VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZŠÍŘENÍ FRAMEWORKU ANACONDA PRO PODPORU KONTRAKTŮ S PARAMETRY A JEJICH OMEZENÍMI

TOWARDS PARAMETERIZED CONTRACT VALIDATOR IN ANACONDA FRAMEWORK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MONIKA MUŽIKOVSKÁ**
AUTHOR

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

BRNO 2018

Zadání bakalářské práce/20642/2017/xmuzik05

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Mužikovská Monika**
Obor: Informační technologie
Téma: **Rozšíření frameworku ANaConDA pro podporu kontraktů s parametry a jejich omezeními**
Towards Parameterized Contract Validator in ANaConDA Framework
Kategorie: Analýza a testování softwaru

Pokyny:

1. Nastudujte metody testování vícevláknových programů. Nastudujte projekt ANaConDA.
2. Analyzujte požadavky pro analýzu parametrizovaných kontraktů pro paralelismus. Navrhněte způsob testování parametrizovaných kontraktů.
3. Implementujte analyzátor pro parametrizované kontrakty jako rozšíření frameworku ANaConDA.
4. Ověřte funkcionalitu dosaženého řešení na množině základních testů. Automatizujte testovací případy k umožnění průběžné integrace frameworku ANaConDA.

Literatura:

- Fiedor, J.; Vojnar. T.: ANaConDA: 2012. *A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level*. In Proc. of 3rd International Conference on Runtime Verification, Springer-Verlag. doi: 10.1007/978-3-642-35632-2_5
- Dias, R. J.; Ferreira, C.; Fiedor, J.; Lourenço, J. M.; Smrčka A.; Sousa, D. G.; Vojnar T.: 2017. Verifying Concurrent Programs Using Contracts. In Proc. of 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE. doi: 10.1109/ICST.2017.25

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se zabývá problematikou kontraktů pro paralelismus. Jedná se o protokol umožňující specifikovat požadavky na atomicitu v paralelních programech a následnou tvorbu automatických nástrojů pro detekci porušení atomicity. Součástí prostředí ANaConDA pro dynamickou analýzu programů je nástroj pro detekci tohoto druhu chyb, ale jeho výsledky mohou být příliš obecné. Cílem práce bylo navrhnout a implementovat metodu, která bude podporovat kontrakty rozšířené o parametry a jejich omezení, což povede k přesnějším výsledkům analýzy. Experimenty provedené pomocí nově vzniklého analyzátoru na programech se známými chybami ukázaly, že díky zahrnutí parametrů do analýzy je možné výsledky zredukovat až o desítky hlášení o situacích, které při zohlednění kontextu nejsou chybné a pouze zbytečně zatěžovaly vývojáře a znesnadňovaly odhalení skutečných chyb.

Abstract

This work deals with problematics of contracts for parallelism. It is a technique allowing to specify requirements for atomicity in parallel programs and to create automatic tools for detection of atomicity violation. ANaConDA framework provides dynamic analyser called Contract-validator which can detect contract violations in parallel programs. However, due to analysis without context, it can produce a lot of warnings about contract violation that are not considered as errors. The aim of this work was to design and implement a method supporting contracts extended with parameters and their constraints which will lead to more accurate results of the analysis. Experiments using newly created analyser on a set of benchmarks with known atomicity violations showed that analysis with parameters can reduce the results by dozens of reports that unnecessarily burdened developers and made it harder to reveal real errors.

Klíčová slova

ANaConDA, dynamická analýza, kontrakty, parametry, porušení atomicity, vícevláknové programování

Keywords

ANaConDA framework, dynamic analysis, contracts, parameters, atomicity violation, multithreading

Citace

MUŽIKOVSKÁ, Monika. *Rozšíření frameworku ANaConDA pro podporu kontraktů s parametry a jejich omezeními*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Rozšíření frameworku ANaConDA pro podporu kontraktů s parametry a jejich omezeními

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Další informace mi poskytl autor nástroje ANaConDA pan Ing. Jan Fiedor, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Monika Mužikovská
10. května 2018

Poděkování

Chtěla bych poděkovat Ing. Alešovi Smrčkovi, Ph.D. a Ing. Janu Fiedorovi, Ph.D. za jejich vedení a rady, které mi při práci velmi pomáhaly.

Obsah

Kapitola 1

Úvod

Aby programy mohly plně využívat výhody zpracování více instrukcí současně na vícejádrových procesorech, je nutné, aby vývojáři pro jejich tvorbu využívali paralelismus, například pomocí vláken v jazyce C++. Paralelní programy sice přináší nesporné výhody z hlediska výkonu, ale jejich tvorba je v porovnání se sekvenčními programy náročnější. Také se vyznačují svým nedeterministickým chováním, což znesnadňuje odhalování a opravy speciálních paralelních chyb, kterými často trpí. V dnešní době již existují účinné metody na testování vícevláknových programů a odhalení nejznámějších chyb (například dynamická analýza, což je jedna z oblastí zájmu výzkumné skupiny VeriFIT[1]), ale u komplexnějších programů může být obtížné vůbec tyto chyby popsat a specifikovat korektní chování programu.

Článek Verifying Concurrent Programs using Contracts [3], který v nedávné době vyšel na konferenci ICST 2017 a ze kterého tato práce vychází, se zabývá problematikou rozhraní modulů. Konkrétně specifikuje protokol nazvaný kontrakt pro paralelismus umožňující popsat, které metody API je nutné vykonat ve vícevláknových programech atomicky. Díky existenci tohoto popisu je možné automatizovaně analyzovat, zda v programech dochází k jeho porušení. Zmíněný článek také navrhuje metody pro statickou a dynamickou analýzu kontraktů, přičemž jedna z těchto metod byla implementována jako analyzátor do prostředí ANaConDA, které poskytuje podporu pro dynamickou analýzu C/C++ programů na binární úrovni.

Kontrakt je v základní podobě specifikován pouze pomocí názvů metod, které se musí vykonat atomicky. Je však možné do specifikace zahrnout také parametry těchto metod a tím dodat informaci o kontextu a toku dat, která zpřesní výsledky analýzy. Taková úprava navíc může podstatně omezit počet nalezených chyb a ušetřit práci při jejich opravách. V rámci této práce bylo navrženo několik přístupů k analýze parametrických kontraktů, z nichž každý má svá úskalí. Metoda, která se jeví jako nejefektivnější pro analýzu reálných programů, byla implementována jako další analyzátor do nástroje ANaConDA. Oba analyzátory byly podrobeny experimentům nad stejnou sadou programů, aby mohly být porovnány jejich výsledky, časová a paměťová náročnost.

