ovšem bude proložena voláním metody `remove`, pak k chybě může dojít. Toto rozšíření tedy umožňuje definovat množinu sekvencí metod (sekvence těchto metod bude dále označována jako *spoiler*), vůči kterým musí být sekvence kontraktu (dále označovaná jako *target*) provedena atomicky [5].

Nechť je tedy  $\mathbb{R}$  množina všech *targetů*, kde každý target  $\varrho \in \mathbb{R}$  je regulární výraz nad množinou  $\Sigma_M$ . Nechť  $\mathbb{S}$  je množina *spoilerů*, kde každý spoiler  $\sigma \in \mathbb{S}$  je regulární výraz nad  $\Sigma_M$ . Dále označme abecedu všech targetů jako  $\Sigma_R \subseteq \Sigma_M$  a všech spoilerů jako  $\Sigma_S \subseteq \Sigma_M$ . Pak *kontrakt* je definován jako relace  $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ , kde pro každý target je definována množina spoilerů, které mohou vyvolat porušení atomicity.

K porušení kontraktu dojde, pokud je nějaká sekvence prováděných metod  $r$ , odpovídající targetu  $\varrho \in \mathbb{R}$  vykonána nad objektem  $o$ , plně proložena celou sekvencí metod  $s$ , odpovídající nějakému spoileru  $\sigma \in \mathbb{C}(\varrho)$ , která je vykonána nad stejným objektem  $o$ . Sekvence targetu  $r$  je plně proložena sekvencí spoileru  $s$  pokud začátek vykonávání  $r$  začíná před začátkem vykonávání  $s$  a současně konec vykonávání  $s$  předchází konec vykonávání  $r$ . [5]

V další části této práce budou uvažovány obě tyto rozšíření, přičemž značení kontraktu bude například pro výše zmínovaný příklad následující:

$$\varrho''_1 : X = \text{indefOf}(\_) \text{ get}(X) \rightsquigarrow \text{remove}(\_)$$

Kontrakty lze detektovat jak na základě statické analýzy, tak na základě dynamické analýzy. Vzhledem k tématu této práce zde bude popsána pouze dynamická analýza, která byla navržena jak pomocí LockSet algoritmu, tak pomocí relace Happens before. Tato dynamická analýza byla také implementována v projektu ANaConDA, která zkoumá paralelní programy psané v jazyce C/C++.

### 3.2.3 Dynamická analýza založena na LockSet algoritmu

Prvním možným způsobem detekce kontraktů pomocí dynamické analýzy je analýza založená na množině zámků držených jednotlivými vlákny (*LockSet-based*), jejíž typickým představitelem je algoritmus Eraser. Díky tomuto typu dynamické analýzy je možné rozšířit proložení, které se událo, a tím nalézt porušení kontraktu, které se přímo nestaly. Tento způsob byl představen v [7] a dokáže pracovat pouze ze základní definicí kontraktu.

V této analýze je nejdříve nutné detektovat kontrakty, které se vyskytují ve vykonávaném programu. Tohoto je dosaženo pomocí konečných automatů, kde pro každou sekvenci každého kontraktu je vytvořen odpovídající konečný automat. Každé vlákno obsahuje množinu instancí těchto automatů, kde instance odpovídají aktuálním nedokončeným sekvencím kontraktů, které se nachází v aktuálním stavu programu. Pokud je zavolána metoda  $m \in \varrho_M$ , pak je nad každou instancí automatu proveden pokus o postup do dalšího stavu pomocí této metody  $m$  (tentotok je označován jako *advance*). Pokud je nový stav automatu stavem konečným, pak je detektován výskyt sekvence kontraktu. Pokud je metoda  $m$  metodou, kterou začíná některá sekvence kontraktu, pak je vytvořena nová instance kontraktu [7].

Pokud je sekvence kontraktu detektována, je nutné ověřit, zda byla prováděna atomicky. Pomocí informací o zámcích, držených jednotlivými vlákny během provádění kontraktu, lze detektovat, zda byla celá sekvence chráněna alespoň jedním zámkem. Pokud byla chráněna alespoň jedním zámkem, pak je pravděpodobné, že byla provedena atomicky. Pravděpodobné to je proto, že může nastat případ, kdy tomu tak nebude. Konkrétně se jedná o situaci, kdy dojde k proložení dvou sekvencí kontraktu nad stejným objektem ve dvou různých vláknech a obě tyto sekvence budou chráněny zámkem, ale tento zámek bude odlišný. Pokud nastane tento případ, mohou být sekvence prováděny souběžně, a tím dojde k porušení kontraktu. Tato situace je diskutována v článku [7] a jako řešení je navržena dynamická analýza založena na relaci Happens before.

### 3.2.4 Dynamická analýza založena na relaci Happens before

Dynamická analýza založena na relaci Happens before, představena v článku [5], je navržena pro rozšířenou definici kontraktu s kontextovými informacemi. Tato analýza využívá Happens before relaci (viz 2.1), kde jako komunikace mezi vlákny jsou uvažovány operace *acquire* a *release* nad stejnými zámkami a operace *fork* a *join*. Tato analýza umožňuje detekci porušení kontraktů za běhu díky technice nazvané *trace window*. Tento koncept umožňuje neuchovávat celou sekvenci všech provedených metod, nýbrž pouze podmnožiny této sekvence (dále označováno jako *okno běhu*), která se postupně pohybuje v průběhu vykonávání programu. Cílem tohoto konceptu je uchovávat okno o co nejmenším počtu naposled vykonaných metod. Události se přidávají do okna běhu  $v$  v momentě, kdy se vyskytnou, a odebírají, pokud již nejsou potřeba. Odebírání je definováno tak, že všechny události z určité instance targetu (spoileru) se mohou odebrat, pokud již nepatří do žádné jiné aktuálně sledované instance targetu nebo spoileru. Velikost okna tak závisí pouze na počtu instancí targetů a spoilerů, nikoli na délce běhu programu. Aby bylo možné odstraňovat z tohoto okna události, je nutné odstraňovat nepotřebné instance targetů a spoilerů. Pro odstraňování těchto instancí jsou definovány následující pravidla [5]:

- Odstranění instance spoileru  $s$  je bezpečné, pokud porušení kontraktu, které může být odhaleno pomocí instance  $s$ , lze také odhalit bez této instance  $s$ .
- Pro každé vlákno a každý spoiler je nutné uchovávat pouze poslední instanci spoileru.

- Pokud existují dvě instance  $r_1, r_2$  targetu  $\varrho \in \mathbb{R}$  takové, že  $end(r_1) \rightarrow_{hb} start(r_2)$ , pak je bezpečné odstranit instanci  $r_1$ , pokud  $s$  začíná až za *oknem běhu* nebo pokud  $start(s) \rightarrow_{hb} start(r_1)$ .
- Pokud existují dvě instance  $r_1, r_2$  targetu  $\varrho \in \mathbb{R}$  takové, že  $end(r_1) \rightarrow_{hb} start(r_2)$ . Nechť  $s$  je instance spoileru  $\sigma \in \mathbb{S}$ , kde  $(\varrho, \sigma) \in \mathbb{C}$ , pak je bezpečné odstranit instanci  $r_1$  s ohledem na  $s$  pokud platí že  $start(s) \in v \wedge end(s) \notin v$  a zároveň  $start(s) \not\rightarrow_{hb} start(r_2)$ .
- Obecně platí, pro každý target  $\varrho \in \mathbb{R}$  je uchováno  $|T| + 1$  instancí, kde  $T$  značí množinu běžících vláken v *okně běhu*. Pro každé vlákno musí být uchována jedna instance a také 1 navíc pro vlákno, které může být potencionálně vytvořeno.

## Kapitola 4

# Návrh analyzátoru kontraktů

[Poznamky z konzultace] Toto musí být uplna kapitola, bude nejvetsi. Tady popisu kompletni navrh tak, aby to podle toho slo implementovat navrh celeho reseni - kazdou cast navrhnu tak jak by se to melo resit musí to být uplne tak aby to nekdo mohl vzít a podle toho to mohl naimplementovat. Narazel by na nejake problemy a o tom bude ta dalsi kapitola. do navrhu same diagramy jak architekturnalni, tak behavioralni (kdo s kym jak jedna - state chart nebo sequence diagram (je jedno kterym))

V této kapitole bude popsán návrh dynamického analyzátoru kontraktů, který odpovídá popisu ... . Jedná se tedy o analýzu kontraktů s parametry, přičemž pro detekci většího počtu chyb byl využitý princip vektorových hodin.

### 4.1 Specifikace požadavků

Samotný analyzátor je navržen jako nástroj ve frameworku RoadRunner (viz. ...následující kapitola...), nicméně musí být také upravena platforma SearchBestie, aby tento analyzátor byla schopná spustit a zpracovat jeho výsledky. Z tohoto důvodu budou v následujících požadavcích také požadavky spojené s touto integrací.

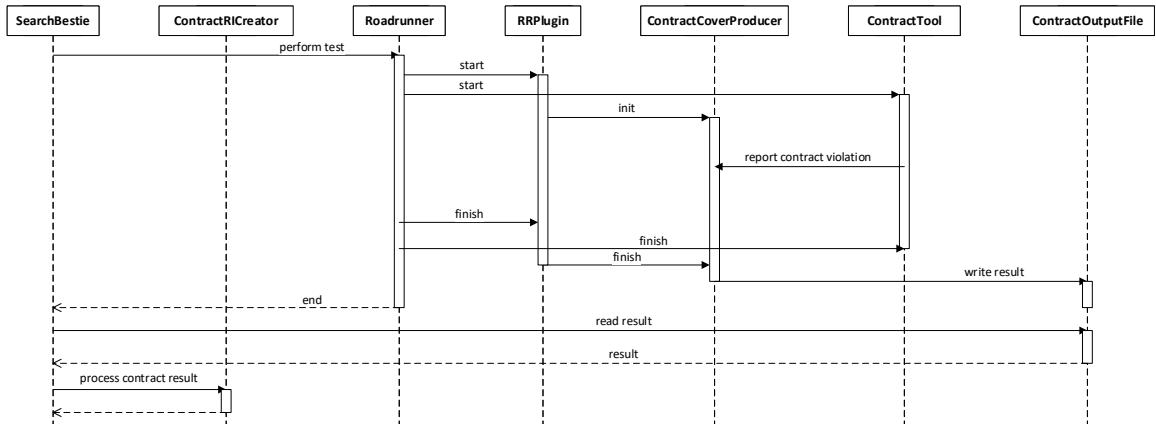
1. Obecné požadavky
  - 1.1. Analyzátor musí být implementován projektu RoadRunner.
  - 1.2. Roadrunner s analyzátem musí být spustitelný z příkazové řádky.
  - 1.3. Roadrunner s analyzátem musí být spustitelný ze SearchBestie.
  - 1.4. RoadRunneru musí uložit informace o nalezených chybách do výstupního souboru.
  - 1.5. SearchBestie musí zpracovat tyto výstupní informace.
  - 1.6. SearchBestie musí na základě nalezených chyb prohledávat stavový prostor parametrů testů.
2. Funkční požadavky
  - 2.1. Analyzátor získá definici kontraktů z konfiguračního souboru.
  - 2.2. Musí být definován formát konfiguračního souboru.
  - 2.3. Cesta ke konfiguračnímu souboru bude předána analyzátoru jako parametr v příkazové řádce.

- 2.4. Cesta ke konfiguračnímu souboru bude předána SearchBestie ve vstupním souboru, která ji předá RoadRunneru.
- 2.5. Analyzátor musí být schopný zpracovat konfigurační soubor a uložit si kontrakty do vnitřní reprezentace.
- 2.6. Porušení konaktu musí být detekováno v následujících případech:
  - i. Analyzátor musí být schopný detekovat porušení konaktu, ke kterým skutečně došlo.
  - ii. Analyzátor musí být schopný detekovat porušení konaktu, ke kterým skutečně nedošlo, ale lze pomocí vektorových hodin odvodit, že k nim může dojít.
  - iii. Analyzátor nesmí detekovat porušení konaktu v žádném jiném případě (pokud k němu nedošlo).
- 2.7. Analyzátor musí podporovat kontrakty různých typů:
  - i. K jednomu targetu může být zadán jeden spoiler.
  - ii. K jednomu targetu může být zadáno více spoilerů.
  - iii. Targetů může být zadáno v konfiguračním souboru více.
- 2.8. Metody v targetech a spoilerech mohou být:
  - i. Bez parametrů,
  - ii. s parametry,
  - iii. s návratovým parametrem.
- 2.9. Metody musí být jednoznačně identifikovány pomocí fully qualified name..
- 2.10. Stejně označený parametr v sekvenci metod je považován za shodný parametr.
- 2.11. Stejně označený parametr v targetu a jemu připojeném spoileru musí být považován za shodný parametr.
- 2.12. Parametr zadaný znakem \_ musí být ignorován.

[Poznamky z konzultace] S: co presne je mojim cilem udelat, kazda odrazka rika co to ma umet, ma to byt spustitelne v prikazove radce, contract - co to ma umet identifikovane odrazky (cislama, viceurovne - ne vic nez tri urovne) - kazda odrazka bude definovat co to ma umet. I z uzivatelskeho hlediska - napriklad ze to ma byt spustitelne z prikazove radky, ma to umet kontrakty (v odrazkach jak presne), dal ma to umet pracovat se search bestie.

## 4.2 Přehled částí systému

Jak bylo zmíněno v kapitole, pro nový analyzátor je nutné vytvořit několik nových komponent. Konkrétně se jedná o samotný analyzátor (ContractTool) a jemu přidružený CoverProducer (konkrétně ContractCoverProducer), přičemž obě tyto části se budou nacházet v projektu RoadRunner. Dále je nutné vytvořit podporu tohoto nástroje v SearchBestie, kde se o zpracování nasbíraných výsledků bude starat ResultItemCreator (konkrétně ContractRICreator). Na diagramu 4.1 je znázorněna komunikace mezi těmito komponentami, přičemž si lze všimnout, že tato komunikace odpovídá standardní komunikaci mezi SearchBestie a Roadrunnerem prezentované v kapitole 2.6.3



Obrázek 4.1: Konkrétní komunikace mezi komponentami dynamického analyzátoru kontraktů a Searchbestie.

### 4.3 Nástroj v Roadrunneru

Nástroj ContractTool (dále jen CT) ve Frameworku Roadrunner je hlavní částí analýzy kontraktů, neboť se samotná analýza provádí právě zde. CT rozšiřuje třídu Tool frameworku Roadrunner, díky čemuž může zachytávat potřebné události. CT využívá události: init, exit, makeShadowVar, acquire, release, create, preStart a postJoin. První metoda (tj. init) je použita pro inicializaci nástroje, další dvě jmenované metody, exit a makeShadowVar, jsou důležité pro detekci porušení konaktu (viz ...) a ostatní metody slouží pro práci s vektorovým časem (viz ...). CT dále definuje parametr `contractFile`, ve kterém musí být zadána cesta ke konfiguračnímu souboru kontraktů. Takto předaný konfigurační soubor je zpracován právě v metodě init za pomoci *Parseru*.

#### Konfigurační soubor

Parser slouží pro zpracování obsahu konfiguračního souboru, ve kterém se nachází definice konaktu, a vytvoření ekvivalentní vnitřní reprezentace. Formát konfiguračního souboru vychází z notace použité v definici kontraktů (viz 3.2) a lze jej popsat gramatikou 4.1. Symbol `<method_name>` zastupuje tzv. *fully qualified name*, které v Javě běžně využíváno pro jednoznačnou identifikaci. V gramatice se rovněž vyskytuje symbol `|`, který je symbolem terminálním, nikoli znakem oddělujícím dvě alternativní přepisovací pravidla. Dále gramatika definuje speciální parametr `_`, který je použit pro ignorovaný parametr (viz 4.1).

Listing 4.1: Gramatika generující validní konfigurační soubor.

```

<contract> → <target> <- { <spoilers> } <contract>
<contract> → ε
<spoilers> → <spoiler> <spoilers'>
<spoilers> → <spoiler>
<spoilers'> → | <spoiler> <spoilers'>
<spoilers'> → ε
<target> → <method sequence>
<spoiler> → <method sequence>
<method sequence> → <method> <method sequence>
<method sequence> → ε
<method> → <ret parameter><method_name>(<parameters>)
<parameters> → <parameter> <parameters'>
<parameters> → <parameter>

```

```

<parameters> → ε
<parameters'> → ,<parameter><parameters'>
<parameters'> → ε
<ret parameter> → <parameter>:
<ret parameter> → ε
<parameter> → a-zA-Z0-9
<parameter> → _
<method_name> → a-zA-Z0-9. $

```

Příklad 4.2 ukazuje obsahu souboru `test1`, který může být vygenerován výše zmíněnou gramatikou a je jedním ze souborů využitých při testování. V tomto souboru je definován kontrakt, kde targetem je sekvence dvou metod (m1 a m2) a spoilerem je jediná metoda (m1).

Listing 4.2: Příklad konfiguračního souboru.

```

cz.vutbr.fit.Test1$Subject.m1() cz.vutbr.fit.Test1$Subject.m2() <-
{ cz.vutbr.fit.Test1$Subject.m1() }

```

Takto definovaný konfigurační soubor je zpracován Parserem do vnitřní reprezentace, která je popsána v následující kapitole.

Pro předání cesty ke konfiguračnímu souboru do nástroje Roadrunner je nutné definovat nový parametr `-contractFile`, který je přidán jako tzv. *Command line option* do nástroje ContractTool. Tento parametr je při spuštění nástroje ContractTool vždy vyžadován.

### 4.3.1 Vnitřní reprezentace

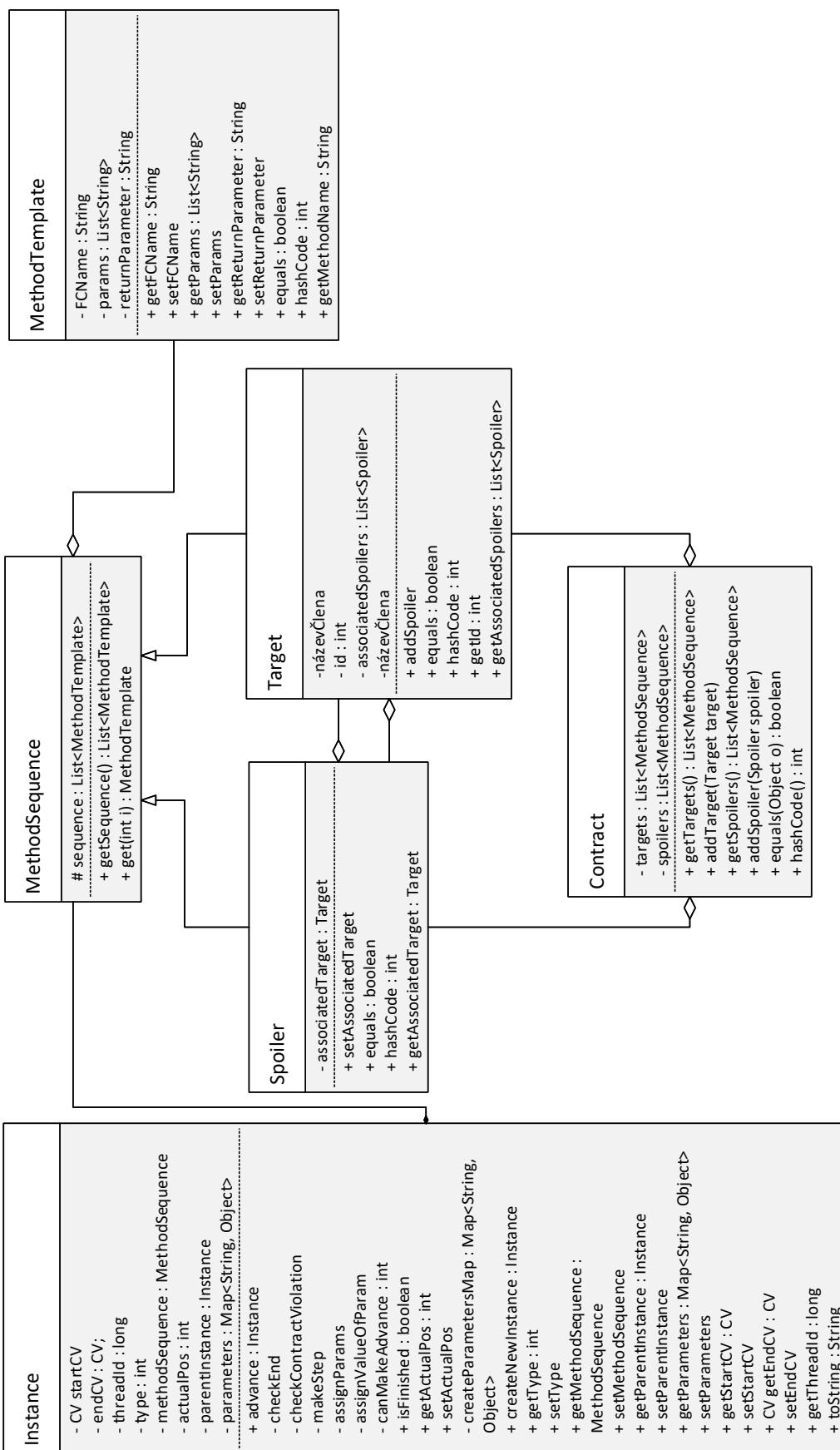
Analyzátor si musí uchovat definici kontraktu, která obsahuje definici targetů a jím asociovaných spoilery. Tyto targety a spoilery jsou používány jako šablony, ze kterých se vytvářejí jejich instance. Targety a spoilery tedy obsahují neměnící se informace, kdežto jejich instance svůj obsah mění. Vztah těchto částí je znázorněn na diagramu 4.2

Definici kontraktu reprezentuje třída `Contract`, která se v systému nachází vždy pouze jednou. Tato třída obsahuje instance tříd `Target` a `Spoiler`, které dědí od střídy `MethodSequence`. `MethodSequence` obsahuje sekvenci instancí třídy `MethodTemplate`, které reprezentují šablonu metody ze zadání. Tato třída obsahuje důležité informace o metodě, která má být dynamickým analyzátorem sledována. Jedná se o jméno metody, konkrétně *fully qualified name*, jména parametrů a jméno návratového parametru<sup>1</sup>. Poslední nezmíněnou třídou je třída `Instance`, které reprezentuje instanci aktuálně rozpracovaného, nebo dokončeného, targetu nebo spoileru. Tato instance se mění při detekování vhodné metody, který je provedena v testovaném programu (viz kapitola ... reakce na příchod metody TODO...). Třída `Instance` obsahuje vektorový čas první a poslední metody v sekvenci metod, odkaz na šablonu této sekvence metod (tj. target nebo spoiler) a jména parametrů ze sekvence metod, společně s již přiřazenými hodnotami. Dále instance obsahuje atribut `actualPos`, tj. index metody ze šablony, která byla jako poslední přijata<sup>2</sup>. Vztah mezi jmény parametrů a jejich hodnotami bude vysvětlen v následujících kapitolách, ovšem zde je důležité si uvědomit, že jména parametrů se nachází v šabloně metod (`MethodTemplate`), zatímco jejich hodnoty až v konkrétních instancích.

Počet šablon (tj. targetů a spoilery) bude neměnný, ovšem počet jejich instancí bude v průběhu analýzy narůstat. Uložení těchto instancí je tedy navrženo tak, aby se mezi nimi dalo co nejrychleji vyhledávat, tj. instance, které spolu souvisejí, jsou uloženy vždy pohromadě a jsou odděleny od

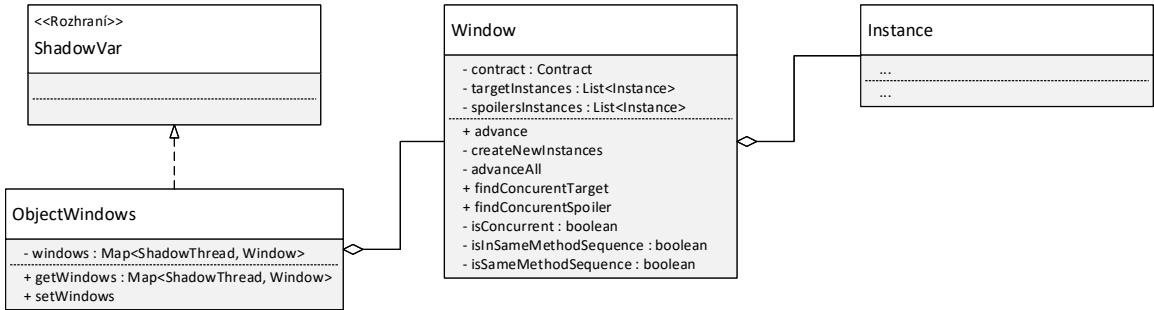
<sup>1</sup>Jména parametrů nejsou skutečnými jmény parametrů, nýbrž jmény zástupnými, které slouží pro vyjádření vztahů, mezi více metodami. Více v kapitole ...reakce na příchod exit metody TODO ...

<sup>2</sup>Atribut `actualPos` ukazuje na poslední pozorovanou událost ze sekvence metod šablony. Tj. pokud je instance vytvořena, je hodnota `actualPos = 0` (ukazuje na první metodu), a pokud je instance dokončena, pak je hodnota `actualPos = length - 1`, kde `length` je počet metod v sekvenci dané šablony.



Obrázek 4.2: Diagram zobrazující třídy Target, Spoiler, Instance a jim přidružené.

ostatních, které s nimi nesouvisí. K tomuto účelu je využita třída *ShadowVar*, která umožňuje ukládat informace ke každému paměťovému místu (tj. ke každému objektu nebo proměnné). Konkrétně je použita instance třídy *ObjectWindows*<sup>3</sup>, která obsahuje přiřazení objektů *Window* ke každému existujícímu vláknu. Instance této třídy je vytvořena při prvním přístupu k paměťovému místu, což je zajištěno v metodě *makeShadowVar*. Třída *Window* je již třídou obsahující jednotlivé instance targetů a spoilerů. Tento vztah je reprezentován na diagramu 4.3, kde jsou pro jednoduchost vynechány položky třídy *Instance*, která je plně zobrazena na diagramu 4.2.



Obrázek 4.3: Diagram zobrazující třídu *ObjectWindows* a jí přidružené trídy.

### 4.3.2 Práce s šablonami a instancemi

V dynamickém analyzátoru se tedy vyskytují šablony targetů, spoilerů a jejich instance. V této kapitole bude popsáno, jak spolu souvisí a jak je s nimi pracováno.

Při inicializaci samotného nástroje jsou vytvořeny všechny targety a spoilery z konfiguračního souboru. Následně je zahájena samotná dynamická analýza, která sleduje důležité události. V kontextu práce s šablonami a instancemi je důležitá pouze událost *exit*, která informuje o ukončení vykonání metody *m<sub>event</sub>*. Tato událost obsahuje informace o objektu, na kterém byla vykonána, identifikaci (tj. jméno metody), hodnoty parametrů, se kterými byla zavolána, a návratovou hodnotu této metody. Důvodem, proč je nutné použít metodu *exit*, namísto metody *enter*, je ten, že při zavolání metody *enter* jsou k dispozici pouze hodnoty vstupních parametrů, ale nikoli návratová hodnota. Oproti tomu v metodě *exit* tato návratová hodnota již známá je, a proto se při dynamické analýze kontraktů s parametry zachytávají vykonalé metody až po jejich provedení - tedy v metodě *exit*. Jakmile dojde k detekci této události, pak dynamický analyzátor provede následujících kroků:

1. V případě, že metoda *m<sub>event</sub>* odpovídá první metodě *m<sub>first</sub>* některého targetu, nebo spoileru, pak vytvoří jeho novou instanci.
2. V případě, že metoda *m<sub>event</sub>* odpovídá metodě *m<sub>expect</sub>*, která je právě očekávána v některé instanci targetu nebo spoileru, pak provede krok *advance* nad touto instancí.
3. Pokud nedojde ani k jednomu z předchozích případů, pak je událost ignorována.

### 4.3.3 Vytvoření nové instance

Pokud dojde k 1. kroku z předchozí kapitoly 4.3.2, je vytvořena nová instance targetu nebo spoileru. V tomto kroku je také zmíněno, že metoda *m<sub>event</sub>* musí odpovídat první metodě *m<sub>first</sub>* některého targetu nebo spoileru. V tomto případě tedy musí dojít k porovnání těchto metod, přičemž při vytváření

<sup>3</sup>Třída *ObjectWindows* dědí od třídy *ShadowVar*.

nové instance ještě nejsou dostupné žádné informace o hodnotách parametrů, a tak je dostačující pouze následující porovnání:

1. Metoda  $m_{event}$  musí mít stejné jméno<sup>4</sup> jako metoda  $m_{first}$ .
2. Metoda  $m_{event}$  musí mít stejný počet parametrů jako  $m_{first}$ .

Na první pohled by se mohlo zdát, že nedochází ke kontrole typů parametrů, a také výstupního parametru. Tato kontrola ovšem není možná, neboť v definici kontraktů se nacházejí pouze zástupná jména parametrů, ale nikoli jejich typy.

Při vytvoření nové instance targetu nebo spoileru je utně provést následující kroky:

- Přiřadit instanci odkaz na šablonu (target nebo spoiler), ze které je vytvářena.
- Aktuální index v sekvenci metod (*actualPos*) nastavit na hodnotu 0.
- Přiřadit instanci identifikátor vlákna, ve kterém byla metoda  $m_{event}$  vyvolána.
- Přiřadit instance vektorový čas metody  $m_{event}$ , který bude značit začátek této instance.
- Vytvořit mapu parametrů *parameters* ze sekvence metod šablony.
- Přiřadit hodnoty parametrů z metody  $m_{event}$  k odpovídajícím klíčům mapy *parameters*.

V kapitole 4.3.1 je zmíněno, že zástupné jména parametrů jsou uloženy v šablonách, ale jejich hodnoty v instancích. A právě poslední krok v předcházejícím výčtu se o toto uložení stará. Uložení je tedy realizováno tak, že v každé instanci existuje mapa *parameters*, kde klíče jsou zástupné jména parametrů, ze sekvence metod související šablony, a hodnoty jsou při inicializaci nastaveny na hodnotu *Undefined*. Jakmile je detekována metoda  $m_{event}$ , pak jsou hodnoty jejích parametrů přiřazeny k patřičným klíčům v mapě *parameters*. V kontextu vytváření instance, budou hodnoty parametrů z metody  $m_{event}$  vždy přiřaditelné. Ovšem pokud bude přijata jiná metoda než první, nemusejí být parametry přiřaditelné, a proto je nutné kontrolovat také jejich hodnoty (více v kapitole 4.3.4). Pro lepší pochopení práce s mapou *parameters* je její inicializace vysvětlena na příkladu 4.1.

**Příklad 4.1.** Necht' existuje target se sekvencí metod:  $m_1(X, Y) \quad m_2(Y, Z)$ . Necht' je právě detekována událost *exit* s metodou  $m_1(5, 7)$ . Vzhledem k tomu, že je metoda  $m_1$  první metodou v sekvenci, bude vytvořena nová instance tohoto targetu podle výše uvedeného postupu. Při vytváření instance bude vytvořena mapa *parameters*, která bude mít následující obsah:

```
parameters = {
    "X" = Undefined,
    "Y" = Undefined,
    "Z" = Undefined
}
```

Následně budou do této mapy dosazeny hodnoty parametrů. V šabloně targetu jsou u metody  $m_1$  uvedeny dva parametry se zástupnými jmény  $X$  a  $Y$ . Takže první hodnota (tj. 5) z detekované metody se přiřadí ke klíči  $X$  a druhá hodnota (tj. 7) ke klíči  $Y$ . Výsledkem tedy bude mapa *parameters* s následujícím obsahem:

```
parameters = {
    "X" = 5,
    "Y" = 7,
    "Z" = Undefined
}
```

---

<sup>4</sup>K porovnávání dochází nad *fully qualified* jménem metod

#### 4.3.4 Krok advance

V předchozí kapitole bylo popsáno, jak probíhá vytvoření nové instance targetu nebo spoileru. V této kapitole bude, oproti tomu, ukázáno jako se instance v průběhu dynamické analýzy vyvíjí. Tento případ byl uveden jako 2. krok v kapitole 4.3.2.

Pro zjištění, zda je splněna podmínka z kroku č. 2 z kapitoly 4.3.2, je opět nutné porovnat metody  $m_{event}$  a  $m_{expect}$ , přičemž toto porovnání navíc obsahuje porovnání hodnot parametrů. Všechny parametry metody  $m_{event}$  musejí být *shodné* nebo *dosaditelné* do parametrů metody  $m_{expect}$ , počátkem dané instance. V předchozí kapitole 4.3.3 bylo vysvětleno jak jsou hodnoty parametrů v instanci uloženy a bylo zde popsáno, že všechny parametry mají po vytvoření hodnotu *Undefined*. Pokud tedy metoda  $m_{event}$  byla volána s určitými hodnotami parametrů, pak platí, že pro všechny její parametry  $p$  mohou nastat pouze tyto případy:

1. V mapě *parameters* nemá parametr  $p$  ještě přiřazenu hodnotu.
2. Parametr  $p$  má být ignorován.
3. V mapě *parameters* má parametr  $p$  již přiřazenu hodnotu a tato hodnota je totožná s hodnotou parametru  $p$ .
4. V mapě *parameters* má parametr  $p$  již přiřazenu hodnotu a tato hodnota není totožná s hodnotou parametru  $p$ .

Pokud u všech parametrů metody  $m_{event}$  nastanou situace z bodu 1., 2. nebo 3., pak jsou parametry považovány za *shodné*. Pokud jsou navíc splněny obě podmínky z kapitoly 4.3.3, pak je proveden krok *advance* nad danou instancí targetu nebo spoileru.

Krok *advance* umožňuje instanci  $i$  posunout se vpřed k jejímu dokončení. Pokud je metoda  $m_{event}$  shodná s metodou  $m_{expect}$  (tj. jsou splněny výše uvedené podmínky), pak je nad danou instancí  $i$  proveden jeden z následujících kroků:

- Pokud  $m_{expect}$  je volána s parametry, z nichž všechny již mají přiřazenu totožnou hodnotu v mapě *parameters*, pak dojde k inkrementaci hodnoty atributu *actualPos* o hodnotu 1.
- Pokud  $m_{expect}$  je volána s parametry, z nichž alespoň jeden nemá přiřazenu hodnotu v mapě *parameters*, pak dojde k vytvoření kopie  $i_{new}$  dané instance  $i$ . Této nové instanci  $i_{new}$  je přiřazen odkaz na rodičovskou instanci  $i$ , inkrementována hodnota atributu *actualPos* o hodnotu 1 a aktualizována mapa *parameters* o nově získané hodnoty parametrů.

Důvodem, proč se ve druhém případě vytváří nová instance, je právě dosazení parametrů. Tato situace je vysvětlena na následujícím příkladu 4.2.

**Příklad 4.2.** Nechť je zadána definice kontraktu následovně:

```
m1(X) m2(Y) <- { m3(X, Y) }
```

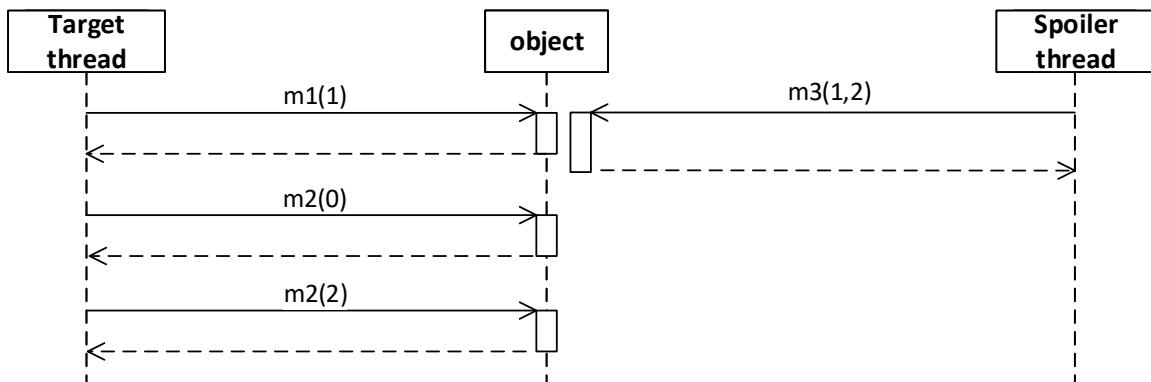
Pak se dynamický analyzátor bude snažit nalézt target se sekvencí metod  $m1$ ,  $m2$ , které jsou volány s parametry  $X$  a  $Y$ . Zároveň se bude snažit nalézt spoiler, který obsahuje pouze samotnou metodu  $m3$ , která je ale volána s totožnými hodnotami  $X$  a  $Y$ . Na diagramu 4.4 je znázorněna situace, kdy je volána metoda  $m2$  dvakrát, pokaždé s jiným parametrem, a až druhé volání této metody vede k vytvoření targetu, který může být porušen zadáným spoilerem.

Pro popis situace je zavedeno následující označení:  $m1(\emptyset)$  značí vykonání metody  $m1$ , s parametrem  $\emptyset$ , Instance{T; MS:  $m1$ ; PAR: X=∅, Y=Undefined} značí instanci targetu (T) se sekvenčí metod (MS) a parametry (PAR).

V tuto chvíli bude popsáno pouze vytváření instancí targetu, nikoli spoileru, protože instance spoileru bude jediná: `Instance{S; MS: m3; PAR: X=1, Y=2}`. Instance targetu tedy budou vytvářeny/upravovány na základě provedených metod následovně:

1. `m1(1)`  
`Instance{T; MS: m1; PAR: X=1, Y=Undefined}`
2. `m2(0)`  
`Instance{T; MS: m1; PAR: X=1, Y=Undefined},`  
`Instance{T; MS: m1, m2; PAR: X=1, Y=0}`
3. `m2(2)`  
`Instance{T; MS: m1; PAR: X=1, Y=Undefined},`  
`Instance{T; MS: m1, m2; PAR: X=1, Y=0},`  
`Instance{T; MS: m1, m2; PAR: X=1, Y=2}`

V 1. kroku je vytvořena nová instance targetu. Ve 2. kroku je viditelné, že nedojde pouze ke kroku *advance* nad existující instancí, ale je nejprve vytvořena kopie této instance, a až poté je vykonán krok *advance* nad touto kopií. Tento krok je důležitý, protože nově vytvořená instance neodpovídá instanci spoileru, neboť má dosazeny jiné hodnoty parametrů, než se vyskytují ve spoileru. Když se v kroku 3. poté objeví vykonání metody `m2` s jiným (tj. správným) parametrem, dojde opět k vytvoření kopie instance z kroku 1 a k provedení kroku *advance* nad touto kopií. Tato nově vytvořená instance již odpovídá spoileru. Pokud by tedy nedošlo k vytvoření kopie instance v kroku 2, a pouze by se provedl krok *advance*, pak by nebylo možné vytvořit správnou instanci ve 3 kroku.



Obrázek 4.4: Příklad demonstруjící vytváření a upravování instancí.

Pokud došlo ke kroku *advance*, je nutné zkontrolovat, zda instance není ukončena. Tento případ nastane, pokud hodnota atributu *actualPos* je o 1 menší, než počet metod v sekvenci šablony. Je-li instance dokončena, je nejprve přiřazen vektorový čas události *m<sub>event</sub>*, který značí čas ukončení instance. Následně jsou hledány související dokončené protějšky<sup>5</sup> a konečně, dojde-li k případu, kdy je nalezen alespoň jeden protějšek, pak je na základě vektorových hodin zkontrolováno, zda jsou tyto dvě instance konkurentní. Pokud platí i poslední podmínka, došlo k detekci porušení kontraktu.

<sup>5</sup>Protějškem je myšlen druhý z dvojice target-spoiler. Tj. Pokud dojde k ukončení instance targetu, pak jsou hledány související ukončené spoilery; pokud dojde k ukončení instance spoileru, pak jsou hledány související ukončené targety.

### 4.3.5 Vektorový čas

V několika posledních kapitolách byla několikrát zmíněna práce s vektorovými hodinami, bez většího vysvětlení. Tato kapitola tedy popisuje, jakým způsobem je navržena práce s těmito hodinami. Vektorové hodiny jsou důležité pro určení zda jsou dvě instance targetu a spoileru konkurentní, tj. zda se mohou vyskytnout v takovém proložení, aby došlo k jejich porušení (více viz 3.2).

Princip práce s vektorovým časem, který je použitý v dynamickém analyzátoru kontraktů, vychází z DJIT+ algoritmu [?]. Vektorový čas je udržován v každém vláknu a objektu, který je použitý jako zámek. V nástroji Roadrunner jsou pro jeho použití využity následující metody:

**Create** V této metodě dochází k vytvoření instance vektorových hodin pro nové vlákno.

**PreStart** V této metodě dochází k inkrementaci vektorových hodin obou vláken, tj. rodičovského i nově vytvořeného vlákna.

**Acquire** Zde je aktualizován vektorový čas vlákna pomocí vektorového času zámku<sup>6</sup>.

**Release** Zde je aktualizován vektorový čas zámku pomocí vektorového času vlákna. Dále je inkrementován čas vlákna v jeho vektorových hodinách.

**PostJoin** V této metodě dochází k aktualizaci hodin rodičovského vlákna pomocí vektorového času ukončovaného vlákna. Dále proběhne inkrementace vektorových hodin rodičovského vlákna.

### 4.3.6 Informace o detekci porušení

V předchozích kapitolách bylo popsáno, jakým způsobem dojde k detekci porušení. V této kapitole jsou pouze definovány důležité parametry, které je potřeba sdělit na výstupu. Jako důležité informace jsou tedy vybrány:

- Metoda, po které došlo k detekci porušení.
- Vlákno, ve kterém byla metoda vyvolána.
- Stacktrace vlákna, ve kterém byla metoda vyvolána.
- Instance targetu, který byl porušen.
- Instance spoileru, které jej porušil.

## 4.4 Podpora nástroje v SearchBestie

V kapitole 4.3 byl popsán návrh nástroje *ContractTool* ve frameworku Roadrunner. Tato kapitola popisuje návrh částí systému, díky kterým je možné tento nástroj používat z platformy SearchBestie. První částí je vytvoření tzv. *cover produceru* pro analyzátor kontraktů. Další částí je úprava nástroje RRPlugin a poslední částí je vytvoření tzv. *result item creatoru*<sup>7</sup>. Kromě vytvoření těchto částí je třeba navrhnout reprezentaci parametrů pro analyzátor kontraktů, které budou předány v konfiguračním souboru pro platformu SearchBestie.