V kapitole 2 jsou popsány metody pro testování vícevláknových programů a framework ANaConDA, který poskytuje podporu pro jejich dynamickou analýzu. Kapitola 3 shrnuje dosavadní poznatky o kontraktech a popisuje analyzátor, který je momentálně součástí prostředí ANaConDA. Kapitola 4 se zabývá možnostmi analýzy parametrických kontraktů a v kapitole 5 je popsána implementace a testování nového analyzátoru pro parametrické kontrakty.

[1] http://www.fit.vutbr.cz/research/groups/verifit

Kapitola 2

Vícevláknové programování

Typické programy napsané v imperativních jazycích jsou tvořeny sekvencí příkazů. Vzájemně nezávislé sekvence poté tvoří procesy či vlákna. Proces často reprezentuje jeden běžící program, má vlastní adresový prostor a je označen jednoznačným identifikátorem. V jednom procesu může běžet několik vláken. Vlákno je tzv. odlehčený proces (*lightweight process*). Všechna vlákna jednoho procesu sdílí paměťový prostor (tedy kód i data). Vícevláknové programování je také méně náročné na režii a přepínání kontextu je rychlejší než v případě procesů.

2.1 Synchronizace vláken

Důležitou součástí paralelních programů je synchronizace jednotlivých vláken. Obvykle se synchronizační mechanismy využívají pro zaručení výlučného přístupu ke sdíleným zdrojům nebo pro synchronizaci jednotlivých akcí, které vlákna vykonávají [21].

Operační systémy poskytují základní synchronizační primitiva, která lze prostřednictvím programovacích jazyků využívat na vyšší úrovni. Mezi synchronizační mechanismy patří:

- Semafory –Semafory lze použít pro zasílání synchronizačního signálu mezi vlákny. POSIX semafory jsou implementovány jako celočíselné proměnné, nad nimiž je možné provádět akci *lock*, resp. *unlock*, která dekrementuje, resp. inkrementuje hodnotu semaforu. Kromě synchronizačního signálu lze semafory využít i pro omezení přístupu vláken do kritické sekce, přičemž inicializační hodnota semaforu udává maximální počet vláken současně přítomných v kritické sekci.

- Mutexy – Mutexy jsou binární semafory, které zajišťují výlučný přístup vláken do kritické sekce. Jedná se o semafor inicializovaný na hodnotu 1.

- Bariéry – Exekuce programu může pokračovat za bariéru až poté, co se všechna vlákna nebo procesy na bariéře zastaví.

- RCU (read-copy-update) – Synchronizační technika poskytovaná jádrem operačního systému, která umožňuje současný přístup vláken do kritické sekce (pro čtení i zápis), čímž odstraňuje problém serializace kritické sekce vznikající u semaforů [2]. Princip synchronizace a implementace popsali McKenney a Slingwine [15].

- Monitory – Monitory patří mezi vysokoúrovňové synchronizační mechanismy a umožňují vláknům čekat na splnění určité podmínky bez nutnosti blokovat kritickou sekci.

Ve standardní knihovně jazyka C++ lze nalézt implementaci mutexů `std::mutex` a monitorů `std::condition_variable`. Celou řadu synchronizačních mechanismů poskytuje také knihovna **pthreads**[1].

2.2 Chyby v paralelních programech

Paralelní programování s sebou přineslo také řadu nových problémů, mezi které patří speciální paralelní chyby a nedeterminismus exekuce programu, který znesnadňuje jejich odhalení. V této a následující sekci budou popsány principy vzniku nejznámějších chyb a možnosti, jak je lze v programech vyhledávat.

2.2.1 Souběh

Souběh neboli *data race* vzniká v případě dvou nesynchronizovaných přístupů ke sdílené proměnné, pokud je alespoň jedna z operací zápis [5]. Nedeterminismus při přepínání jednotlivých vláken může způsobit, že proměnná v různých běžích programu může nabýt až několika různých hodnot podle aktuálního pořadí vláken a jejich operací. Jedná se o velmi častou chybu a pro její odhalení byly navrženy a implementovány statické i dynamické techniky. Framework ANaConDA disponuje několika analyzátory zaměřenými na detekci chyb souběhu a budou blíže popsány v sekci 2.4.

2.2.2 Porušení atomicity

Porušení atomicity neboli *atomicity violation* vzniká v případě chybného předpokladu, že úsek kódu je atomický, což může vést k neočekávanému a potažmo i chybnému chování programu. Aby porušení atomicity mohlo být detekováno, musí existovat popis, které bloky kódu mají být vykonány atomicky, což může být poměrně problematické. Proto se algoritmy pro detekci této chyby často zabývají pouze atomicitou u operací nad proměnnou.

Příkladem komplexnějšího zápisu požadavků na atomicitu jsou kontrakty, které vyžadují atomické vykonání určitých metod nad objektem. Konkrétněji se kontraktům věnuje kapitola 3.

2.2.3 Nesprávné pořadí

K chybě zvané nesprávné pořadí neboli *order violation* dochází v situacích, kdy se jednotlivé instrukce programu vykonají v jiném než očekávaném pořadí. I zde se naráží na problém, jak specifikovat, zda k chybě došlo.

Chyba nesprávné pořadí se může projevovat například při práci se soubory či sockety, pokud jedno vlákno sdílený prostředek uzavře dříve, než do něj jiné stihne zapsat. Pro tento konkrétní případ je v prostředí ANaConDA analyzátor, ale vzhledem k charakteru chyby je velmi specifický a nelze ho použít na obecnou detekci nesprávného pořadí.

2.2.4 Uváznutí

Je-li S množina všech vláken v programu, pak k uváznutí (*deadlock*) dochází tehdy, když jsou všechna vlákna z této množiny S zablokována a čekají na událost, která by je odblokovala, ale tu může vygenerovat pouze vlákno z této množiny S. Pro vznik uváznutí musí být splněny 4 tzv. Coffmanovy podmínky:

[1] https://computing.llnl.gov/tutorials/pthreads/

1. Zajištění výlučného přístupu vláken (procesů) ke sdíleným prostředkům.
2. Vlákna vlastní alespoň jeden sdílený prostředek, jsou pozastavena a čekají na další.
3. Držené prostředky může vrátit pouze vlákno, které je vlastní, a to až poté, co dokončí jejich využití.
4. Čekající vlákna jsou na sobě cyklicky závislá.

Pro odhalení deadlocku existuje algoritmus Goodlock [11], který je implementován v prostředí ANaConDA.

2.3 Možnosti odhalování chyb ve vícevláknových programech

Obecně lze chyby v programech odhalovat pomocí testování nebo komplexnější analýzy, což platí i pro vícevláknové programy. Testování je založeno na systematickém nebo náhodném spouštění analyzovaného programu za účelem zjištění, zda program porušuje specifikaci. Jedná se o víceúčelovou a pro velké aplikace někdy i jedinou metodu, která pomáhá zlepšit kvalitu programu. Pro odhalení paralelních chyb jsou však klasické metody testování nedostatečné. Jednak chyba, která se v programu nachází, nemusí vést k selhání a pokud vede, tak může být velmi složité defekt najít, jelikož výsledkem testu je pouze informace o tom, zda prošel nebo neprošel.