<sup>6</sup>Detailnější popis aktualizací vektorových časů je popsán ve zmíněném článku [?].

<sup>7</sup>Komunikace mezi těmito částmi byla popsána v kapitole 4.2

#### 4.4.1 Cover producer a úprava nástroje RRPlugin

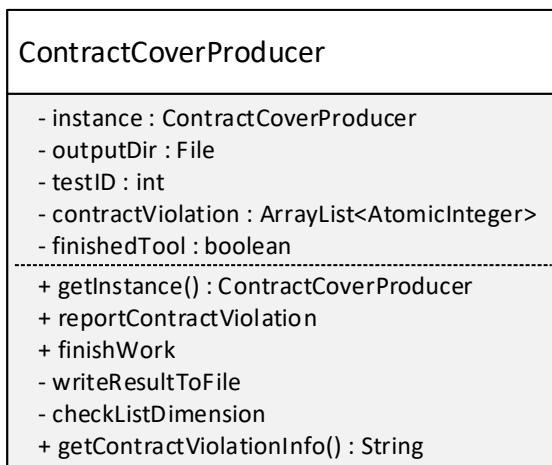
*ContractCoverProducer* (CCP) má za úkol sbírat informace o porušení kontraktů a po skončení testování tyto informace uložit do výstupního souboru. Tato třída je stejně jako *ContractTool* součástí projektu Roadrunner a chyby jsou reportovány přímo z tohoto nástroje. K reportování chyb ovšem dochází pouze v případě, že je analyzátor spuštěn z prostředí SearchBestie, tj. kromě nástroje *ContractTool* je v řetězci nástrojů také RRPlugin, který vytváří instanci této třídy. Pokud RRPlugin není v řetězci nástrojů při spuštění RoadRunneru, nedojde k vytvoření instance CCP a *ContractTool* vypisuje nalezené chyby pouze na výstup.

Struktura CCP je znázorněna na diagramu 4.6. Důležité jsou především metody *reportContractViolation* a *writeResultToFile*. První zmíněná metoda slouží pro zaznamenání porušení kontraktu a druhá metoda slouží pro vytvoření výstupního souboru a zapsání výsledku. CCP si zaznamenává počet porušení jednotlivých targetů, které jsou identifikované pomocí přiděleného identifikátoru. Obsah výstupního souboru je navržen jako posloupnost čísel oddělených dvojtečkou:

$$x_0 : x_1 : \dots : x_{n-1}$$

kde  $x_i$  značí počet nalezených porušení targetu s identifikátorem  $i$  a  $n$  značí počet všech targetů v kontraktu. Takto formátovaný výstupní soubor je později zpracován třídou *ContractRICreator*, která je popsána v kapitole 4.4.2.

Výše popsaný CCP je vytvářen v nástroji RRPlugin tak, jako ostatní cover producery. Z tohoto důvodu musí být CCP také korektně ukončen v momentě, kdy dochází k ukončení nástroje RRPlugin. Proces vytvoření, běhu a ukončení CCP je znázorněn na diagramu 4.1.



Obrázek 4.5: Diagram zobrazující třídu *ContractCoverProducer*.

#### 4.4.2 ContractRICreator a úprava SearchBestie

Pro podporu analyzátoru kontraktů v platformě SearchBestie je nejprve třeba definovat nové parametry do vstupního XML konfiguračního souboru. Těmito novými parametry jsou:

- <parameter key="contractFile"/> – Parametr definující cestu ke konfiguračnímu souboru kontraktu.
- <parameter key="CTcontract"/> – Parametr povolující vytvoření cover producera *ContractCoverProducer*.

Tyto parametry jsou vnořeny do elementu `parameters`, díky čemuž jsou automaticky parsovaný. Parametry, jejichž klíč začínající na prefix `CT`, jsou považovány za parametry povolující cover producery a ostatní parametry jsou předávány Roadrunneru. Proto musí být klíč shodný jako parametr definovaný v kapitole 4.2. Pro parametr `CTcontract` je ještě nutné přidat hodnotu výčtového typu (`CONTRACT(...)`) do třídy `ConcurrencyCoverage`<sup>8</sup>. Díky tomuto kroku je možné upravit fitness funkci, která se používá pro ohodnocení jednoho běhu například takto:

```
<fitness class="cz.vutbr.fit.sbestie.search.fitness.FitnessExpression"
    name="MyFitness">
    <parameters>
        <parameter key="expression" value="CONTRACT" />
    </parameters>
</fitness>
```

Takto definovaná fitness funkce se bude snažit maximalizovat počet nalezených porušení kontraktů porušení.

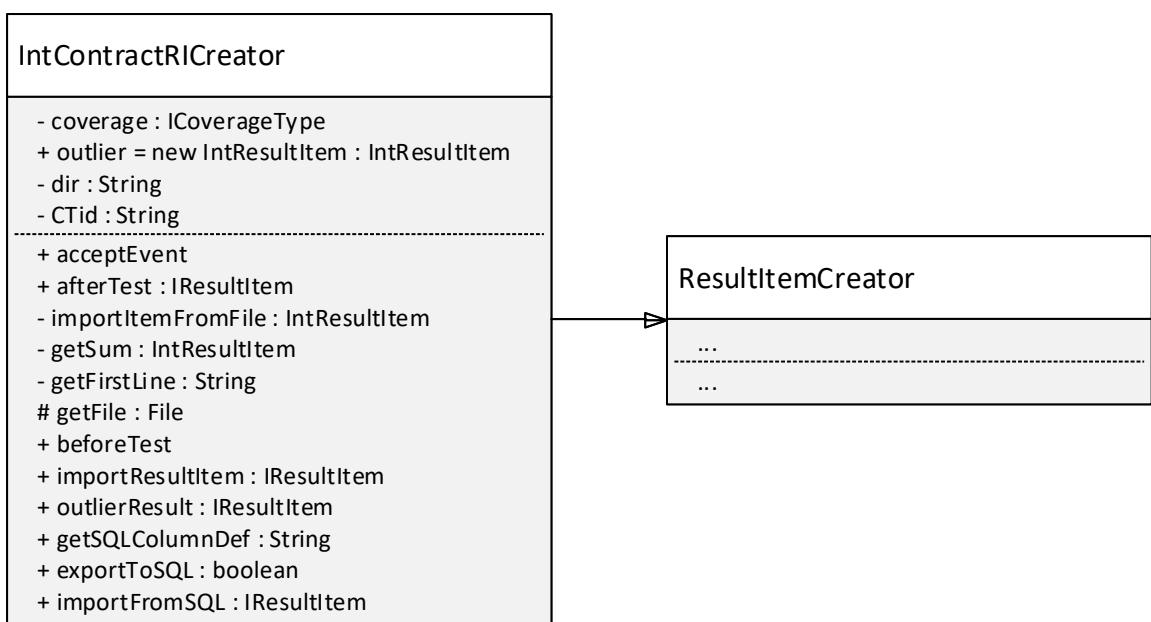
Posledním krokem je vytvoření `ContractResultItemCreatoru`. Tato třída slouží ke zpracování výsledků z RoadRunneru (konkrétně z `ContractCoverProduceru`) a vytvoření výsledku, se kterým dokáže pracovat `SearchBestie`. Jako tento výsledek byla použita třída `IntResultItem`, která reprezentuje výsledek jednoho testu pouze číselnou hodnotou. `ContractResultItemCreator` tedy zpracuje sekvenci hodnot reprezentující počty nalezených porušení jednotlivých targetů, tyto výsledky seče a vytvoří instanci třídy `IntResultItem`, do které uloží tuto sumu. Tento výsledek bude tedy reprezentovat počet všech nalezených porušení kontraktu, čehož je využito ve fitness funkci definované v předchozí kapitole.

Na diagramu je znázorněna struktura třídy `ContractResultItemCreator`, která rozšiřuje třídu třídu `ResultItemCreator`, a je tedy nutné, zařadit výše popsanou funkcionality do správných metod této nadtířidy. Důležité jsou zejména metody `acceptEvent` a `afterTest`. První jmenovaná metoda slouží pro uložení názvu podsložky<sup>9</sup> a identifikátoru testu, který byl spuštěn. Druhá metoda je zavolána po skončení testu a je využita pro zpracování výstupního souboru z RoadRunneru. Výstupem této metody je popsaný číselný výsledek testu, reprezentovaný instancí třídy `IntResultItem`.

---

<sup>8</sup>Tato hodnota definuje prefixy pro výstupní soubory, názvy složek atd.

<sup>9</sup>V případě hledání porušení kontraktů je podsložka vždy `contract/`. Název této složky je definován ve výčtu `ConcurrencyCoverage` zmíněném výše.



Obrázek 4.6: Diagram zobrazující třídu ContractResultItemCreator.

# Kapitola 5

## Implementace

Tato kapitole popisuje implementační detaily a řešení problémů, které při implementaci vznikly. Implementace přímo vychází z předchozí kapitoly 4.

Porovnávání parametrů ??

### 5.1 Implementační problémy

Během implementace se vyskytlo několik problémů, které musely být nutně vyřešeny, jinak by nebylo možné tuto práci dokončit. Tato kapitola tedy popisuje tyto problémy a jejich řešení.

Při návrhu dynamického analyzátoru ve frameworku RoadRunner bylo předpokládáno, že v metodách *enter* a *exit* jsou k dispozici hodnoty parametrů, se kterými byly metody zavolány, počátkem návratová hodnota metody. V průběhu implementace bylo ale zjištěno, že objekty, které jsou předány do zmíněných metod, obsahují pouze statické informace o těchto parametrech, tj. počet parametrů a jejich datové typy. Bylo tedy nutné upravit RoadRunner tak, aby hodnoty parametrů byly v metodách *enter* a *exit* k dispozici. Úpravy, které byly provedeny jsou posány v následujících kapitolách ?? a ??.

#### 5.1.1 Rozšíření RoadRunneru o parametry metod

RoadRunner reprezentuje událost vstupu (i výstupu) objektem třídy *MethodEvent*. Instance této třídy je předána jako parametr, jak do metody *enter*, tak i do metody *exit* třídy *Tool*. Je tedy nutné rozšířit třídu *MethodEvent* o atribut, který obsahuje hodnoty parametrů. Tento parametr byl nazván *params* a je typu *Object[]*. Důvod, proč byl zvolen tento typ je, že v Javě se mohou vyskytovat jak primitivní, tak referenční datové typy. Všechny referenční datové typy rozšiřují třídu *Object*, takže mohou být v tomto poli uloženy. Primitivní datové typy třídu *object* nerozšiřují, nicméně, ke každému primitivnímu datovému typu existuje referenční datový typ. Převod primitivního datového typu na referenční se provádí pomocí metody *valueOf* daného referenčního typu. Pokud tedy mají být uloženy všechny datové typy do jednoho pole, je nutné všechny typy převést na referenční a tyto referenční datové typy uložit do tohoto pole. Výsledkem tedy je pole, obsahující jak původní referenční datové typy, tak původně primitivní datové typy.

Objekty třídy *MethodEvent* jsou generovány ve třídě *RREventGenerator*, konkrétně v metodě *enter*. Do této metody je nutné přidat další parametr *Object[] params*, ve kterém budou zaslány hodnoty parametrů. Uvnitř této metody probíhá vytvoření instance třídy *MethodEnter*, které je třeba přiřadit hodnoty parametrů z parametru *params*. Tato metoda je vyvolávána staticky dle její deklarace ve třídě *Constant*. Tento záznam je tedy třeba modifikovat a přidat parametr *params*. Vyvořená

instance třídy *MethodEvent* je předána prvnímu nástroji v řetězci nástrojů v metodě *enter*. Zjednodušeně bude tedy upravená metoda *enter* třídy *RREventGenerator* vypadat následovně:

```
public static void enter(final Object target, final int methodDataId, final
    ShadowThread td, Object[] params) {
    ...
    final MethodEvent me = td.enter(target, methodData);
    me.setParams(params);
    ...
    firstTool.enter(me);
}
```

Dalším krokem je získání hodnot parametrů, se kterými byla zavolána metoda v testovaném programu. K tomuto kroku je nutné nejprve uvést, jakým způsobem Roadrunner získává informace o testovaném programu. RoadRunner využívá knihovnu ASM<sup>1</sup>, která slouží pro manipulaci a analýzu Java bytekódu. Konkrétně se analýza a manipulace s metodami provádí ve třídě SyncAndMethodThunkInserter v metodě *createMethodThunk*, kde je dostupný kontext vláken testovaného programu (tj. zásobník, registry, atd.) prostřednictvím objektu *mv* typu *MethodVisitor*<sup>2</sup>. Odtud je dále volána metoda *enter* třídy *RREventGenerator*, přičemž k jejímu volání dochází ihned po vstupu do metody v testovaném programu. Metoda *createMethodThunk* třídy *SyncAndMethodThunkInserter* byla původně volána následujícím způsobem:

```
1 private void createMethodThunk(int access, String name, String desc,
    String signature, String[] exceptions, String wrappedMethodName, int
    maxLocals) {
2     ...
3     MethodInfo m = method;
4     mv.push(m.getId());
5     mv.invokeStatic(Constants.THREAD_STATE_TYPE,
                      Constants.CURRENT_THREAD_METHOD);
6     mv.invokeStatic(Constants.MANAGER_TYPE, Constants.ENTER_METHOD);
7     ...
8 }
```

Ve výše uvedeném kódu proběhne na řádku 4 uložení *methodDataId* na zásobník. Na řádku 5 dojde k vyvolání statické metody *getCurrentShadowThread*, definované ve třídě *Constants*, která uloží na vrchol zásobníku objekt *ShadowThread* asociovaný k aktuálnímu vláknu. Jako poslední proběhne na řádku 6 vyvolání statické metody *enter* třídy *RREventGenerator*. Pro správné vyvolání této metody, je nutné mít umístěny všechny její parametry na zásobníku, což ale není splněno, jelikož byl do této metody přidán parametr *Object[] params*. Z tohoto důvodu je mezi 5. a 6. řádek programu vložen kód, který na zásobník umístí pole obsahující hodnoty parametrů metody z testovaného programu. Upravená metoda *createMethodThunk* je zobrazena v 5.2:

Listing 5.1: Upravená metoda *createMethodThunk* zajišťující vytvoření pole hodnot parametrů volané metody.

```
1 private void createMethodThunk(int access, String name, String desc,
    String signature, String[] exceptions, String wrappedMethodName, int
    maxLocals) {
2     ...
3     MethodInfo m = method;
4     mv.push(m.getId());
5     mv.invokeStatic(Constants.THREAD_STATE_TYPE,
                      Constants.CURRENT_THREAD_METHOD);
```

<sup>1</sup>Knihovna ASM je dostupná na adrese <http://asm.ow2.org/>.

<sup>2</sup>MethodVisitor je třída z knihovny ASM, sloužící k analýze a manipulaci s bytekódem souvisejícím s metodami.

```

6
7 //////////////////////////////////////////////////////////////////
8 // ZACATEK: vlozeny kod vytvarejici pole hodnoty parametru
9 //////////////////////////////////////////////////////////////////
10
11 Type[] paramTypes = Type.getArgumentTypes(method.getDescriptor());
12 int paramLength = paramTypes.length;
13
14 if(paramLength > 0) {
15
16     // ulozeni velikosti pole parametru na zasobnik
17     mv.visitIntInsn(Opcodes.BIPUSH, paramLength);
18
19     // vytvoreni pole params o velikosti umistene na vrcholu
20     // zasobniku
21     mv.visitTypeInsn(Opcodes.ANEWARRAY, "java/lang/Object");
22
23     // ulozeni tohoto pole do locals
24     mv.visitVarInsn(Opcodes.ASTORE, paramLength +
25         PARAM_OFFSET);
26
27     Integer i = new Integer(0);
28
29     // cyklus pres vsechny parametry dle jejich typu
30     for (Type type : paramTypes) {
31
32         // ulozeni pole params na vrchol zasobniku
33         mv.visitVarInsn(Opcodes.ALOAD, paramLength +
34             PARAM_OFFSET);
35
36         // ulozeni idenxu na vrchol zasobniku
37         mv.visitIntInsn(Opcodes.BIPUSH, i);
38
39         // zapouzdreni primitivnich typu a zduplikovani
40         // hodnoty i-teho parametru
41         // na vrchol zasobniku
42         i = boxAndVisitVariable(mv, i, type);
43
44         // ulozeni hodnoty parametru do pole params na
45         // index i
46         mv.visitInsn(Opcodes.AASTORE);
47         i++;
48     }
49
50     // ulozeni pole params na vrchol zasobniku
51     mv.visitVarInsn(Opcodes.ALOAD, paramLength +
52         PARAM_OFFSET);
53 } else {
54     // ulozeni hodnoty NULL na vrchol zasobniku
55     mv.visitInsn(Opcodes.ACONST_NULL);
56
57     ...
58 }
59
60 //////////////////////////////////////////////////////////////////
61 // KONEC: vlozeny kod vytvarejici pole hodnoty parametru
62 //////////////////////////////////////////////////////////////////
63
64 mv.invokeStatic(Constants.MANAGER_TYPE, Constants.ENTER_METHOD);
65
66 ...
67
68 }

```

V kódu výše je metoda `boxAndVisitVariable`, která provádí zapouzdření primitivních typů do referenčních datových typů. Hodnoty parametrů metody v testovaném programu jsou umístěny v `locals` a proto tento převod musí probíhat opět na úrovni Java bytekódu. Kód této metody a je znázorněn v příkladu ???. Pokud hodnota parametru na indexu  $i$  je primitivního typu, pak tento kód načte tuto hodnotu z `locals` a uloží ji na vrchol zásobníku. Poté zavolá metodu `valueOf` odpovídajícího datového typu, která vytvoří hodnotu referenčního datového typu z hodnoty na vrcholu zásobníku a výsledek umístí opět na vrchol zásobníku. Pokud již parametr je referenčního datového typu, pak je pouze načten z `locals` na vrchol zásobníku.

Listing 5.2: Metoda zajišťující zapouzdření primitivních datových typů.

```

1  private Integer boxAndVisitVariable(RRMethodAdapter mv, Integer i, Type
2   type) {
3     if (type.equals(Type.BOOLEAN_TYPE)) {
4       mv.visitVarInsn(Opcodes.ILOAD, i + 1);
5       mv.visitMethodInsn(Opcodes.INVOKESTATIC,
6         "java/lang/Boolean", "valueOf",
7         "(Z)Ljava/lang/Boolean;", false);
8     } else if (type.equals(Type.BYTE_TYPE)) {
9       mv.visitVarInsn(Opcodes.ILOAD, i + 1);
10      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
11        "java/lang/Byte", "valueOf",
12        "(B)Ljava/lang/Byte;", false);
13    } else if (type.equals(Type.CHAR_TYPE)) {
14      mv.visitVarInsn(Opcodes.ILOAD, i + 1);
15      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
16        "java/lang/Character", "valueOf",
17        "(C)Ljava/lang/Character;", false);
18    } else if (type.equals(Type.SHORT_TYPE)) {
19      mv.visitVarInsn(Opcodes.ILOAD, i + 1);
20      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
21        "java/lang/Short", "valueOf",
22        "(S)Ljava/lang/Short;", false);
23    } else if (type.equals(Type.INT_TYPE)) {
24      mv.visitVarInsn(Opcodes.ILOAD, i + 1);
25      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
26        "java/lang/Integer", "valueOf",
27        "(I)Ljava/lang/Integer;", false);
28    } else if (type.equals(Type.LONG_TYPE)) {
29      mv.visitVarInsn(Opcodes.LLOAD, i + 1);
30      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
31        "java/lang/Long", "valueOf",
32        "(J)Ljava/lang/Long;", false);
33      i++;
34    } else if (type.equals(Type.FLOAT_TYPE)) {
35      mv.visitVarInsn(Opcodes.FLOAD, i + 1);
36      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
37        "java/lang/Float", "valueOf",
38        "(F)Ljava/lang/Float;", false);
39    } else if (type.equals(Type.DOUBLE_TYPE)) {
40      mv.visitVarInsn(Opcodes.DLOAD, i + 1);
41      mv.visitMethodInsn(Opcodes.INVOKESTATIC,
42        "java/lang/Double", "valueOf",
43        "(D)Ljava/lang/Double;", false);
44      i++;
45    } else {
46      mv.visitVarInsn(Opcodes.ALOAD, i + 1);
47    }

```

```
31         return i;  
32     }
```

Výše uvedeným způsobem je docíleno zduplikování hodnot parametrů volané metody a jejich postupnému předání až do metody *enter* jednotlivých nástrojů. Dálším krokem je obdobným způsobem získat a předat návratovou hodnotu z metod testovaného programu.

### 5.1.2 Rozšíření RoadRunneru o návratový parametr

Rozšíření o návratovou hodnotu metod je provedeno stejným způsobem jako rozšíření a parametry popsaný v předchozí kapitole 5.1.1.

Nejprve je třeba přidat atribut `Object returnValue` do třídy `MethodEvent`. Dalším krokem je přidat stejný parametr do metody *exit* ve třídě `RREventGenerator`, ve které proběhne (stejně jako v metodě *enter* též třídy) vytvoření instance třídy `MethodEvent` a přiřazení parametru `returnValue` této instanci. Tato metoda je stejně jako metoda *enter* volána z metody `createMethodThunk`, kde musí být návratová hodnota také stejným způsobem získána. Vzhledem k tomu, že kód získání návratové hodnoty je téměř stejný jako kód získání ostatních parametrů, není zde uveden, nicméně je dohledatelný ve zdrojových souborech.