Další nevýhodou testování je, že analyzuje pouze jedno konkrétní spuštění paralelního a tudíž nedeterministického programu. Ani opakované spuštění testu nemůže zaručit, že dojde k jinému proložení vláken, protože to může být často ovlivněno jak jinými běžícími procesy, tak i architekturou a operačním systémem, na kterém program běží.

Analýza sbírá informace o běhu programu, na základě kterých může podávat komplexnější závěry, např. zda došlo k paralelní chybě. Lze ji provádět staticky nebo dynamicky.

2.3.1 Statická analýza

Při statické analýze se informace o programu získávají ze zdrojového kódu a samotný program se nevykonává. Tento přístup zahrnuje metody jako manuální inspekci zdrojového kódu, automatické syntaktické a sémantické kontroly, které provádí překladače programovacích jazyků, ale i formální techniky jako kontrolu modelem (angl. *model-checking*), analýzu datových toků (angl. *data-flow*), symbolickou exekuci a abstraktní interpretaci [20].

Extrakcí informací přímo ze zdrojového kódu je možné najednou analyzovat všechny exekuce programu, které by teoreticky mohly nastat. U vícevláknových programů může být počet všech možných cest programu tak obrovský, že způsobí nepoužitelnost algoritmu pro jejich analýzu. Proto existují snahy o redukci toho počtu zaváděním různých aproximací, které s sebou ale nesou riziko falešných chybových hlášení (*false positive*) [18].

Za účelem detekce paralelních chyb se navrhují speciální algoritmy, které se poté implementují jako statické či dynamické analyzátory a zaměřují se na odhalení často jedné konkrétní chyby.

2.3.2 Dynamická analýza

Dynamická analýza získává informace o programu na základě provádění kódu. Stejně jako při testování je tedy možné analyzovat pouze jeden běh programu, ale získané závěry jsou komplexnější a přesnější. Statická analýza může oproti dynamické výrazněji napomoci při

lokalizaci defektu, výhodou dynamické analýzy je zase nezávislost na implementačních detailech a možnost analýzy bez znalosti zdrojového kódu.

Pro fungování dynamické analýzy je potřeba zajistit sběr dat o běžícím programu a algoritmus, který data analyzuje a podává informace o detekovaných chybách. Sběr dat zajišťují sondy pomocí instrumentace. Tu je možné provádět na různých úrovních (instrumentace zdrojového či binárního kódu, platformy) a může být statická nebo dynamická. Statická instrumentace probíhá před spuštěním kódu a dochází ke změně původních binárních souborů. Dynamická instrumentace vkládá do programu kód potřebný k monitorování až za běhu, takže se původní binární soubory nemění. Je sice pomalejší, ale analýza nebrání možnosti využívat monitorovaný program běžným způsobem, a navíc umožňuje monitorování i dynamicky generovaného kódu [8].

Dynamickou analýzu lze doplnit ještě o další metody, které mají za úkol zvýšit pravděpodobnost výskytu a odhalení chyby – extrapolace a vkládání šumu.

Extrapolace umožňuje vyvozovat z dostupných dat závěry o situacích, které v aktuálním plánu přepínání kontextu nezpůsobily chybu, ale v jiném podobném by mohly. Pokud například ve dvou vláknech nastanou nesynchronizované události, které momentálně nepovedou k chybě, analyzátor bude zvažovat, zda by mohlo dojít k přepnutí kontextu tak, aby se operace vykonaly v jiném pořadí a tím pádem chybu způsobily. Extrapolace je záležitostí algoritmů pro detekci chyb, v kapitole 3 bude vysvětleno, jak ji využívá algoritmus pro analýzu kontraktů.

Vkládání šumu je technika, která do analyzovaného programu vkládá instrukce ovlivňující přepínání vláken. Cílem je způsobit přepnutí kontextu, ke kterému by za normálního běhu programu nemuselo dojít, a zvýšit tak pravděpodobnost výskytu chyb. Paralelní chyby, které se projeví za normálního běhu, lze odhalit i testováním při vývoji programu, šum by měl pomoci odhalit chyby, ke kterým by mohlo docházet jenom výjimečně.

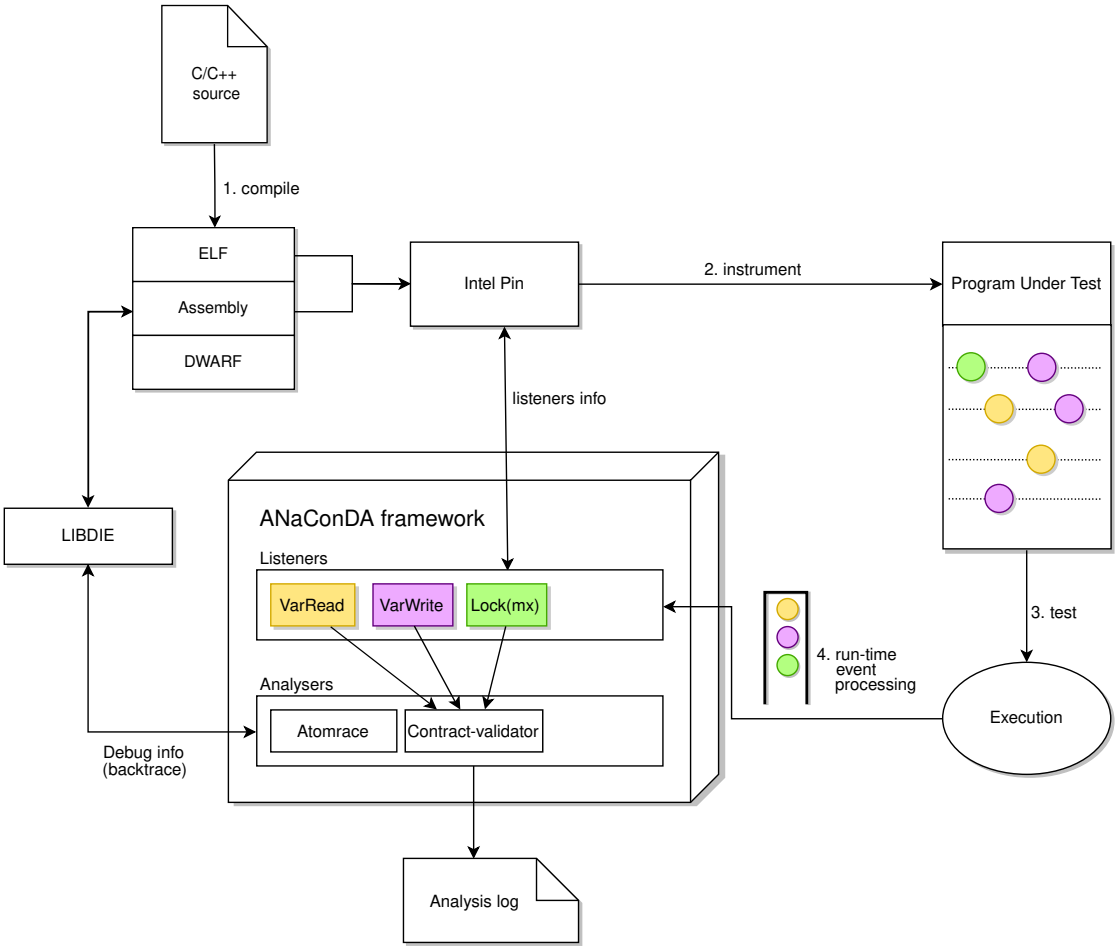
Při vkládání šumu je potřeba nastavit jeho parametry – typ, frekvenci a sílu. Typy šumu lze rozdělit na dvě hlavní kategorie. Těmi jsou uspání vlákna `sleep` a systémové volání `yield`, pomocí kterého se vlákno vzdá procesoru. Důležitou a problematickou částí je rozhodnutí, na která místa v programu by bylo vhodné šum vložit. Pokud bude šumu hodně, vykonávání programu se výrazně zpomalí. Také se může stát, že se chování programu nezmění, protože se šumy vzájemně vyruší. Nastavení parametrů proto často vyžaduje dlouhé ladění. Typicky je ale vhodné umístit šum před obsluhu vláken, synchronizační funkce nebo přístupy do paměti [8].

2.4 Framework ANaConDA

Přestože je dynamická analýza při testování paralelních programů úspěšná, jedná se o složitý proces, který není možné zahrnout jak běžnou součást vývoje programů. Obtížné může být zejména monitorování běhu programu, a proto se vyvíjí nástroje, které by tuto funkcionalitu poskytovaly vývojářům a umožnily jim soustředit se na tvorbu nových analyzátorů. Jedním z takových prostředí je ANaConDA.

ANaConDA[2] (Adaptable Native-code Concurrency-focused Dynamic Analysis) je open-source framework pro dynamickou analýzu vícevláknových C/C++ programů na binární úrovni [7]. Poskytuje podporu pro tvorbu dynamických analyzátorů a zajišťuje instrumentaci

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



Obrázek 2.1: Schéma prostředí ANaConDA. Binární program, který vznikl překladem zdrojového souboru v jazyce C/C++, je podroben dynamické instrumentaci pomocí nástroje Intel Pin. O běhu programu a událostech, které nastaly, je pak informováno jádro prostředí ANaConDA. Tyto informace jsou dále předávány analyzátorům, které je využívají pro detekci paralelních chyb.

programu a získávání informací o důležitých událostech, které se vyskytly za jeho běhu. Podporuje techniku vkládání šumu a umožňuje specifikovat jeho parametry.

Motivací pro vznik prostředí byla především nedostatečná nabídka podobných nástrojů, přestože pro jazyk Java již prostředí podporující dynamickou analýzu existují (ConTest [4], RoadRunner [10]). Důvodem je zřejmě nutnost analyzovat C/C++ programy na binární úrovni.

Prostředí ANaConDA, jehož schéma se nachází na obrázku 2.1, je tvořeno ze tří hlavních částí – PIN, framework, analyzátory. PIN slouží k dynamické instrumentaci binárního kódu a monitoruje běh programu. Při důležitých událostech informuje jádro prostředí ANaConDA, které získané informace dále poskytuje analyzátorům. Analyzátory pracují na vyšší úrovni a většinou se jedná o implementace algoritmů pro detekci paralelních chyb. Mají k dispozici API funkce, které by vývojářům měly ulehčit nové implementace, a proto se neustále pracuje na jejich rozšiřování.

2.4.1 Nástroj Pin

Nástroj Intel Pin[3] slouží k dynamické instrumentaci binárního programu a umožňuje tvorbu nástrojů pro dynamickou analýzu. ANaConDA je na tomto nástroji postavená a díky němu získává informace o běhu programu.

Za podobným účelem bylo možné využít nástroj Valgrind [17], ale ten je oproti Pinu dostupný pouze pro platformu Linux (Pin je podporován na Linux, Windows i OS X), a navíc serializuje běh programu.

Pin disponuje bohatým API, které umožňuje získávat nejen informace o tom, že daná událost v programu nastala, ale poskytuje také další důležité kontextové informace, jako je obsah registrů a podobně. Díky tomu je v prostředí ANaConDA možné monitorovat také volání konkrétních funkcí a sledovat hodnoty jejich parametrů, což je pro tuto práci klíčové.

Princip nástroje Pin

Pin funguje podobně jako JIT (*just in time*) překladače, které překládají *byte code* do strojového kódu za běhu programu (např. Java). Vstupem pro nástroj Pin ovšem není *byte code*, ale spustitelný soubor. Pin provede první instrukci programu a poté vygeneruje vlastní kód, který vloží do aktuální sekvence v monitorovaném programu. Spustí exekuci programu a po skončení sekvence díky vloženému kód opět získá řízení. Uživatelé nástroje Pin také mohou do generovaného kódu vkládat vlastní kód. Díky tomu je možné zavolat konkrétní funkci analyzátoru například při přístupu do paměti v monitorovaném programu. Nástroj upravující kód, který generuje Pin, se nazývá *pintool*.

2.4.2 Jádro prostředí ANaConDA

Nový analyzátor by bylo samozřejmě možné vytvořit přímo jako *pintool* bez využití prostředí ANaConDA, ale pro komplexnější analýzu by to bylo velmi složité, protože API nástroje Pin je poměrně nízkoúrovňové. ANaConDA poskytuje rozhraní na vyšším stupni abstrakce, implementuje funkce, které jsou pro analýzu často potřebné, a poskytuje sadu nástrojů, které lze v analyzátorech snadno využít bez nutnosti vlastní implementace (např. načítání konfiguračních souborů, získání informací o průběhu stopy neboli *backtrace*). Navíc umožňuje vkládání šumu.

Pro to, aby mohl být mezi instrukce monitorovaného programu vložen uživatelský kód, se využívají tzv. zpětná volání (*callback*). Vývojář analyzátoru pomocí registračních funkcí z API prostředí ANaConDA registruje zpětná volání pro události, o kterých chce být informován (vytvoření nového vlákna, volání funkce, získání nebo uvolnění zámku, přístup do paměti atd.). Registrační funkce poté podobně registrují zpětná volání pomocí API nástroje Pin. Dojde-li v programu k monitorované události, zavolá se požadované zpětné volání. To může být buďto přímo funkce analyzátoru, nebo může dojít k předzpracování získaných informací na úrovni prostředí ANaConDA (např. zpracování argumentů a návratových hodnot). V obsluze zpětných volání potom analyzátor provádí potřebné úkony pro odhalení chyb.