Psat to "je to vyreseno"[Poznamky z konzultace] Implementacni detaily - jak je to vyreseno a problemy ktere vznikly. Strukturalni a behavioralni diagramy Co bylo zajimave, co sem jak musel vyresit kdyz nastaly nejake problemy

-implementacni problemy - v cem je to implementovane, ceho se využiva - jak se spousti (priklady)

## 5.2 Implementační detaily

# Kapitola 6

## Testování

### 6.1 Jednotkové testování

### 6.2 Testování dynamického analyzátoru ve frameworku RoadRunner

... TODO ... popis jednoho testu Následující testy jsou popsány jednotným způsobem, a proto bude tato reprezentace nejprve vysvětlena. Každý test se skládá z textového popisu a sekvenčního diagramu. Součástí textového popisu jsou následující položky: *kategorie*, *popis*, *kontrakt*, *očekávaný* a *skutečný výsledek*. *Kategorie* udává na co se daný test zaměřuje, přičemž u jednoho testu může být kategorií více. Kategorie jsou:

*JEDNODUCHÝ KONTRAKT* – Testování probíhá pouze na jednoduchých kontraktech.

*SLOŽITĚJŠÍ KONTRAKT* – Testování probíhá pouze na složitějších kontraktech.

*VEKTOROVÝ ČAS* – K odhalení porušení je nutné použít vektorového času.

*VÍCE OBJEKTU* – Metody targetů a spoilerů jsou volány na více objektů stejných typů.

*VÍCE SPOILERU* – K jednomu targetu je definováno více spoilerů.

*VÍCE VLÁKEN* – Metody targetů a spoilerů jsou volány z více vláken.

*IGNOROVÁNÍ OSTATNÍCH METOD* – V programu se nachází volání metod, které je nutné ignorovat.

*NEDOKONČENÉ SEKVENCE* – V programu se vyskytují nedokončené sekvence metod.

*PARAMETRY* – Kontrakty jsou zadány s parametry.

*RŮZNÉ TYPY PARAMETRU* – V metodách se vyskytují parametry různých typů.

*IGNOROVÁNÍ PARAMETRU* – V kontraktu je zadán parametr *\_*, která je nutné ignorovat.

*VÍCE PARAMETRU* – V metodách se vyskytuje více parametrů.

*NÁVRATOVÝ PARAMETR* – V kontraktu je zadán návratový parametr.

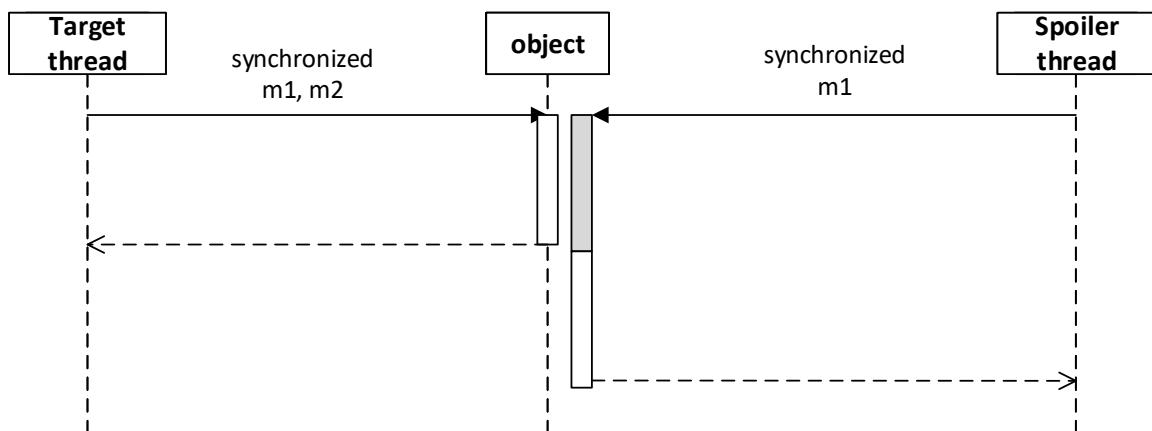
*Popis* uvádí detailnější informace o důvodu porušení (nebo neporušení) konaktu. Položka *kontrakt* uvádí definici konaktu, který je hledán a poslední dvě položky (*očekávaný* a *skutečný výsledek*) uvádějí, kolik porušení konaktu mělo být nalezeno a kolik jich bylo skutečně nalezeno.

Na sekvenčním diagramu je znázorněna testovaná situace. Vyskytuje se zda vlákna targetů (*Target thread n*), vlákna spoilerů (*Spoiler thread n*) a objekty (*Object n*), na které jsou jednotlivé metody volány<sup>1</sup>. Metody objektů jsou volány z vláken a to buď synchronizovaně nebo bez synchronizace. Synchronizovaným voláním<sup>2</sup> je myšleno volání metod zapouzdřené do vhodné synchronizace, tj. uzamknutí zámku před voláním první metody v sekvenci a uvolnění zámku po provedení poslední metody v sekvenci. Takovéto volání metod je znozorněno na digramu ...TODO.... Pokud je navrácena návratová hodnota v takto prováděné sekvenci metod, pak je to znázorněno pomocí popisu hrany. Tj. například v příklad ...TODO... je návratová hodnota metody *m2* hodnota 5, což je zapsáno jako: 5 = *m2()* ...TODO upravit... Volání metody bez použití synchronizace je zobrazena na diagramu ...TODO... Pokud dojde k takovému volání, není zaručeno pořadí volání mezi voláním metod z různých vlákna. Jediná situace, kdy je zaručeno pořadí vykonávání metod napříč různými vlákny je tedy případ, kdy obě vlákna použijí synchronizaci nad stejným objektem (tj. zámkem).

V příkladech se také vyskytuje uspaní vláken na určitou dobu z důvodu nasimulování různého proložení (volání *sleep*).

## Test 1

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Obě vlákna využívají korektní synchronizaci.
- Kontrakt: *m1()* *m2()* <- *m1()*
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- Test proběhl úspěšně.

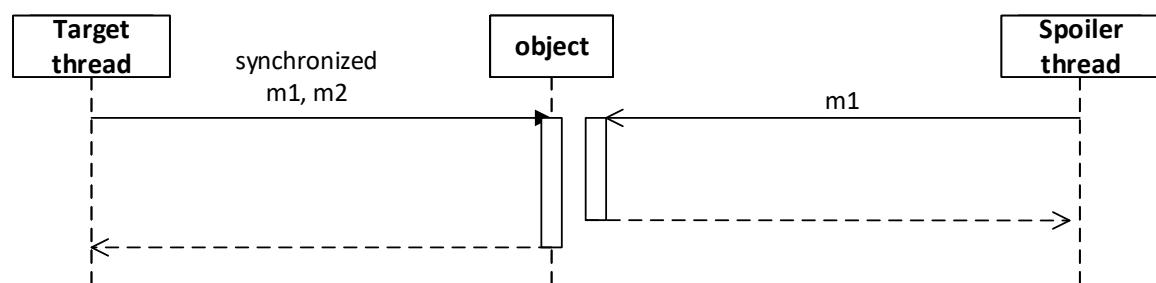


<sup>1</sup>Písmeno n v názvu vláken a objektů značí jejich číslo

<sup>2</sup>Synchronizované a nesynchronizované volání metod je na diagramech rozlišeno pomocí typu šipky a také klíčovým slovem synchronized.

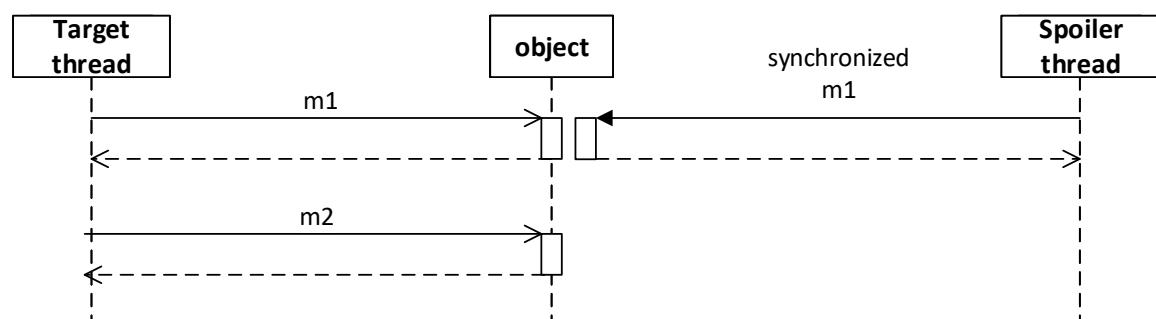
## Test 2

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Vlákno  $S$  volá metodu  $m1$  bez synchronizace.
- Kontrakt:  $m1() \quad m2() \leftarrow \quad m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



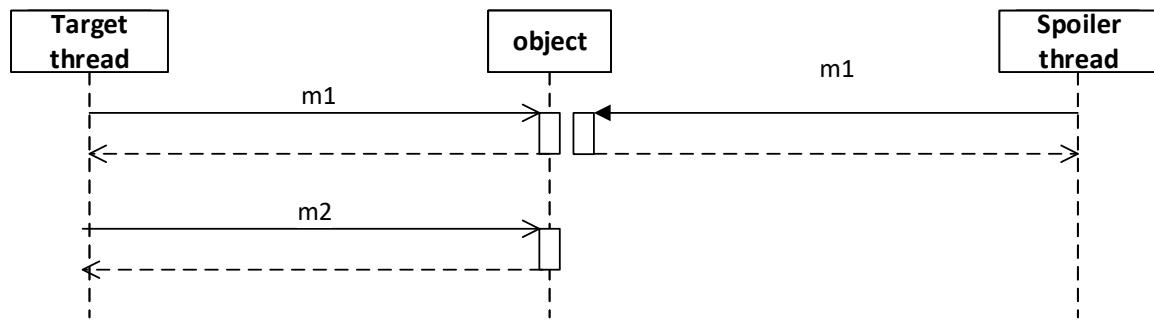
## Test 3

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Vlákno  $T$  volá metody  $m1$  a  $m2$  bez synchronizace.
- Kontrakt:  $m1() \quad m2() \leftarrow \quad m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



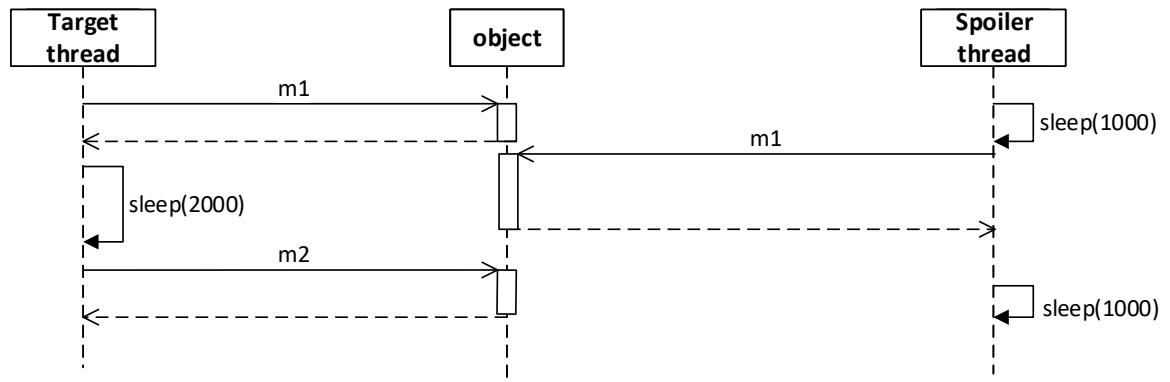
## Test 4

- Kategorie: JEDNODUCHÝ KONTRAKT.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt:  $m1() \quad m2() \leftarrow m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



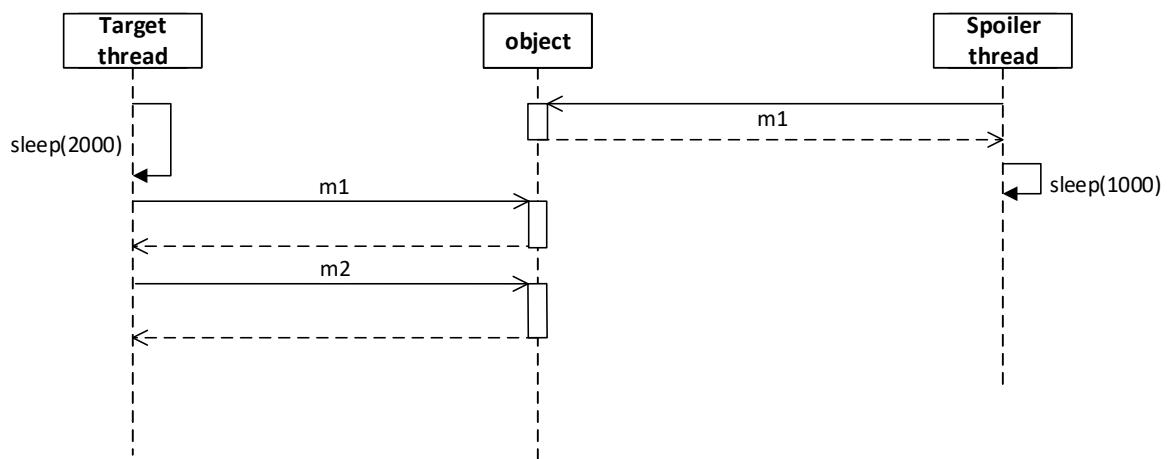
## Test 5

- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je spojován čas vykonání mezi první a poslední metodou targetu. K detekování není nutný vektorový čas.
- Kontrakt:  $m1() \quad m2() \leftarrow m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



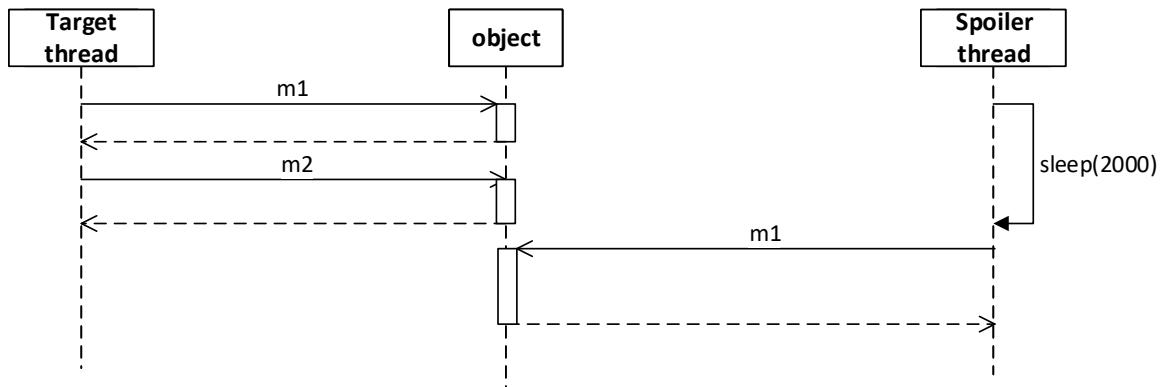
## Test 6

- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je spoiler v čase vykonán dříve, než je vykonána první metoda targetu. K detekování je nutný vektorový čas.
- Kontrakt:  $m1() \ m2() \leftarrow m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



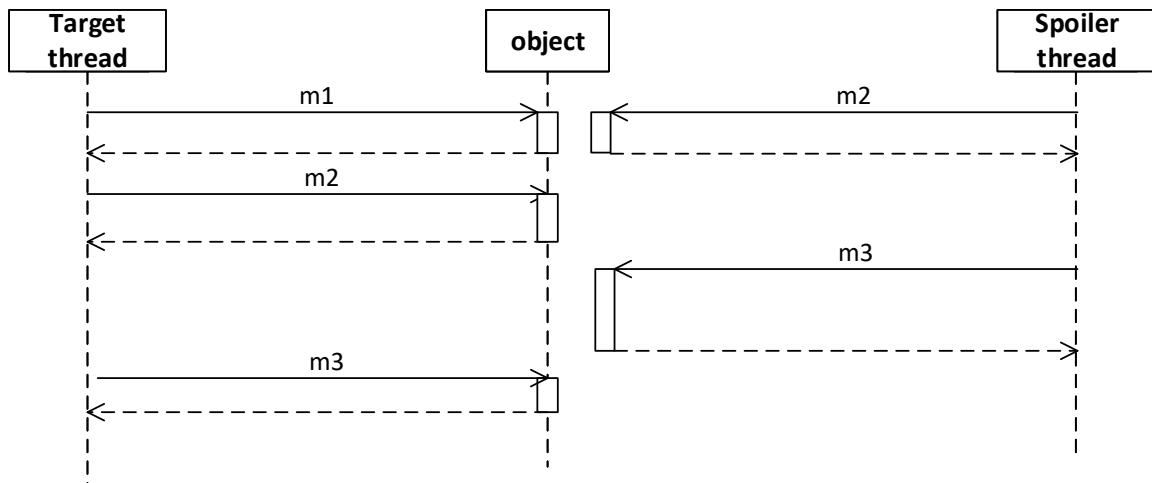
## Test 7

- Kategorie: JEDNODUCHÝ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je celý target v čase vykonán dříve, než první metoda spoileru. K detekování je nutný vektorový čas.
- Kontrakt:  $m1() \ m2() \leftarrow m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



### Test 8

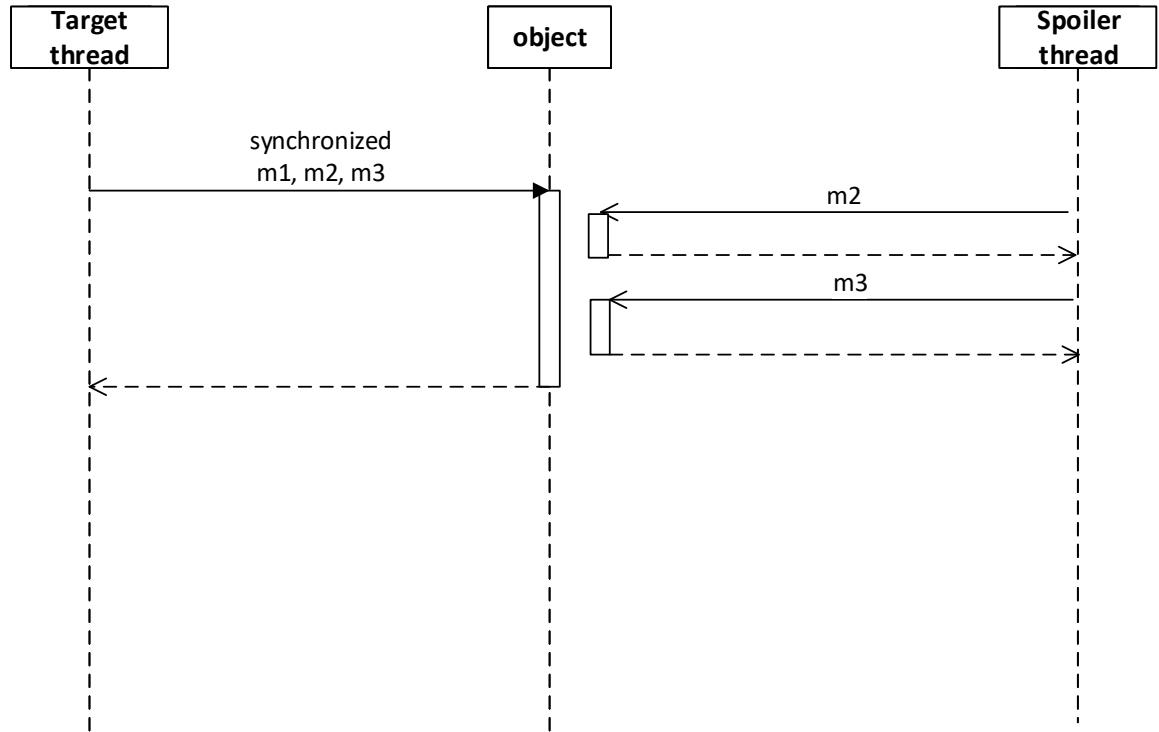
- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



### Test 9

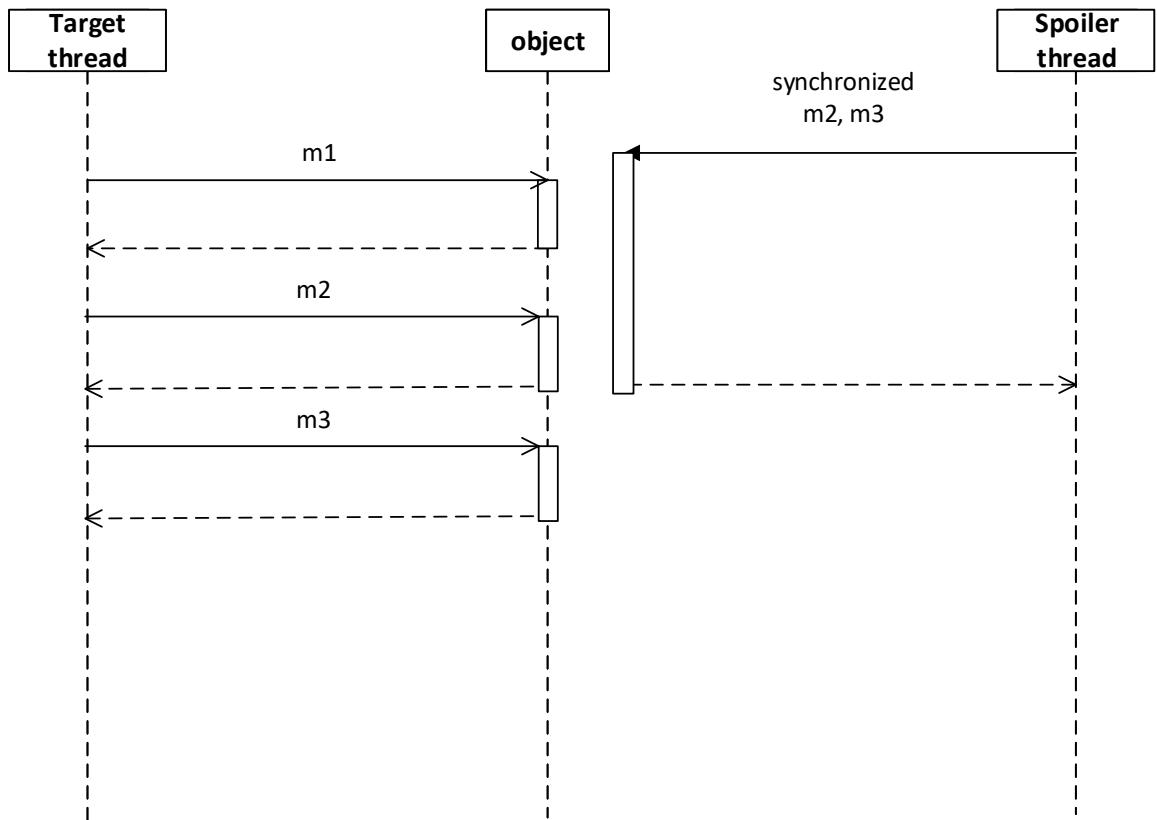
- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno s volá všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 1 nalezených porušení.

- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



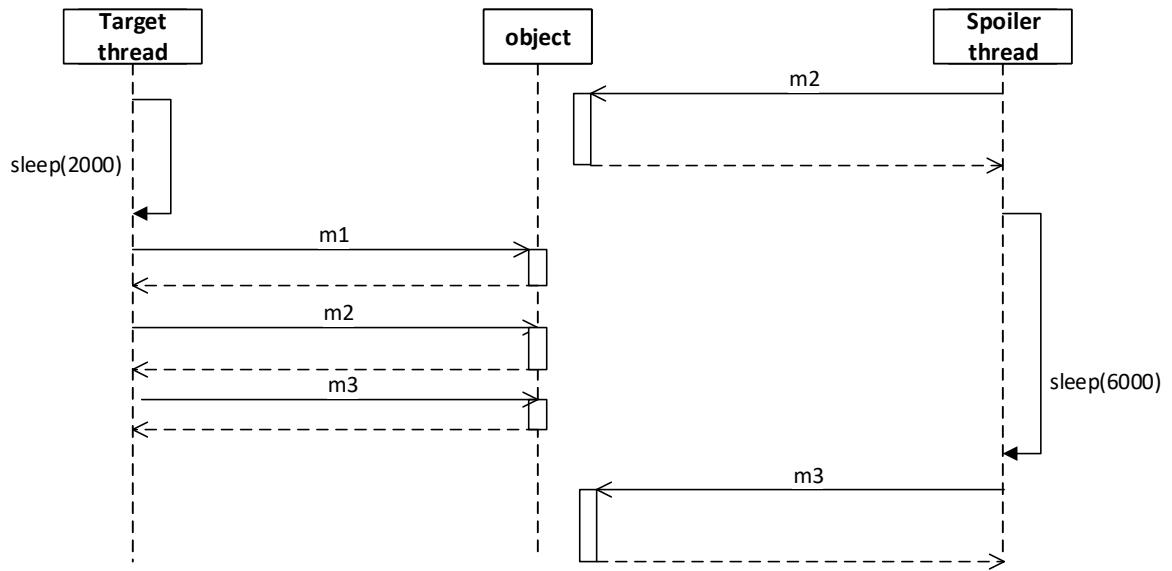
### Test 10

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno  $t$  volá všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow m2() \ m3()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



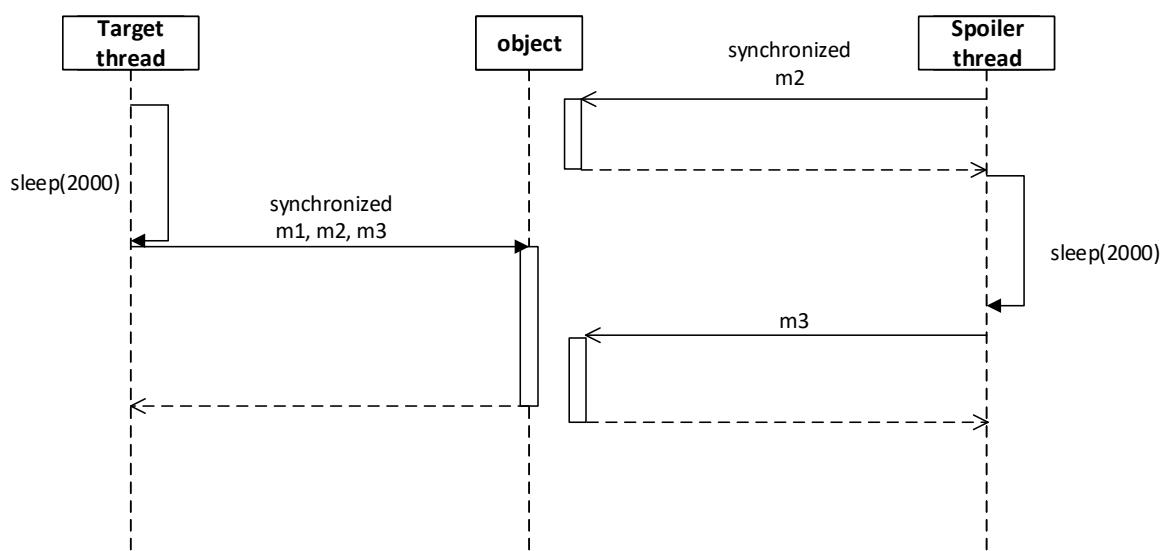
### Test 11

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace. Pomocí uspání vláken, je target v čase vykonán mezi první a poslední metodou spoileru. K detekování je nutný vektorový čas.
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



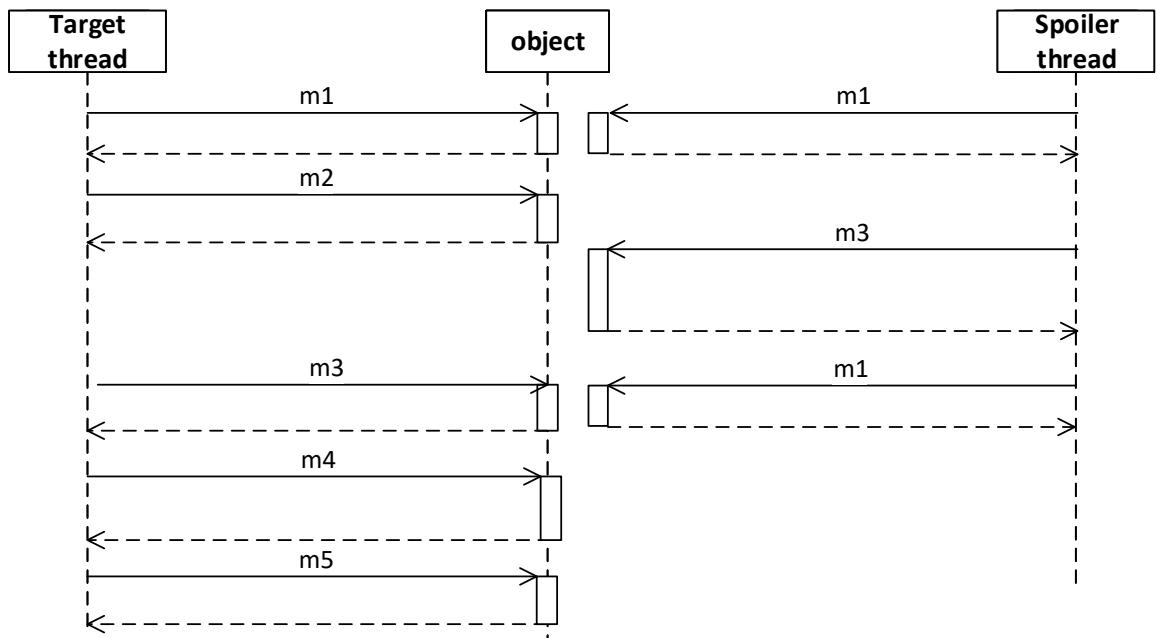
### Test 12

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno  $t$  volá všechny metody za použití správné synchronizace. Metoda  $m2$  vlákna  $s$  volá první metodu za použití synchronizace, ale druhou nikoli. K porušení kontraktu nedochází, neboť spoiler nemůže být celý vložen v targetu.
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



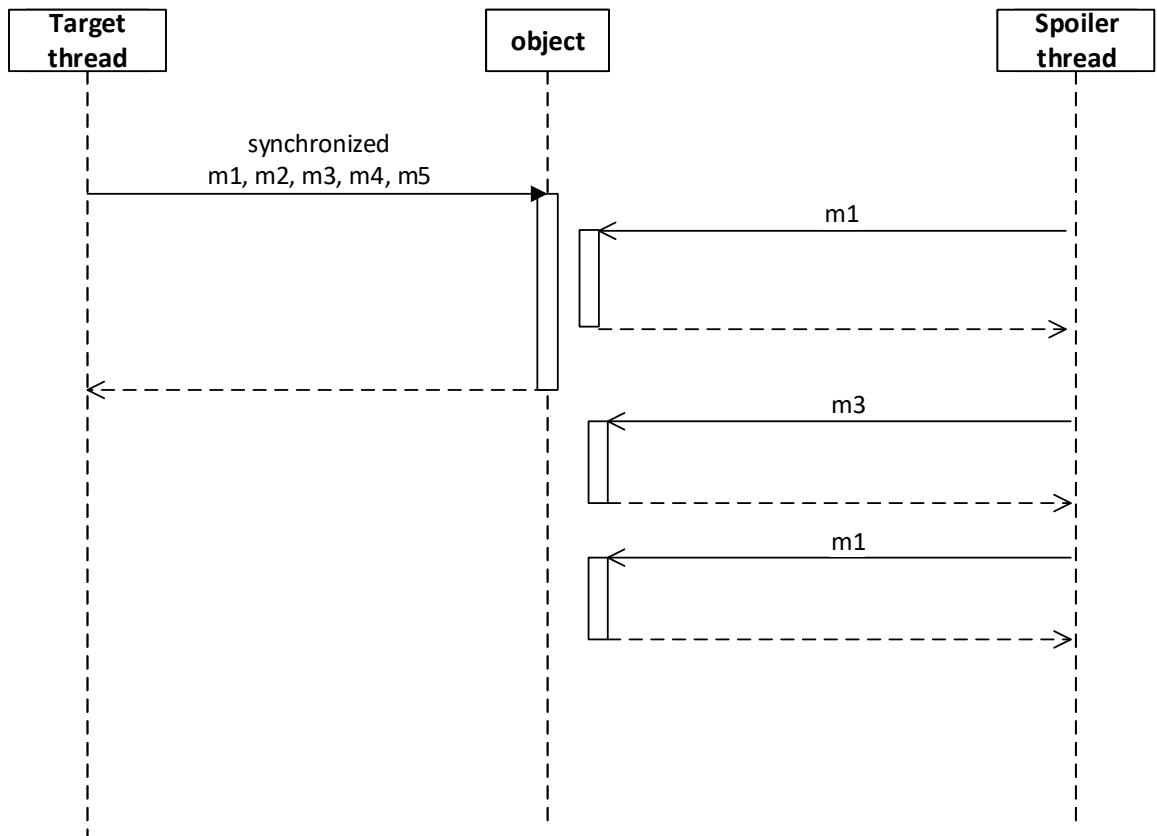
### Test 13

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Obě vlákna volají všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



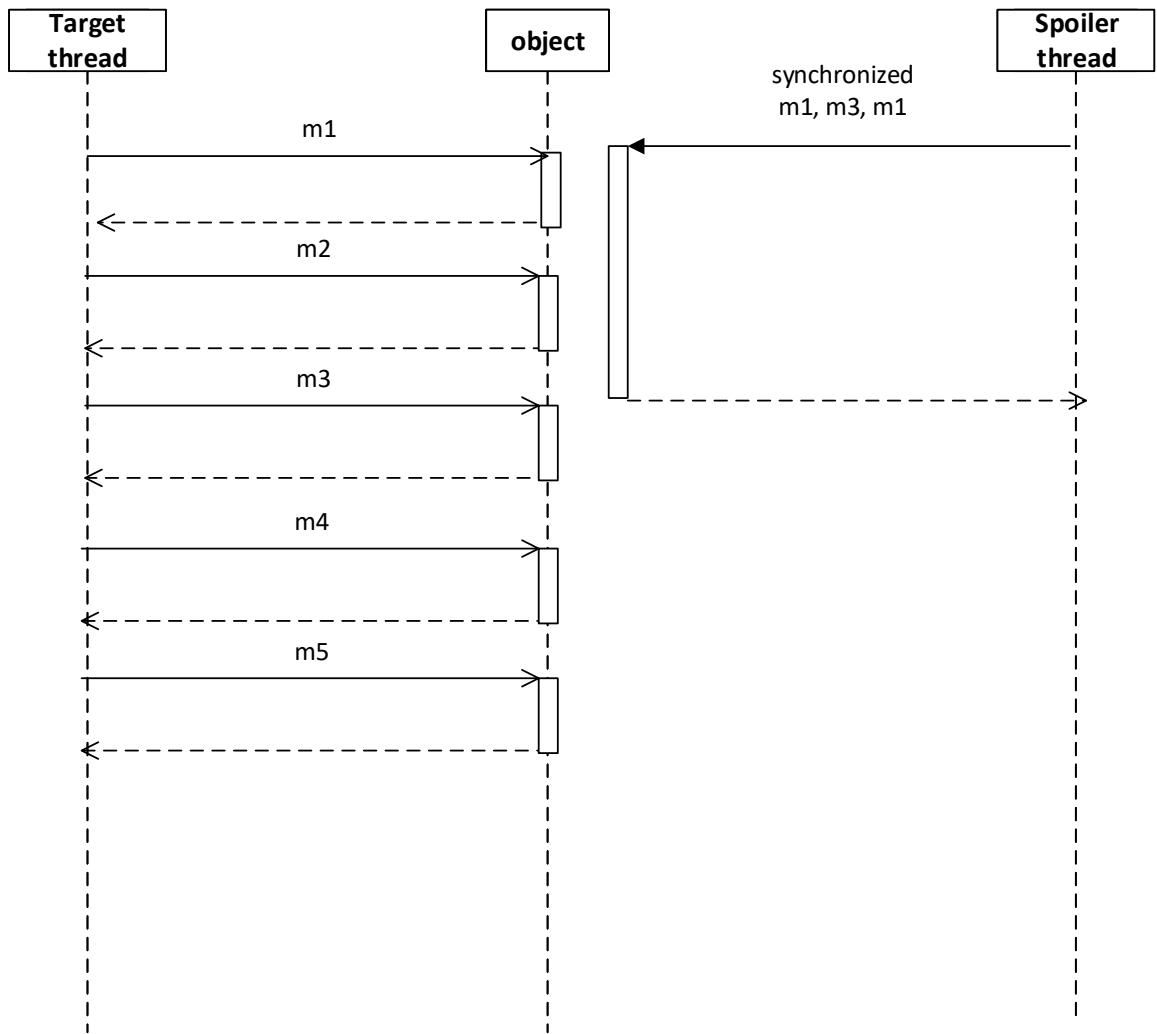
### Test 14

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno  $S$  volá všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



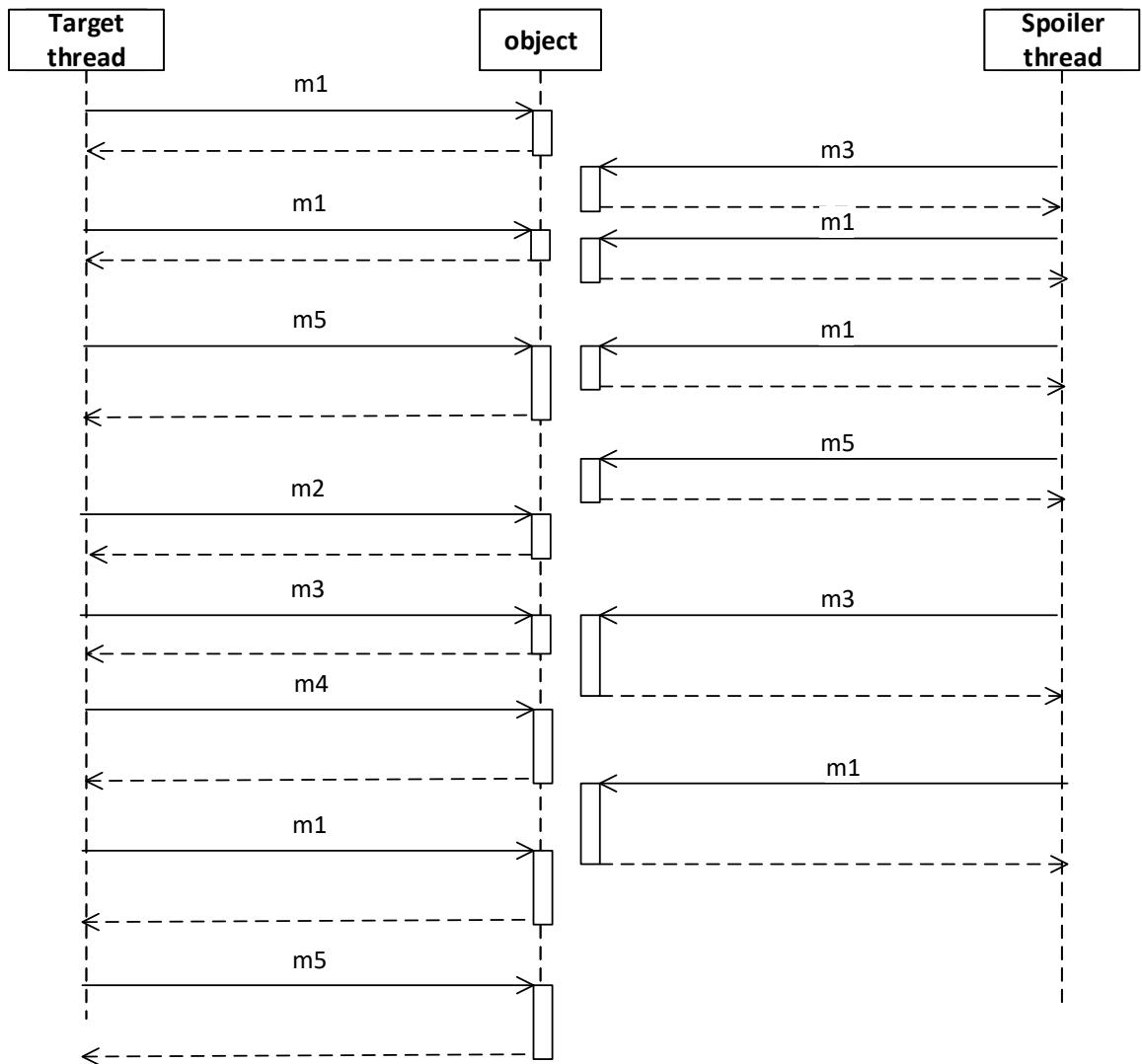
### Test 15

- Kategorie: SLOŽITĚJŠÍ KONTRAKT, VEKTOROVÝ ČAS.
- Popis: Vlákno  $T$  volá všechny metody bez synchronizace.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow \ m1() \ m3() \ m1()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- **Test proběhl úspěšně.**



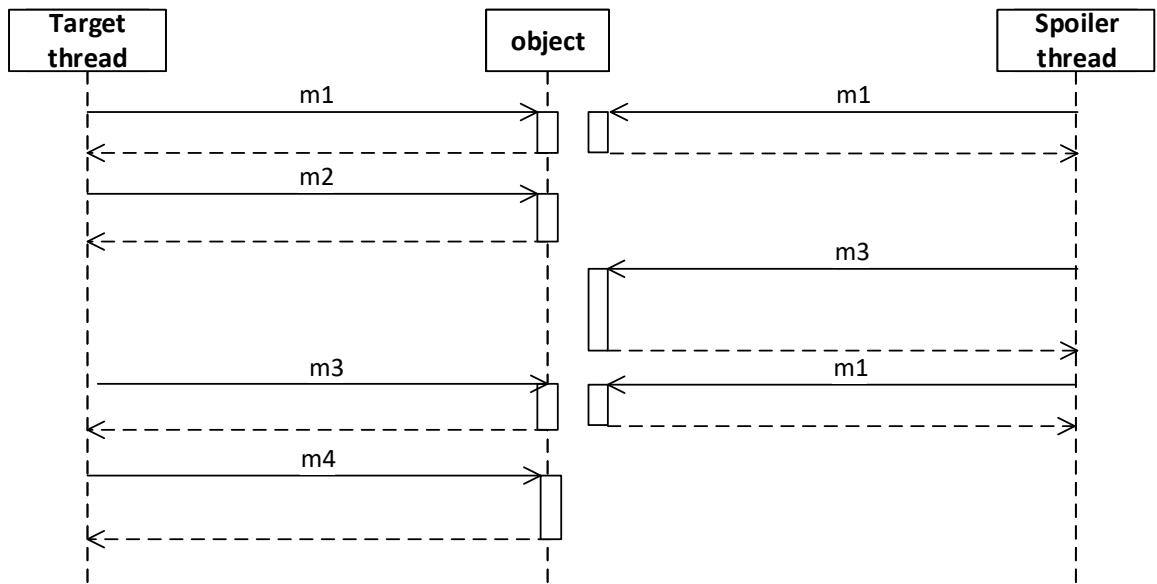
### Test 16

- Kategorie: IGNOROVÁNÍ OSTATNÍCH METOD.
- Popis: Ve vlákně  $T$  i  $S$  vznikne více instancí targetů a spoilerů. Testuje se ignorování ostatních metod.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 4 nalezených porušení.
- Skutečný výsledek: 4 nalezených porušení.
- **Test proběhl úspěšně.**



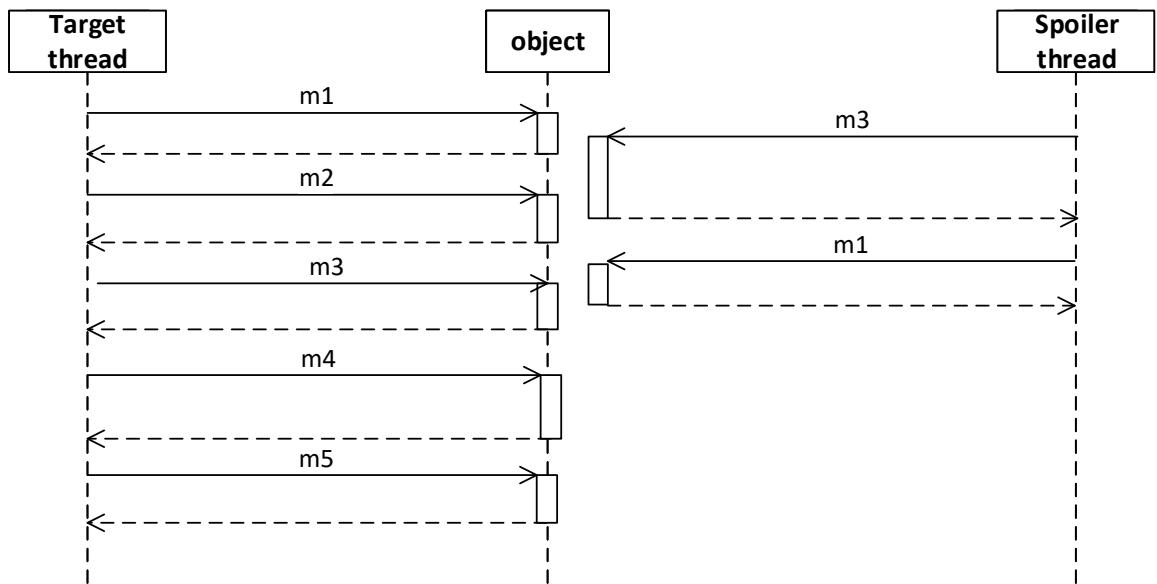
### Test 17

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Target ve vlákně  $T$  není dokončen - chybí poslední metoda.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



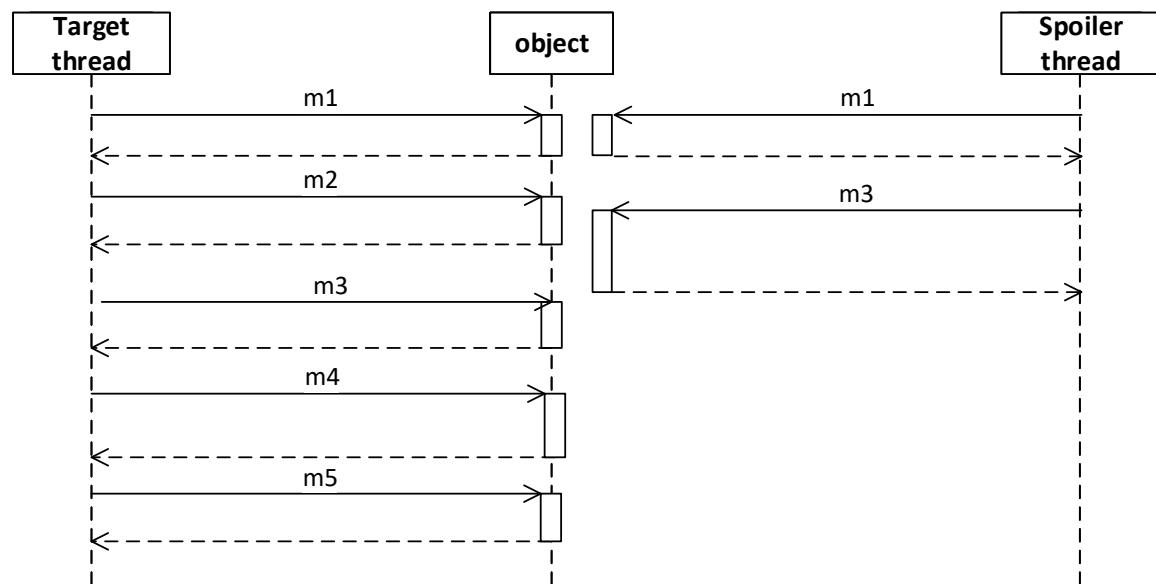
### Test 18

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně  $S$  není dokončen - chybí první metoda.
- Kontrakt:  $m1() \circ m2() \circ m3() \circ m4() \circ m5() \leftarrow m1() \circ m3() \circ m1()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



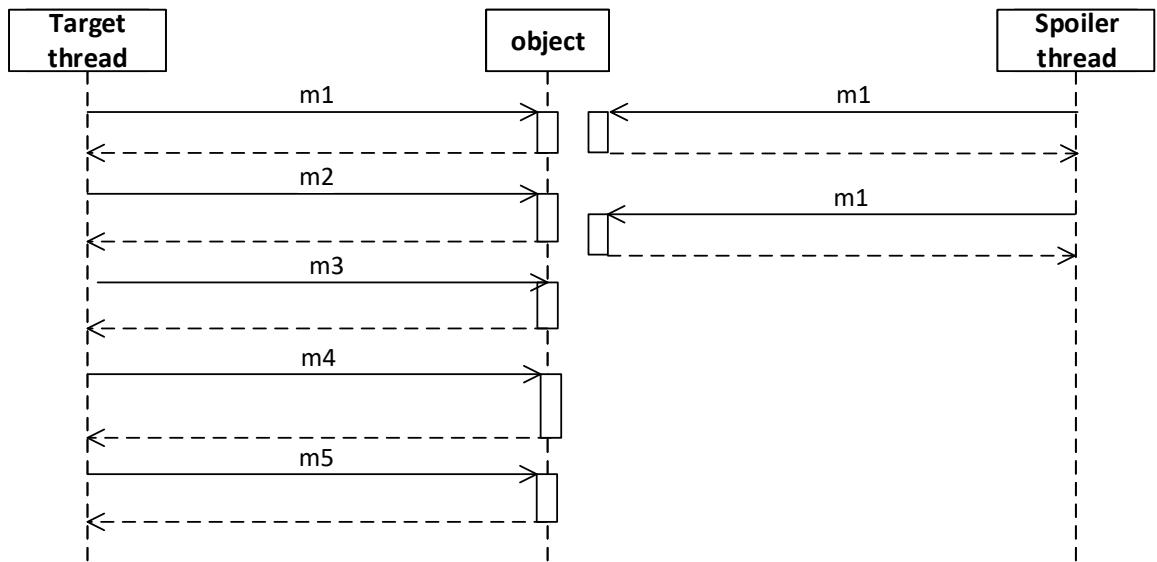
### Test 19

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně  $S$  není dokončen - chybí poslední metoda.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



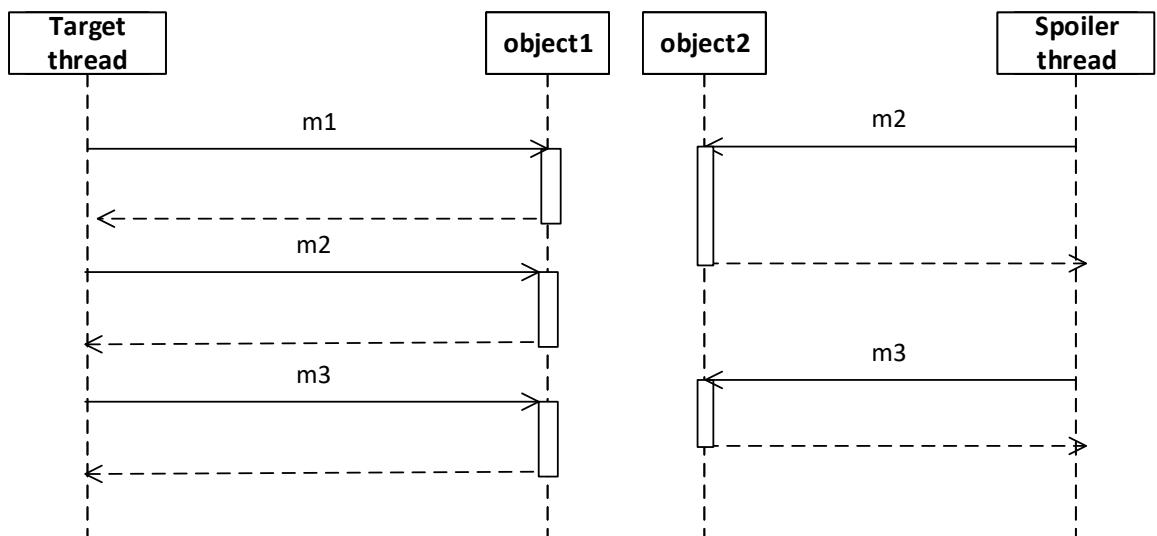
### Test 20

- Kategorie: NEDOKONČENÉ SEKVENCE.
- Popis: Spoiler ve vlákně  $S$  není dokončen - chybí prostřední metoda.
- Kontrakt:  $m1() \ m2() \ m3() \ m4() \ m5() \leftarrow m1() \ m3() \ m1()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



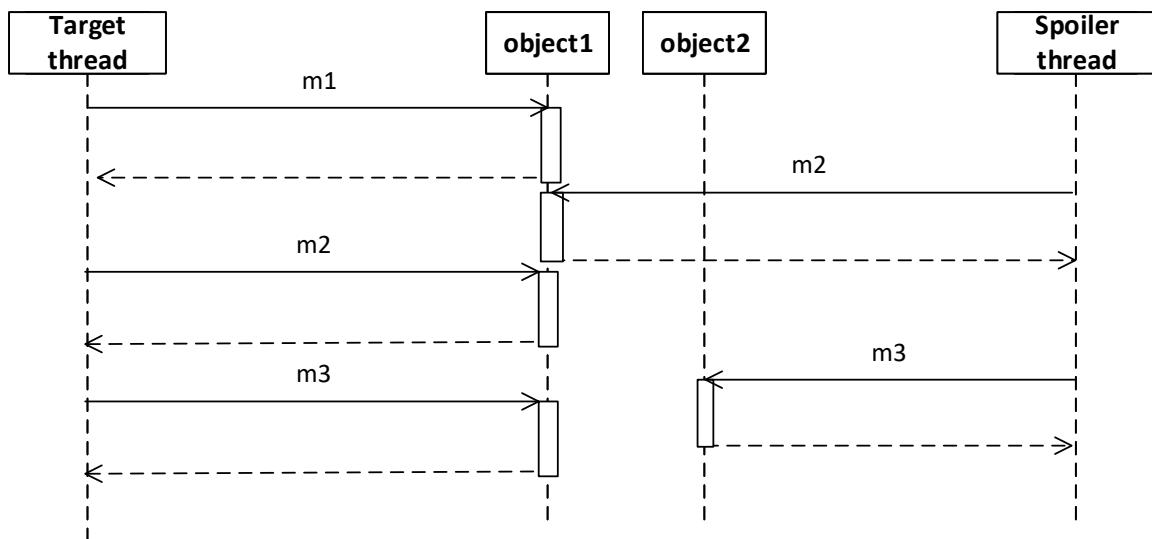
### Test 21

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Target ve vlákně  $T$  je volán na objekt  $o_1$ , ale spoiler ve vlákně  $S$  je volán na objekt  $o_2$ .
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow m2() \ m3()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



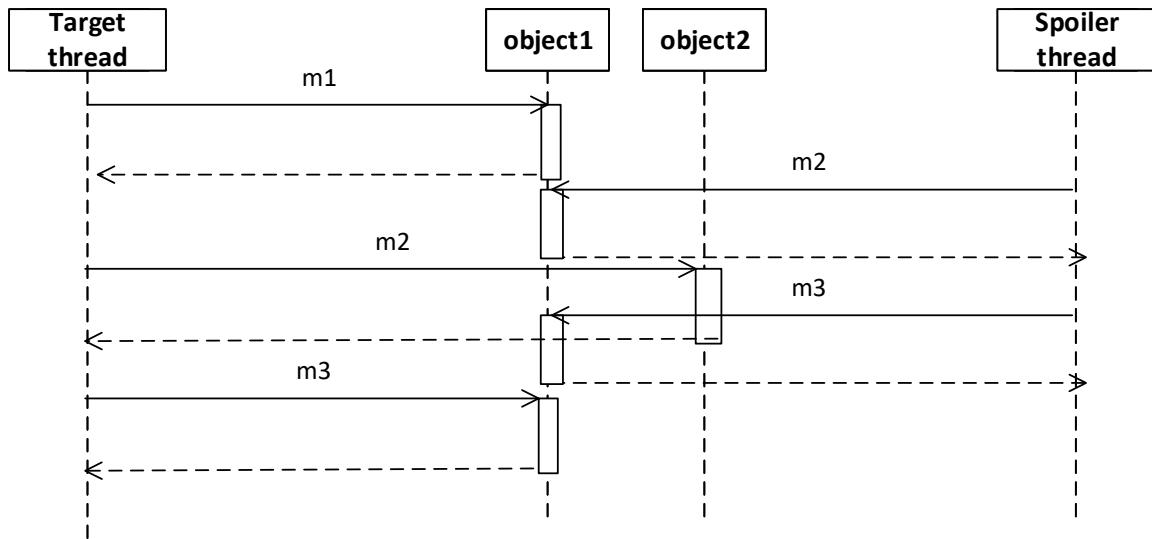
## Test 22

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Spoiler ve vlákně  $S$  volá první metodu  $m2$  na objekt  $o_1$ , ale druhou metodu  $m3$  na objekt  $o_2$ .
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



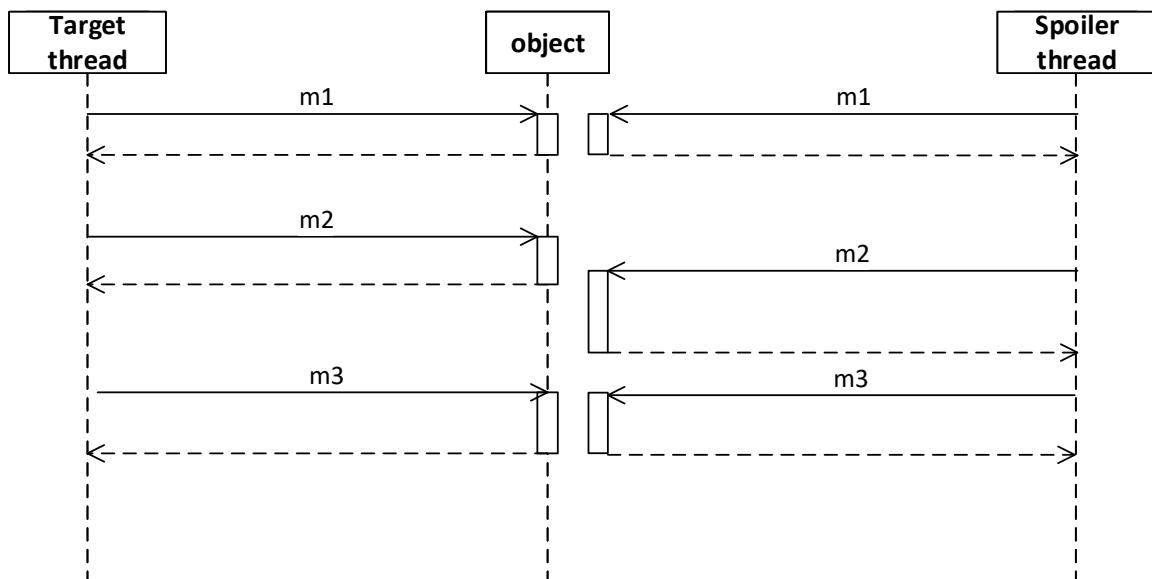
## Test 23

- Kategorie: VÍCE OBJEKTŮ.
- Popis: Target ve vlákně  $T$  volá první metodu  $m2$  a poslední metodu  $m3$  na objekt  $o_1$ , ale druhou metodu  $m2$  na objekt  $o_2$ .
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow \ m2() \ m3()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



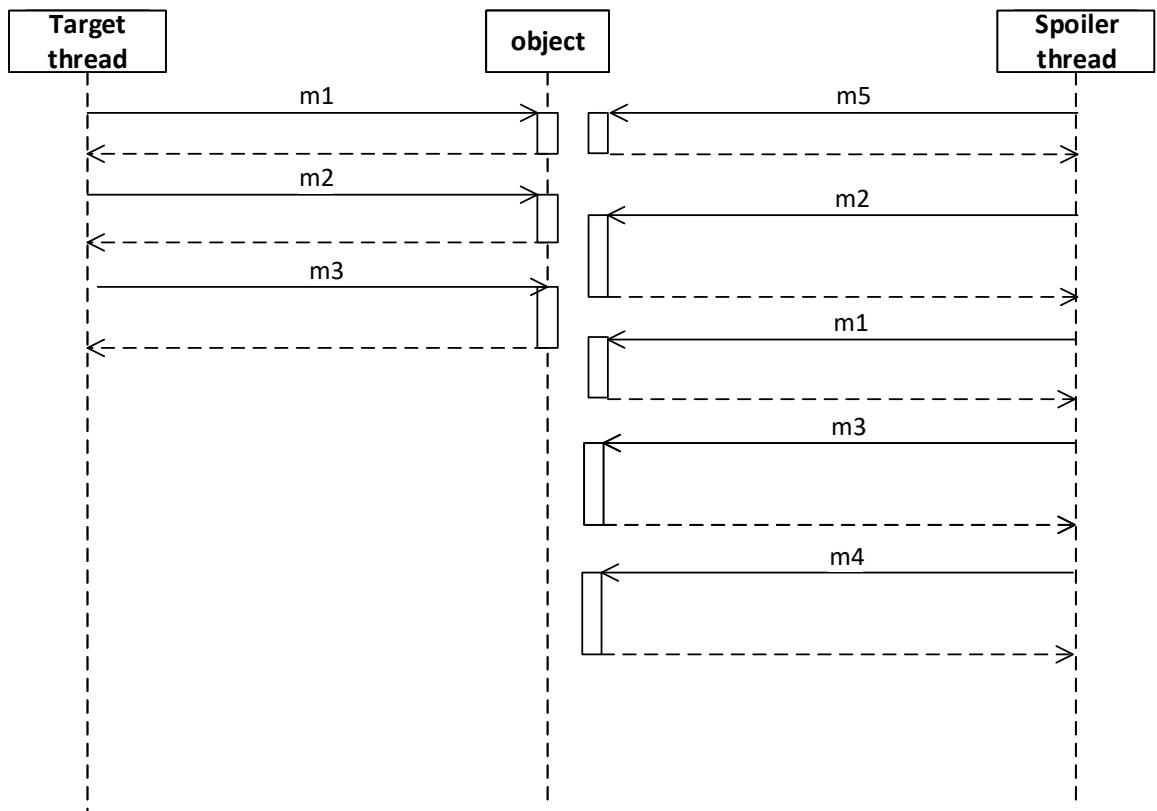
### Test 24

- Kategorie: VÍCE SPOILERŮ.
- Popis: Je provedena pouze jedna definice spoileru. Dojde k nalezení 2 porušení, neboť spoiler i target mohou být nalezeny jako ve vlákně  $T$ , tak i  $S$ .
- Kontrakt:  $m1() \quad m2() \quad m3() \leftarrow m1() \quad m3() \mid m2() \quad m4() \mid m5()$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.
- **Test proběhl úspěšně.**