[3] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

2.4.3 Základní sada analyzátorů v prostředí ANaConDA

Sada analyzátorů, které jsou v prostředí ANaConDA k dispozici, se stále rozšiřuje. Aktuálně je možné pro analýzu využít:

- AtomRace – Implementace algoritmu AtomRace [14] pro detekci chyby souběhu.
- Contract-validator – Analýza kontraktů bez podpory parametrů, viz 3.3.
- Event-printer – Nedetekuje žádné chyby, pouze na standardní výstup hlásí informace o událostech, které v monitorovaném programu nastaly.
- GoodLock – Implementace algoritmu Goodlock [11] pro detekci uváznutí.
- HLDR-detector – Analyzátor slouží k detekci vysokoúrovňového souběhu.
- TX-monitor – Analýza použití transakčních pamětí.

V letošním roce ještě přibyly implementace algoritmů Eraser [19] a FastTrack [9], které taktéž slouží k detekci souběhu.

Kapitola 3

Kontrakty pro paralelismus a jejich analýza

Kontrakt pro paralelismus je protokol, který vyjadřuje, které metody z veřejného rozhraní modulu je nutné vykonat atomicky, pokud operují nad stejným objektem. O programu, který odpovídá kontraktu, lze prohlásit, že neporušuje atomicitu [3].

Formálně je kontrakt pro paralelismus množina klauzulí $\mathbb{R}$, přičemž platí, že každá klauzule $\varrho \in \mathbb{R}$ je regulárním výrazem nad množinou názvů všech veřejných metod modulu $\Sigma_{\mathbb{M}}$. Jestliže je libovolná sekvence, pro níž existuje reprezentace klauzulí $\varrho \in \mathbb{R}$, proložena voláním libovolné metody z množiny $\Sigma_{\mathbb{M}}$ nad stejným objektem, dochází k porušení kontraktu (*contract violation*).

3.1 Použití základních kontraktů

Význam a použití kontraktů si lze snadno představit na kontejnerech, proto bude pro následující příklady využita pseudoimplementace kontejneru zobrazená v ukázce 3.1. Pro demonstraci kontraktů postačuje znát veřejné rozhraní modulu (zde třídy).

```
template <typename T>
class Container {
    // private data and methods
public:
    T get(int idx); // Get data on given index.
    void set(int idx, T data); // Save data on given index.
    void rmv(int idx); // Remove data on index.
    int indexOf(T data); // Get index of given data.
    int size(); // Get size of the container.
    bool contains(T data); // Check whether container contains given data.
}
```

Zdrojový kód 3.1: Veřejné rozhraní kontejneru.

Třída `Container` obsahuje základní metody pro manipulaci s daty, která jsou uložena v soukromé struktuře. Pomocí metody `get(idx)` lze přistoupit k datům na daném indexu, metodou `set(idx, data)` lze uložit data na daný index. Metoda `rmv(idx)` odstraní element na daném indexu a zároveň všechny elementy na vyšších indexech posune, `indexOf(data)` vrací index, na kterém se nachází hledaná data, metoda `size()` vrací velikost kontejneru

(počet aktuálně uložených elementů) a konečně metoda `contains(data)` vrací informaci o existenci elementu v kontejneru.

Pro výše uvedený příklad je nyní možné sestavit množinu $\Sigma_{\mathbb{M}}$ obsahující jména veřejných metod, tedy $\Sigma_{\mathbb{M}} = \{$`get`, `set`, `rmv`, `indexOf`, `size`, `contains`$\}$.

Jednotlivé klauzule tvořící kontrakt pro třídu `Container` mohou vypadat takto:

(ϱ_1) `indexOf (get|set|rmv)`
(ϱ_2) `size (get|set|rmv)`
(ϱ_3) `contains indexOf`

Dle klauzule ϱ_1 je nutné, aby sekvence metod pro obdržení indexu pomocí `indexOf` a pro modifikaci dat pomocí `get, set` nebo `rmv` proběhla atomicky. Pokud by jiné vlákno volalo metodu z množiny $\Sigma_{\mathbb{M}}$ nad stejným objektem, mohlo by dojít ke změně v kontejneru a získaný index by už nemusel být platný. Podobné chování vynucuje i klauzule ϱ_2, zde by při vymazání elementu způsobeným jiným vláknem mohlo dojít k přístupu za hranici platných dat. Klauzule ϱ_3 vynucuje atomicitu operací pro ověření existence dat a následné zjištění jejich indexu. Pokud by došlo ke změně nebo vymazání dat po tom, co byla potvrzena jejich existence, metoda `indexOf` by hledala neexistující data.

3.2 Možná rozšíření kontraktů

Základní definice kontraktů může být pro analýzu příliš hrubá, jelikož nedokáže zachytit kontextové rozdíly, které mohou rozhodovat o tom, zda je nalezené porušení kontraktu chyba či nikoli. Velké množství chybových hlášení by mohlo analýze ubírat na popularitě, a proto byla navržena dvě možná rozšíření kontraktů, která budou dále popsána.

3.2.1 Cíle a spoilery

Definice kontraktu říká, že k jeho porušení dochází při volání libovolné metody z množiny $\Sigma_{\mathbb{M}}$. Na příkladu výše je ale patrné, že ačkoli volání některých metod skutečně může atomicitu porušit, jiné metody nad kontejnerem neprovedou žádné změny, které by v programu způsobily chybu. Bude-li sekvence libovolné klauzule kontraktu ze sekce 3.1 proložena voláním metody `size` v jiném vlákně, bude se jednat o porušení kontraktu, přestože se nejedná o operaci, která by v programu způsobila chybu či nekonzistenci. Proto článek [3] rozšiřuje definici kontraktů o tzv. spoilery (angl. originál zde nebude přeložen), které umožní takové situace popsat a detekovat detailněji.

Klauzule kontraktu tak, jak byly popsány doteď, se budou nazývat cíle (z angl. *target*) a budou uváděny ve dvojicích spolu se spoilery. Spoiler reprezentuje sekvenci metod, které mohou cíl porušit. Není tedy nutné, aby sekvence cíle proběhla atomicky vzhledem ke všem metodám veřejného rozhraní modulu, ale její atomicita nesmí být porušena sekvencí spoilerů (opět platí, že všechny metody musí být vykonány nad stejným objektem).