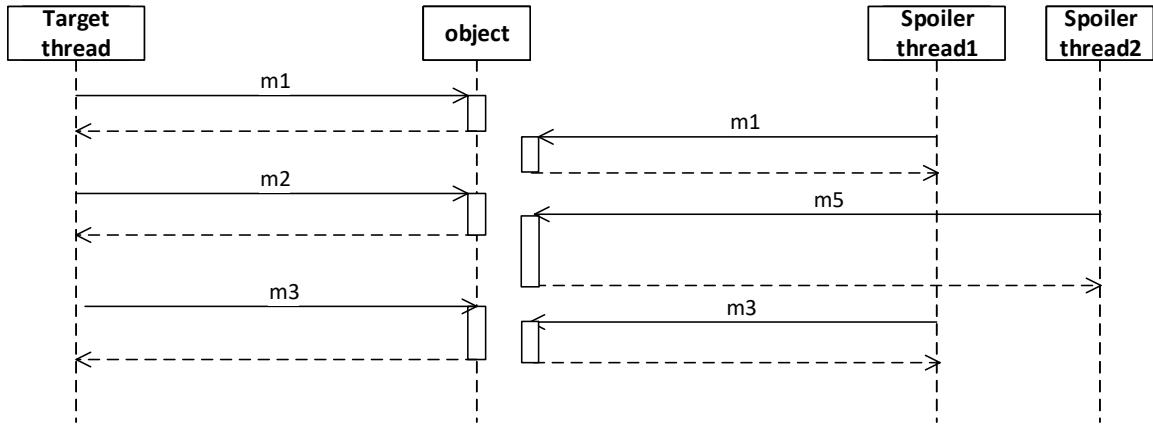
### Test 25

- Kategorie: VÍCE SPOILERŮ.
- Popis: Dojde k nalezení 2 různých spoilerů ( $m1()$   $m3()$ ) a  $m5()$ ) ve vlákne  $S$ .
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow m1() \ m3() \mid m2() \ m4() \mid m5()$
- Očekávaný výsledek: 3 nalezených porušení.
- Skutečný výsledek: 3 nalezených porušení.
- **Test proběhl úspěšně.**



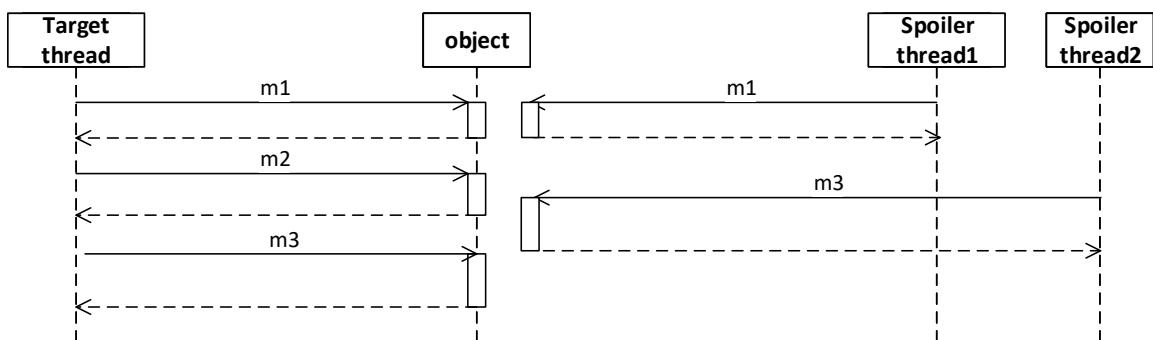
### Test 26

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Dojde k nalezení 2 různých spoilerů ( $m1()$   $m3()$ ) a  $m5()$ ) ve vláknech  $S1$  a  $S12$ .
- Kontrakt:  $m1() \ m2() \ m3() \leftarrow m1() \ m3() \mid m2() \ m4() \mid m5()$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.
- **Test proběhl úspěšně.**



### Test 27

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Nedoje k nalezení žádného spoileru, neboť spoiler musí být vykonán v jednom vlákně.
- Kontrakt:  $m1() \quad m2() \quad m3() \leftarrow m1() \quad m3() \mid m2() \quad m4() \mid m5()$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**

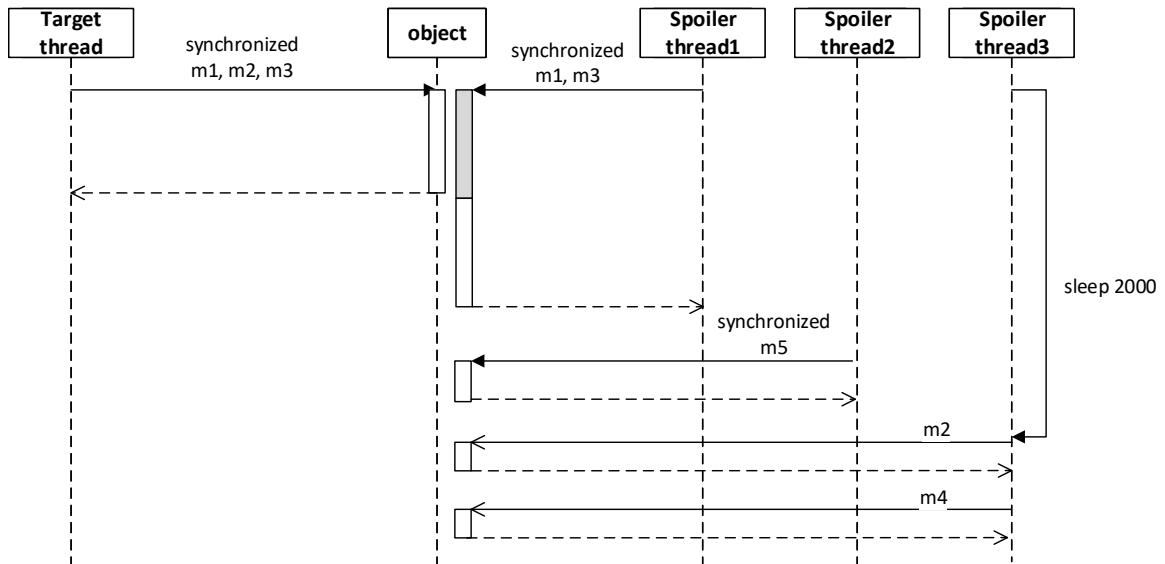


### Test 28

- Kategorie: VÍCE SPOILERŮ, VÍCE VLÁKEN.
- Popis: Dojde k nalezení pouze jednoho spoileru ( $m2() \quad m4()$ ) ve vlákně S 3, neboť v ostatních vláknech je použita správná synchronizace.
- Kontrakt:  $m1() \quad m2() \quad m3() \leftarrow m1() \quad m3() \mid m2() \quad m4() \mid m5()$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.

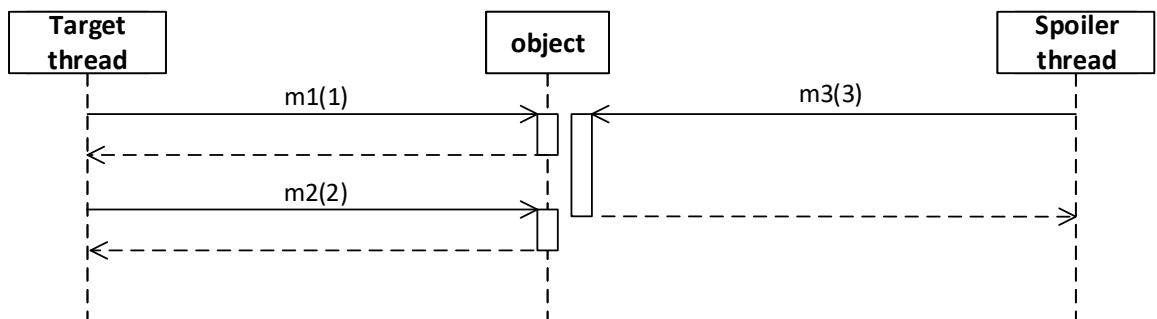
- Test proběhl úspěšně.

... TODO spätý diagram ...



### Test 29

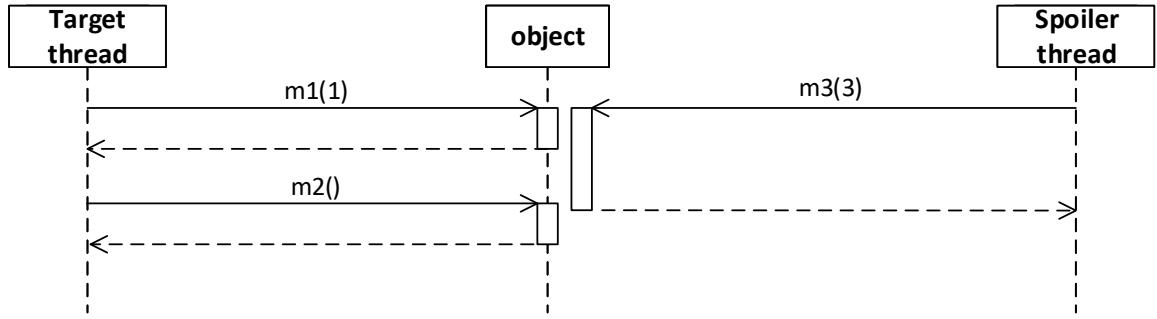
- Kategorie: PARAMETRY.
- Popis: Parametry metod se shodují s definicí kontraktu.
- Kontrakt:  $m1(A) \quad m2(B) \quad \leftarrow \quad m3(C)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.
- Test proběhl úspěšně.



### Test 30

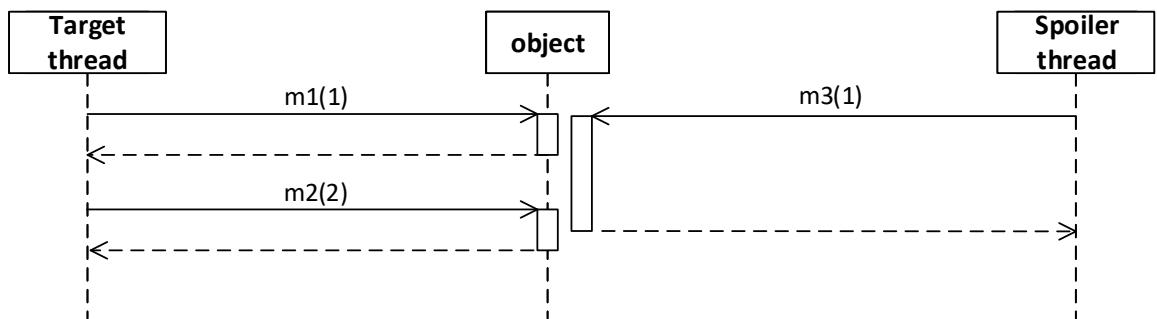
- Kategorie: PARAMETRY.
- Popis: Metoda `m2()` je zavolána bez parametru.

- Kontrakt:  $m1(A) \ m2(B) \leftarrow \ m3(C)$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.
- **Test proběhl úspěšně.**



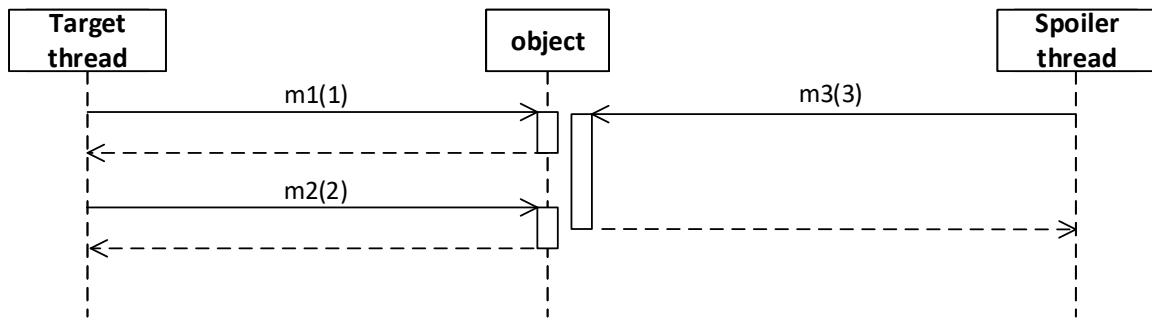
### Test 31

- Kategorie: PARAMETRY.
- Popis: Parametry metod  $m1$  a  $m3$  se shodují.
- Kontrakt:  $m1(A) \ m2(B) \leftarrow \ m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



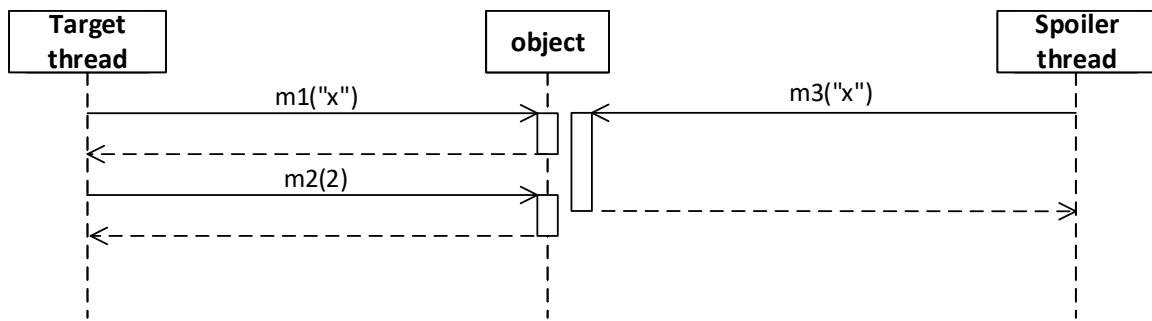
### Test 32

- Kategorie: PARAMETRY.
- Popis: Parametry metod  $m1$  a  $m3$  se neshodují.
- Kontrakt:  $m1(A) \ m2(B) \leftarrow \ m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



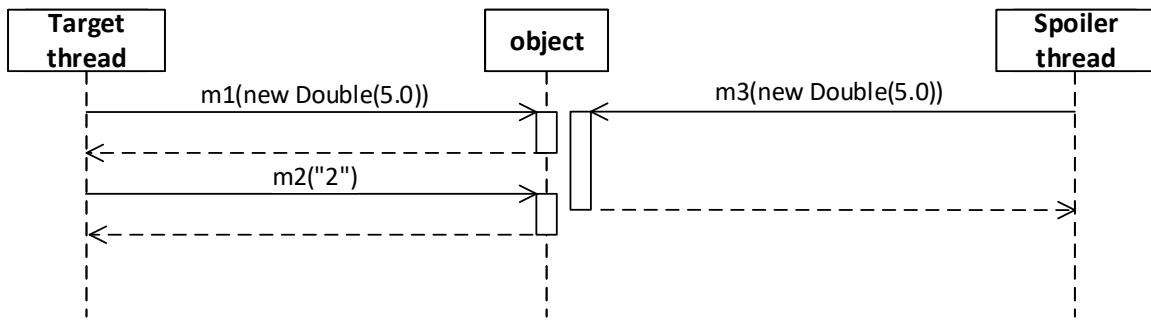
### Test 35

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách  $m1$  a  $m3$  jsou typu *String*, oproti tomu parametr metody  $m2(2)$  je primitivního typu *int*.
- Kontrakt:  $m1(A) \ m2(B) \leftarrow \ m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



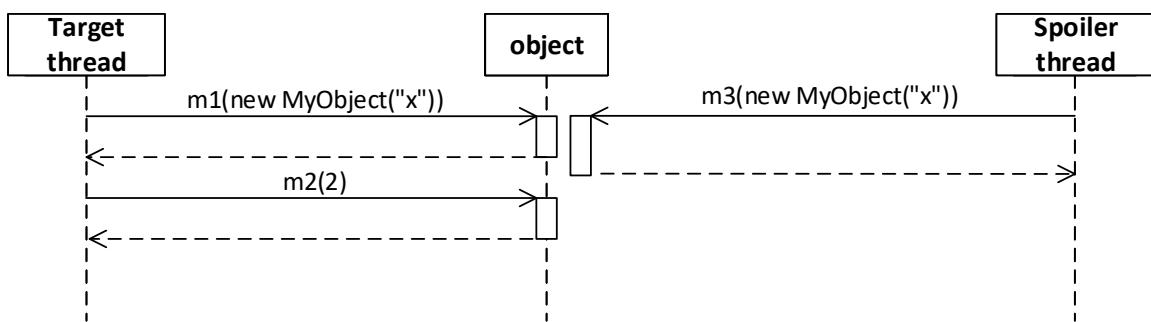
### Test 36

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách  $m1$  a  $m3$  jsou typu *Double*, oproti tomu parametr metody  $m2$  je primitivního typu *int*.
- Kontrakt:  $m1(A) \ m2(B) \leftarrow \ m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



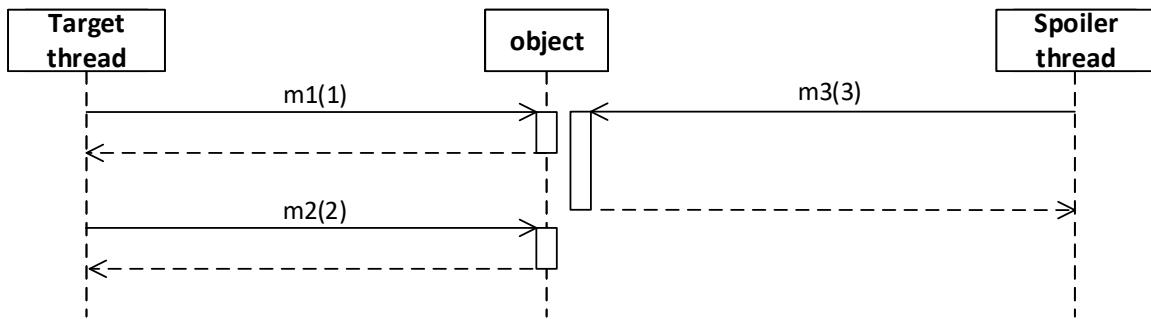
### Test 37

- Kategorie: RŮZNÉ TYPY PARAMETRŮ.
- Popis: Parametry v metodách *m1* a *m3* jsou instance nové třídy *MyObject* s definovanou *equals* metodou, oproti tomu parametr metody *m2* je primitivního typu *int*.
- Kontrakt:  $m1(A) \quad m2(B) \leftarrow m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



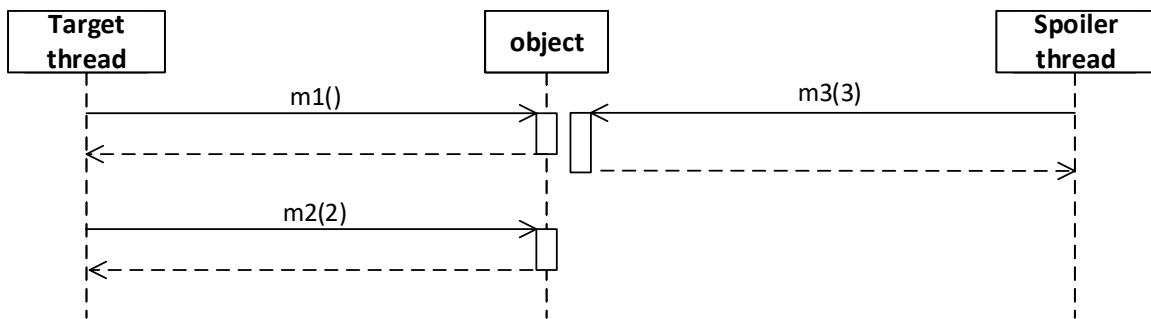
### Test 38

- Kategorie: IGNOROVÁNÍ PARAMETRŮ.
- Popis: Parametr metod *m1* a *m3* je ignorován, takže nezáleží na jeho hodnotě.
- Kontrakt:  $m1(\_) \quad m2(X) \leftarrow m3(\_)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



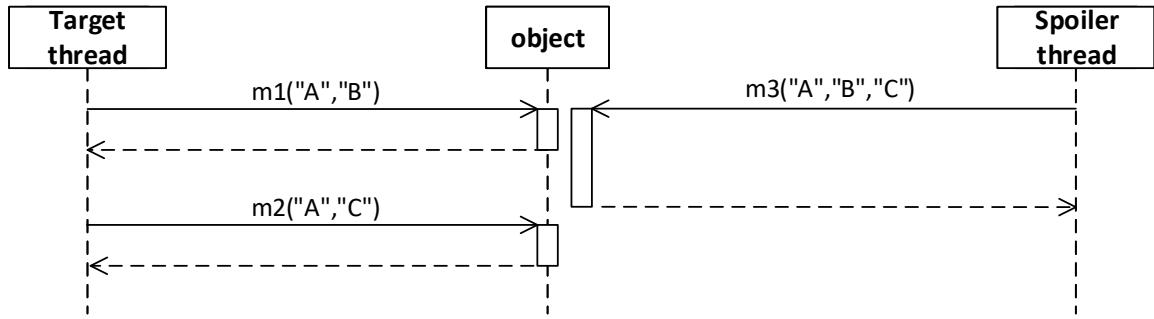
### Test 39

- Kategorie: IGNOROVÁNÍ PARAMETRŮ.
- Popis: Parametr metody  $m1$  není zadán. Hodnota a typ parametru jsou ignorovány, ale počet musí souhlasit. Tím pádem není nalezena žádná instance targetu.
- Kontrakt:  $m1(\_) \ m2(X) \leftarrow \ m3(\_)$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.