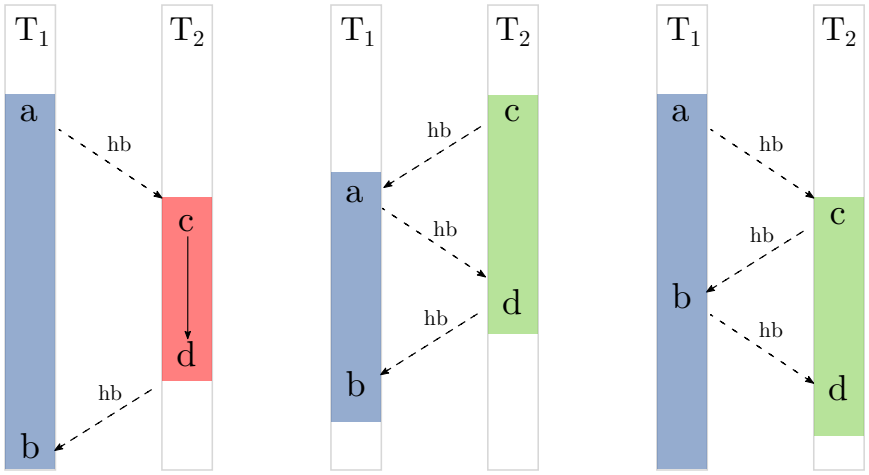
Formálně je nyní $\mathbb{R}$ množina cílů, přičemž každý cíl $\varrho \in \mathbb{R}$ je regulárním výrazem nad množinou názvů všech veřejných metod modulu $\Sigma_{\mathbb{M}}$. $\mathbb{S}$ je množina spoilerů, přičemž každý spoiler $\sigma \in \mathbb{S}$ je regulárním výrazem nad množinou názvů všech veřejných metod modulu $\Sigma_{\mathbb{M}}$. Dále je definována abeceda $\Sigma_{\mathbb{R}} \subseteq \Sigma_{\mathbb{M}}$, resp. $\Sigma_{\mathbb{S}} \subseteq \Sigma_{\mathbb{M}}$, která obsahuje názvy metod vyskytujících se v cílech, resp. spoilerech. Kontrakt je definován jako relace $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, která pro každý cíl definuje spoiler, který může způsobit porušení atomicity. Platí, že cíl může být porušen více spoilery a stejně tak spoiler může porušit více cílů. K porušení kontraktu dochází, pokud je sekvence metod nad objektem o reprezentovaná cílem $\varrho \in \mathbb{R}$

plně proložena vykonáním sekvence metod spoileru $\sigma \in \mathbb{C}(\varrho)$ nad objektem o. K úplnému proložení sekvence cíle ϱ sekvencí spoileru σ dochází právě tehdy, když exekuce ϱ začne dříve než exekuce σ a zároveň exekuce σ skončí dříve než exekuce ϱ. Monitorování částečného proložení se neuvažuje, pokud by to bylo potřeba, je možno sekvenci jednoho spoileru rozdělit na několik jiných sekvencí, což povede ke kýženému výsledku. Tato závislost je definována pomocí tzv. *hb*-relace (bude vysvětleno blíže v sekci 3.3.2). Splnění podmínek je ilustrováno na obrázku 3.1.

S využitím rozšíření definice kontraktů o spoilery by mohly jednotlivé klauzule z předchozího příkladu vypadat takto:

$$
\begin{array}{ll}
(\varrho'_1) & \texttt{indexOf (get|set|rmv)} \leftarrow \texttt{set|rmv} \\
(\varrho'_2) & \texttt{size (get|set|rmv)} \leftarrow \texttt{rmv} \\
(\varrho'_3) & \texttt{contains indexOf} \leftarrow \texttt{set|rmv}
\end{array}
$$

Klauzule ϱ'_1 omezuje spoiler na metody `set` a `rmv`, protože pouze ty mohou v kontejneru způsobit takové změny, že index získaný metodou `indexOf` bude při následné úpravě dat neplatný. Klauzule ϱ'_2 má ve spoileru uvedenou pouze metodu `rmv`, protože ta jediná může vést k přístupu za hranici platných dat. Klauzule ϱ'_3 vyžaduje, aby potvrzení existence dat metodou `contains` a následné vyhledání jejich indexu metodou `indexOf` nebylo proloženo metodami pro úpravu či mazání dat v kontejneru, protože by v okamžiku hledání indexu již data v kontejneru existovat nemusela.



Obrázek 3.1: Ukázka splnění podmínek pro porušení kontraktu `a b` ← `c d`. V případě a) exekuce spoileru plně prokládá exekuci cíle a tudíž dojde k porušení kontraktu. V situaci b), resp. c) exekuce spoileru začala dříve, resp. skončila později než exekuce cíle a tudíž k porušení kontraktu nedošlo.

3.2.2 Parametry

Podobně jako s využitím spoilerů lze definice kontraktů zpřesnit využitím parametrů. Například klauzule `indexOf (get|set|rmv)` ze sekce 3.1 vyžaduje atomicitu pro dané metody vždy, bez ohledu na jejich argumenty. Přesto má smysl ji vyžadovat pouze v případě, kdy metody `get`, `set` a `rmv` budou mít jako argument index získaný metodou `indexOf`.

Článek [3] navrhuje rozšíření specifikace kontraktů o parametry metod a jejich návratové hodnoty pomocí proměnných, aby bylo možné lépe zachytit kontext a tok dat. Značení takového vztahu je blízké specifikaci klauzulí ve funkcionálních jazycích (např. Prolog). Opakované použití proměnné se stejným názvem v jedné klauzuli kontraktu značí, že se hodnoty argumentů či návratových hodnot musí rovnat. Je také možné využít podtržítko ve smyslu anonymní proměnné, jejíž hodnota není důležitá.

Specifikaci kontraktů z předchozího příkladu lze parametry doplnit takto:

(ϱ_1'') `indexOf(X,_) (get(X,_)|set(X,_,_)|rmv(X,_))` $\leftarrow$ `set(X,_,_)|rmv(X,_)`
(ϱ_2'') `size(X) (get(X,_)|set(X,_,_)|rmv(X,_))` $\leftarrow$ `rmv(X,_)`
(ϱ_3'') `contains(X,_) indexOf(X,_)` $\leftarrow$ `set(X,_,_)|rmv(X,_)`

Příklad výše používá parametr X pro označení objektu, nad kterým jsou jednotlivé metody vykonány. Protože se objekt ve většině případu metodě předává jako speciální první argument, je parametr X použit vždy právě na této pozici. Ostatní argumenty metody jsou tedy posunuty o jednu pozici doprava a zde označeny podtržítkem, protože se jejich hodnoty nebudou monitorovat. U uvedeného příkladu je možné definovat také další parametry, které zpřesní výsledky analýzy, konkrétně takto:

(ϱ_1''') `Y=indexOf(X,_) (get(X,Y)|set(X,Y,_)|rmv(X,Y))` $\leftarrow$ `set(X,Y,_)|rmv(X,_)`
(ϱ_2''') `Y=size(X) (get(X,Y)|set(X,Y,_)|rmv(X,Y))` $\leftarrow$ `rmv(X,_)`
(ϱ_3''') `contains(X,Y) indexOf(X,Y)` $\leftarrow$ `set(X,_,_)|rmv(X,_)`

Význam parametru X zůstává stejný, stále se jedná o označení objektu. Dále je ve všech případech použit parametr Y, který označuje závislost mezi argumenty a návratovými hodnotami metod. Klauzule ϱ_1''' pomocí parametru Y vyžaduje, aby získání indexu a následná úprava dat na tom samém získaném indexu proběhla atomicky. Pokud metody `get, set` nebo `rmv` budou upravovat data na jiném indexu, než vrátila metoda `indexOf`, není atomicita kontraktem vyžadována. Aby při porušení kontraktu došlo k chybě, je nutné, aby metoda `set` (uvedená ve spoileru) také změnila data na stejném indexu. U metody `rmv` ve spoileru parametr Y uveden není, protože pokud dojde ke smazání libovolného elementu v kontejneru, dojde k posunu zbývajících dat a chybu tedy může způsobit i mazání dat na jiných indexech. Ostatní klauzule využívají parametr Y obdobným způsobem.

3.3 Základní principy analýzy

V této a následující sekci bude popsán princip dynamické analýzy kontraktů rozšířených o spoilery (ve zbytku textu bude pro zjednodušení používáno pouze označení kontrakt). Metoda neuvažuje parametrické kontrakty, princip analýzy je však téměř shodný a analyzátor s podporou parametrů z něho velmi úzce vychází. Problémy parametrické analýzy a jejich řešení budou popsány v kapitole 4. Pro analýzu kontraktů byla navržena i statická metoda, její popis je ale nad rámec této práce a detaily je možné nalézt v [3].

Na vstupu analyzátoru je vícevláknový program a specifikace kontraktů. Dynamická analýza je problematická zejména z toho důvodu, že analyzuje pouze jeden běh programu (mimo jiné není možné prohlásit, že analyzovaný program neobsahuje porušení kontraktu), ve kterém se hledaná chyba nemusí vyskytnout. Metoda popsaná dále využívá extrapolaci pomocí tzv. *happens-before* relace (dále značena jako *hb*-relace), která zachycuje pořadí

událostí ve stopě programu. Díky ní je možné sledovat synchronizaci mezi vlákny a odhalit porušení kontraktu, ke kterému sice v aktuálním plánu přepínání kontextu nedošlo, ale kvůli chybějící synchronizaci to není vyloučeno pro jiný podobný běh.

Před popisem samotné metody je nutné definovat určité pojmy a jejich značení. Nechť $\mathbb{T}$ je množina vláken, $\mathbb{R}$ množina cílů, $\mathbb{S}$ množina spoilerů, $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ množina kontraktů a $\mathbb{L}$ množina zámků. Pak $\mathbb{E}_t$ značí množinu všech událostí, které mohou být vygenerovány vláknem $t \in \mathbb{T}$ a $\mathbb{E} = \cup_{t \in \mathbb{T}} \mathbb{E}_t$. Mezi události, které vlákno může vygenerovat, patří vstup do metody nebo výstup z metody, získání nebo uvolnění zámku a operace *fork* a *join* pro vytvoření či čekaní na ukončení jiného vlákna.

3.3.1 Instance cílů a spoilerů

Stopa (*trace*) $\tau = e_1 \cdots e_n \in \mathbb{E}^+$ je definována jako posloupnost událostí vygenerovaných libovolným vláknem z množiny $\mathbb{T}$ a *start(t)/end(t)* značí první/poslední událost vygenerovanou vláknem t. Část této posloupnosti $r = e_{i_1} e_{i_2} \cdots e_{i_k}$, kde $1 < k \leq n$, je instancí cíle $\varrho \in \mathbb{R}$ právě tehdy, když:

- jednotlivé události, ze kterých se r skládá, představují dvojice vstup/výstup do/z metody vykonané stejným vláknem $t \in \mathbb{T}$,
- r odpovídá regulárnímu výrazu cíle ϱ (uvažují se pouze události vstupu) a
- mezi jednotlivými událostmi $e_{i_1} e_{i_2} \cdots e_{i_k}$ nebyla vláknem t vykonána žádná jiná událost z abecedy cíle ϱ.

Podobně je definována také instance s spoileru $\sigma \in \mathbb{S}$ ve stopě τ. Bude-li například specifikován cíl $\varrho = abc$, spoiler $\sigma = def$ a stopa $\tau = aabdcdeef$, pak lze ve stopě na indexech 2 až 5 (tedy události $abdc$) nalézt instanci cíle, přestože je volání metod a, b a c proloženo voláním metody d (takový případ je v pořádku, jelikož metoda d nepatří do abecedy cíle). Instance spoileru se ale v dané stopě nenachází žádná, protože nelze nalézt posloupnost def, která by nebyla narušena voláním metod z abecedy spoileru σ.

První událost instance r je definována jako $start(r) = e_{i_1}$, poslední jako $end(r) = e_{i_k}$. $[\varrho]^\tau$ značí množinu všech instancí cíle $\varrho \in \mathbb{R}$ ve stopě τ. $[\mathbb{R}]^\tau = \cup_{\varrho \in \mathbb{R}} [\varrho]^\tau$ značí množinu všech instancí všech cílů z množiny $\mathbb{R}$ ve stopě τ. Podobně $[\sigma]^\tau$ značí množinu všech instancí spoileru $\sigma \in \mathbb{S}$ ve stopě τ a $[\mathbb{S}]^\tau = \cup_{\sigma \in \mathbb{S}} [\sigma]^\tau$ značí množinu všech instancí všech spoilerů z množiny $\mathbb{S}$ ve stopě τ.

3.3.2 *hb*-relace

Pro detekci porušení kontraktu je nezbytné mít informace o pořadí jednotlivých událostí, které mohou být vykonány různými vlákny. Za tímto účelem se využívá *hb*-relace, která určuje uspořádání dvou událostí s ohledem na teoreticky dosažitelné plánování přepnutí kontextu na základě vybraných synchronizačních primitiv. Nejedná se o speciální metodu navrženou pro analýzu kontraktů, ale využívá se i v jiných analyzátorech, např. FastTrack pro detekci chyb souběhu nad proměnnou [9], a její definice vychází z [13] již z roku 1978.

hb-relace $\prec_{hb}$ nad stopou $\tau = e_1 \cdots e_n \in \mathbb{E}^+$ je definována jako nejmenší tranzitivně uzavřená relace nad množinou událostí $\{e_1, \cdots, e_n\}$ stopy τ. Dvě události e_j a e_k jsou v *hb*-relaci $e_j \prec_{hb} e_k$ vždy, když platí $j < k$ a zároveň je splněna alespoň jedna z podmínek:

- Obě události e_j a e_k byly vykonány stejným vláknem. To, že událost e_j předchází události e_k je tedy zaručeno jejich pořadím v programu.