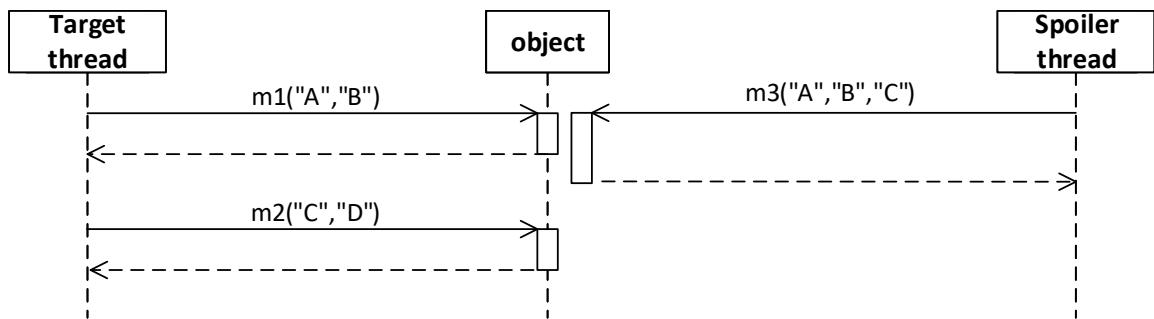
### Test 40

- Kategorie: VÍCE PARAMETRŮ.
- Popis: Parametry mezi targetem a spoilerem souhlasí, takže porušení je nalezeno.
- Kontrakt:  $m1(A,B) \ m2(A,C) \leftarrow \ m3(A,B,C)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



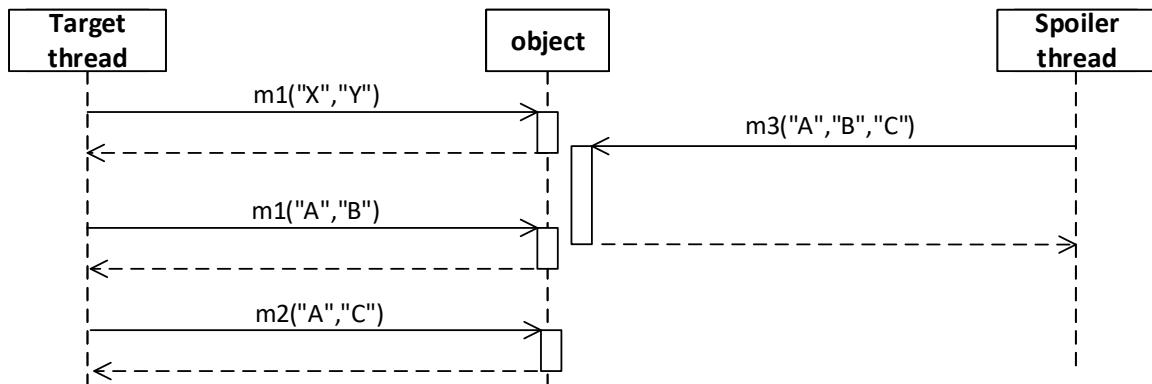
### Test 41

- Kategorie: VÍCE PARAMETRŮ.
- Popis: V tagetu, ve vlákně  $T$ , se neshoduje první parametr u metod  $m1$  a  $m2$ , takže neexistuje instance tagetu.
- Kontrakt:  $m1(A, B) \quad m2(A, C) \leftarrow m3(A, B, C)$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.



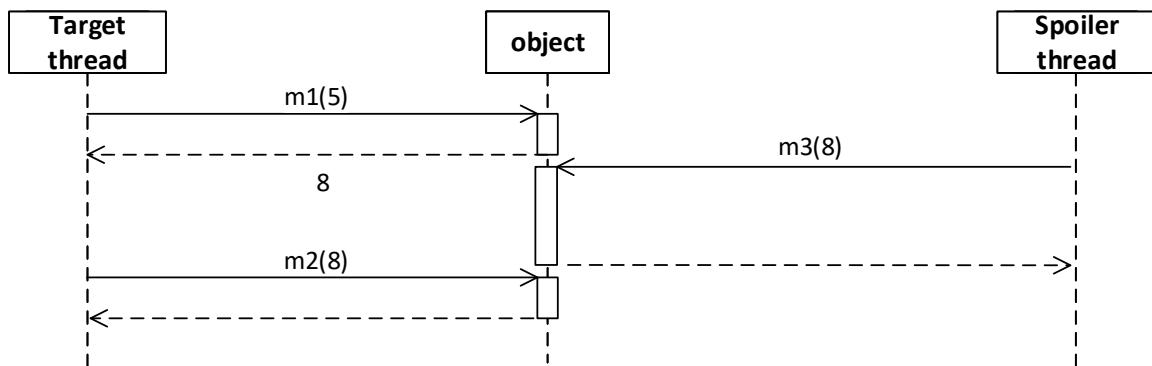
### Test 42

- Kategorie: VÍCE PARAMETRŮ.
- Popis: Dokončená instance tagetu obsahuje až druhou metodu  $m1$ , protože u první metody  $m1$  se neshodují parametry s metodou  $m2$  a ani s metodou  $m3$  ve vláknu spoileru  $S$ .
- Kontrakt:  $m1(A, B) \quad m2(A, C) \leftarrow m3(A, B, C)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



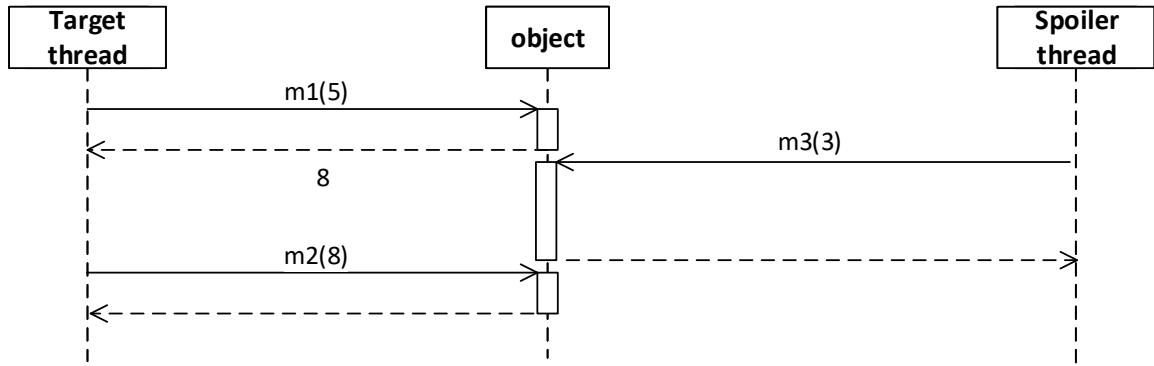
### Test 43

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr metody  $m1$  je shodný s parametry metod  $m2$  a  $m3$ .
- Kontrakt:  $A:m1(\_) \quad m2(A) \leftarrow m3(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



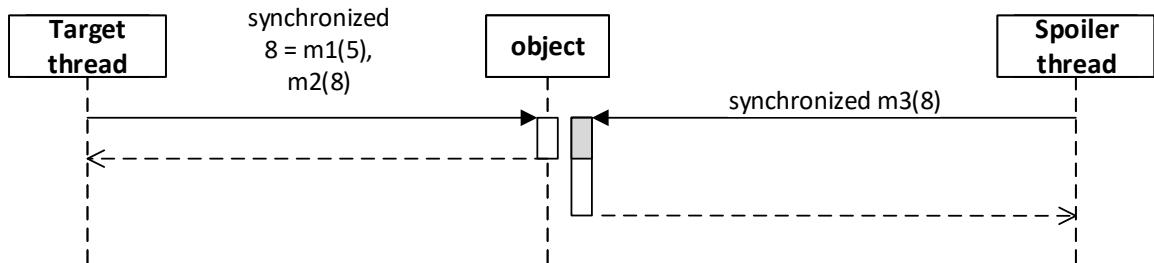
### Test 44

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr metody  $m1$  není shodný s parametrem metody  $m3$  ve vláknu spoileru  $S$ .
- Kontrakt:  $A:m1(\_) \quad m2(A) \leftarrow m3(A)$
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.



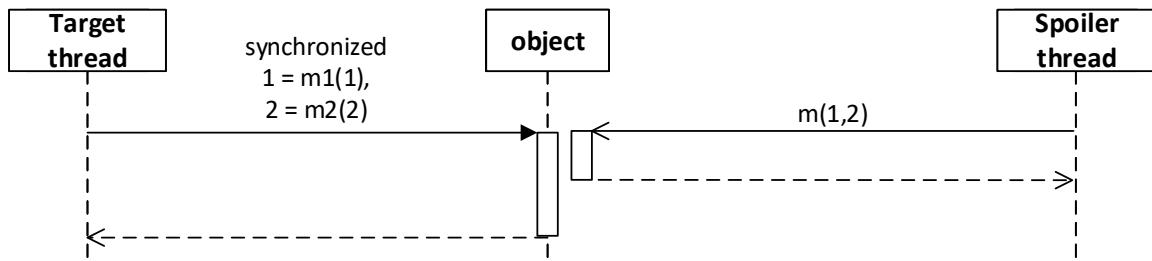
### Test 45

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Parametry jsou shodné, ale je použita správná synchronizace a proto k porušení nedojde.
- Kontrakt: A:m1(\_) m2(A) <- m3(A)
- Očekávaný výsledek: 0 nalezených porušení.
- Skutečný výsledek: 0 nalezených porušení.



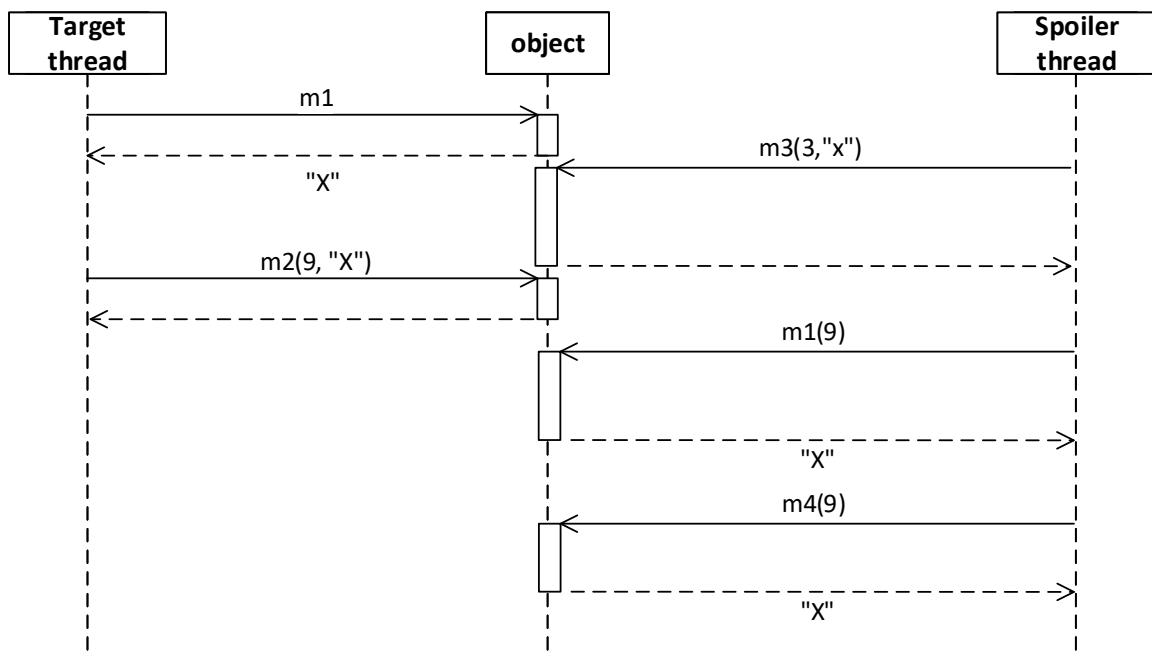
### Test 46

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Návratový parametr je kontrolován u dvou metod targetu a parametry z obou metod jsou použity jako vstupní parametry do metody ve spoileru. Metody ve vláknu targetu T jsou volány se správnou synchronizací, ale ve vláknu spoileru S, jsou volány bez synchronizace.
- Kontrakt: X:m1() Y:m2() <- m3(X, Y)
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



### Test 47

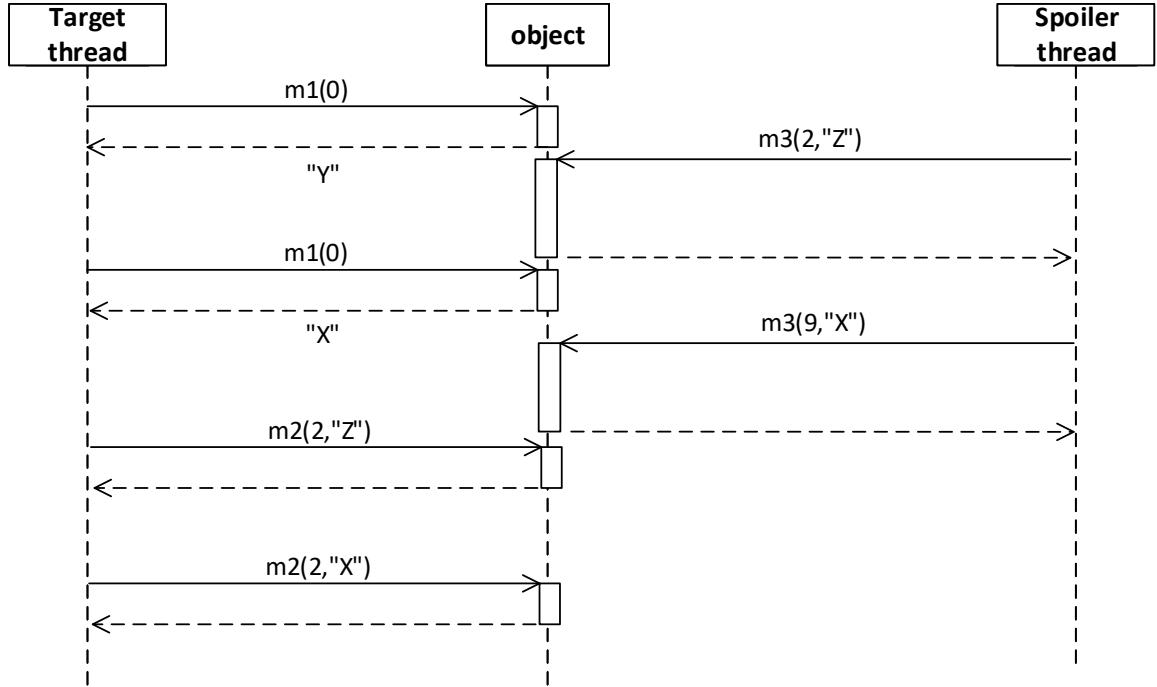
- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Je definováno více spoilerů na jeden target s kombinací parametrů a návratového parametru.
- Kontrakt:  $X:m1(\_) \quad m2(A, X) \leftarrow m3(\_, X) \mid m1(A) \quad X:m4(A)$
- Očekávaný výsledek: 2 nalezených porušení.
- Skutečný výsledek: 2 nalezených porušení.



### Test 48

- Kategorie: VÍCE PARAMETRŮ, IGNOROVÁNÍ PARAMETRŮ, NÁVRATOVÝ PARAMETR.
- Popis: Dokončená instance tageru, ve vláknu  $S$ , je tvořena až druhými metodami  $m1$  a  $m2$ , protože by jinak nesouhlasí parametry s ohledem na definici kontraktu. Totéž platí u metody  $m3$  ve vláknu targetu  $T$ .

- Kontrakt:  $X:m1(\_) \quad m2(A, X) \leftarrow m3(\_, X) \mid m1(A) \quad X:m4(A)$
- Očekávaný výsledek: 1 nalezených porušení.
- Skutečný výsledek: 1 nalezených porušení.



### 6.3 Testování SearchBestie

[Poznamky z konzultace] Cim vic testu, tim lip - bud kapitola, nebo priloha. Popsat samotny test co co znamena a pak vytvorit reprezentaci jednotlivych testu a ty tam dat. Idealne diagram + kratky popis. Vstupy + vystupy a jak to dopadlo. 40 nebo i 100 ruznych testu - jak mi zbude cas Kdyz toho bude moc, tak to bude priloha, ale musi tam byt to, jak se to cte. Vymyslet reprezentaci jak budou ty testy reprezentovany v dokumentaci. Dat tam diagram a popsat co se tam deje. (co je na vstupu)

## **Kapitola 7**

### **Závěr**

[Poznamky z konzultace] Dalsi pokracovani v projektu, sebekritiky! (co to neumi, jak by se to melo upravit aby to umelo, kde je slaba stranka, neni to optimalizovane na velke programy atd.), cim vic se priznam, ze neco neni dodelane, tim lip Nedostatek je treba, ze se nevypousti ty metody z toho okna.

kdyz je provede target se spoilerem ve stejnem vlakne, tak to nezdetekuje velke mnozstvi instanci muze byt problem - nezahazuji se

# **Kapitola 8**

## **SP: 1. Testování paralelních programů**

### **8.1 Paralelní program**

# Literatura

- [1] Andrews, G. R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000, ISBN 0-201-35752-6.
- [2] Center, N. A. R.: *Java<sup>TM</sup> Pathfinder*. [Online; navštíveno 09.01.2017]. URL <https://http://babelfish.arc.nasa.gov/trac/jpf>
- [3] Edelstein, O.; Farchi, E.; Goldin, E.; aj.: *Framework for testing multi-threaded Java programs*. *Concurrency and Computation: Practice and Experience*, ročník 15, 2003: s. 485–499, doi:10.1002/cpe.654.  
URL <http://doi.wiley.com/10.1002/cpe.654>
- [4] Ferreira, C.; andand Diogo G. Sousa, J. L.; Dias, R. J.: *Preventing atomicity violations with contracts*. In *eprint arXiv:1505.02951*, 2015.  
URL <https://arxiv.org/abs/1505.02951>
- [5] Ferreira, C.; Fiedor, J.; Lourenco, J.; aj.: *Verifying Concurrent Programs Using Contracts*. To appear In *Proceedings of ICST Conference*, 2017.
- [6] Fiedor, J.; Dudka, V.; Křena, B.; aj.: *Advances in Noise-based Testing of Concurrent Programs*. Software Testing, Verification and Reliability, ročník 25, č. 3, 2015: s. 272–309, ISSN 1099-1689.  
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=10275>
- [7] Fiedor, J.; Letko, Z.; Lourenco, J.; aj.: *Dynamic Validation of Contracts in Concurrent Code*. In Proceedings of the 15th International Conference on Computer Aided Systems Theory, *The Universidad de Las Palmas de Gran Canaria*, 2015, ISBN 978-84-606-5438-4, s. 177–178.  
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=10817>
- [8] Flanagan, C.; Freund, S. N.: *FastTrack: Efficient and Precise Dynamic Race Detection*. In Proceedings of the 15th International Conference on Computer Aided Systems Theory, ročník 53, *Communications of the ACM*, 2010, s. 93–101, doi:10.1145/1839676.1839699.  
URL <http://portal.acm.org/citation.cfm?doid=1839676.1839699>
- [9] Flanagan, C.; Freund, S. N.: *The RoadRunner Dynamic Analysis Framework for Concurrent Programs*. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, *ACM New York, USA*, 2010, ISBN 978-1-4503-0082-7, s. 1–8.  
URL <https://users.soe.ucsc.edu/~cormac/papers/paste10.pdf>
- [10] Kshemkalyani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge Press, 2011, ISBN 9780521189842.

- [11] Křena, B.; Letko, Z.; Ur, S.; aj.: *A Platform for Search-Based Testing of Concurrent Software*. In PADTAD '10, Proceedings of the 8th Workshop on Parallel and Distributed Systems, Association for Computing Machinery, 2010, ISBN 978-1-60558-823-0, str. 11.  
URL <http://www.fit.vutbr.cz/research/viewpub.php?id=9275>
- [12] Křena, B.; Letko, Z.; Vojnar, T.: *Noise Injection Heuristics for Concurrency Testing*. Lecture Notes in Computer Science, ročník 2012, č. 7119, 2012: s. 123–131, ISSN 0302-9743.  
URL <http://www.fit.vutbr.cz/research/viewpub.php.cs?id=9725>
- [13] Lamport, L.: *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM, ročník 21, 1978: s. 558–565, doi:10.1145/359545.359563.  
URL <http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf>
- [14] Meyer, B.: *Applying Design by Contract*. In Computer, IEEE Computer Society Press Los Alamitos, CA, USA, 1992, s. 40–51, doi:10.1109/2.161279.  
URL <http://se.ethz.ch/~meyer/publications/computer/contract.pdf>
- [15] Oracle: Java documentation. [Online; navštíveno 08.01.2017].  
URL <https://docs.oracle.com/javase/8/docs/>
- [16] Oracle: Processes and Threads. [Online; navštíveno 08.01.2017].  
URL <https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>
- [17] Per, B. H.: *Operating system principles*. Englewood Cliffs, 2003, ISBN 0-13-026611-6.
- [18] Savage, S.; Burrows, M.; Nelson, G.; aj.: *Eraser: A Dynamic Data Race Detector for Multi-threaded Programs*. In Proceedings of the sixteenth ACM symposium on Operating systems principles, ACM Transactions on Computer Systems, 1997.  
URL <http://homes.cs.washington.edu/~tom/pubs/eraser.pdf>
- [19] Stallings, W.: *Operating Systems: Internals and Design Principles*. Pearson, 8 vydání, 2015, ISBN 978-0133805918.
- [20] Stevens, R.: *UNIX Network Programming Second Edition: Interprocess Communications*, ročník 2. Prentice Hall, 1999, ISBN 0-13-081081-9.