- Obě události e_j a e_k získávají nebo uvolňují stejný zámek.
- Jedna z událostí e_j a e_k je operace *fork* nebo operace *join* provedená vláknem t na vlákno u a druhá událost je provedena ve vláknu u. Události jsou tedy synchronizovány pomocí *fork-join* synchronizace.

Pokud mezi dvěma událostmi neexistuje *hb*-relace, potom jsou považovány za souběžné a mohou být zdrojem paralelních chyb.

Pro potřeby dynamické analýzy je možné definovat porušení kontraktu pomocí *hb*-relace následovně. K porušení kontraktu $(\varrho, \sigma) \in \mathbb{C}$ ve stopě τ dojde právě tehdy, když ve stopě τ existuje instance cíle $r \in [\varrho]^\tau$ a spoileru $s \in [\sigma]^\tau$, pro které platí, že $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$. Uvedená definice koresponduje s tvrzením uvedeným v sekci 3.2.1, které říká, že exekuce cíle musí začít dříve a skončit později než exekuce spoileru.

3.3.3 Vektorové hodiny

Pro realizaci *hb*-relace se využívají tzv. vektorové hodiny (z angl. *vector clock*). Možností, jak je implementovat, existuje několik, ale metoda pro analýzu kontraktů je založena na implementaci využité v algoritmu FastTrack [9]. Zde je samozřejmě kladen důraz na odhalení chyb při přístupu ke sdílené proměnné a daná implementace zohledňuje operace, které pro kontrakty nejsou podstatné (např. zápis do proměnné). Zde budou popsány vlastnosti a operace potřebné pro analýzu kontraktů.

Vektorové hodiny $VC : \mathbb{T} \to \mathbb{N}$ obsahují informaci o logickém čase pro každé vlákno $t \in \mathbb{T}$. V programovacím jazyce by bylo možné je vyjádřit pomocí pole, které na indexech odpovídajících identifikátorům jednotlivých vláken obsahuje číslo značící logický čas synchronizačních událostí. Nad vektorovými hodinami je definováno částečné uspořádání ($\sqsubseteq$), operace *join* pro sloučení dvou vektorů ($\sqcup$), minimální prvek ($\bot_V$) a pomocná funkce pro inkrementaci t-komponenty vektoru (inc_t). Formálně jsou jednotlivé operace definovány takto:

$$V_1 \sqsubseteq V_2 \iff \forall t.\, V_1(t) \leq V_2(t) \tag{3.1}$$

$$V_1 \sqcup V_2 = \lambda t.\, max(V_1(t), V_2(t)) \tag{3.2}$$

$$\bot_V = \lambda t.\, 0 \tag{3.3}$$

$$inc_t(V) = \lambda u.\, \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u) \tag{3.4}$$

Jednoduše řečeno, zápis $V_1 \sqsubseteq V_2$ říká, že události zachycené ve vektoru V_2 nastaly později než události zachycené ve vektoru V_1. Při spojení dvou vektorů se vždy ze stejných indexů vybere větší hodnota. Inicializační hodnota, na kterou jsou všechny vektory nastaveny, je 0. A konečně při inkrementaci se o jedna zvýší hodnota pouze pro vlákno t.

Při analýze kontraktů se pro každé vlákno $t \in \mathbb{T}$ udržují vektorové hodiny $\mathbb{C}_t$ a pro každý zámek $l \in \mathbb{L}$ vektorové hodiny $\mathbb{L}_l$. Vektorové hodiny pro jednotlivá vlákna $\mathbb{C}_t$ udržují na pozici t informaci o aktuálním logickém čase daného vlákna a na pozicích pro ostatní vlákna $u \in \mathbb{T}$ se nachází záznam o čase, ve kterém byla daným vláknem provedena poslední operace, která se stala před aktuální operací vlákna t.

Inkrementace logického času a aktualizace vektorových hodin

Inkrementace logického času se neděje při každé události, která se v programu vyskytne, ale pouze při synchronizaci pomocí zámků nebo operací *fork* a *join*, a to dle určitých pravidel.

Pokud budou dodána potřebná pravidla, je možné monitorovat v podstatě libovolnou uživatelskou synchronizaci. Pro potřeby kontraktů jsou momentálně implementována pravidla vycházející z [9]:

- Při **získání zámku** m vláknem t se vektorové hodiny vlákna nahradí výsledkem operace spojení nad nimi a vektorovými hodinami zámku. Úspěšné získání zámku způsobí synchronizaci mezi všemi vlákny, které s daným zámkem někdy operovaly, a proto je bezpečné prohlásit, že všechny události, které dosud nastaly, splňují podmínky pro *hb*-relaci. Operací spojení se vektorové hodiny vlákna aktualizují tak, že obsahují časy událostí, které se prokazatelně staly před získáním zámku.

$$\mathbb{C}'_t = \mathbb{C}_t \sqcup \mathbb{L}_m \tag{3.5}$$

- Při **uvolnění zámku** m vláknem t se aktualizují vektorové hodiny zámku na aktuální vektorové hodiny vlákna, takže bude obsahovat důležité informace o synchronizaci pro případná další vlákna, která budou se zámkem operovat. Logický čas vlákna t se inkrementuje.

$$\begin{aligned} \mathbb{L}'_m &= \mathbb{C}_t \\ \mathbb{C}'_t &= inc_t(\mathbb{C}_t) \end{aligned} \tag{3.6}$$

- Při **operaci fork**, kdy se z vlákna t vytvoří nové vlákno u, se pomocí operace spojení nastaví vektorové hodiny vlákna u a poté se inkrementuje logický čas vlákna t. Po inicializaci vektorových hodin vlákna u jsou všechny komponenty nastaveny na hodnotu 0 kromě komponenty u, která má počáteční hodnotu 1. Při operaci spojení si tedy vlákno u přebere informace o všech ostatních vláknech podle hodin vlákna t, ale hodnota $\mathbb{C}_t(u)$ je nulová, tudíž $\mathbb{C}_u(u)$ zůstane nezměněna.

$$\begin{aligned} \mathbb{C}'_u &= \mathbb{C}_u \sqcup \mathbb{C}_t \\ \mathbb{C}'_t &= inc_t(\mathbb{C}_t) \end{aligned} \tag{3.7}$$

- Při **operaci join**, kdy vlákno t ve své exekuci nepokračuje, dokud není ukončeno vlákno u, se pomocí operace spojení aktualizují vektorové hodiny vlákna t a inkrementuje se logický čas vlákna u.

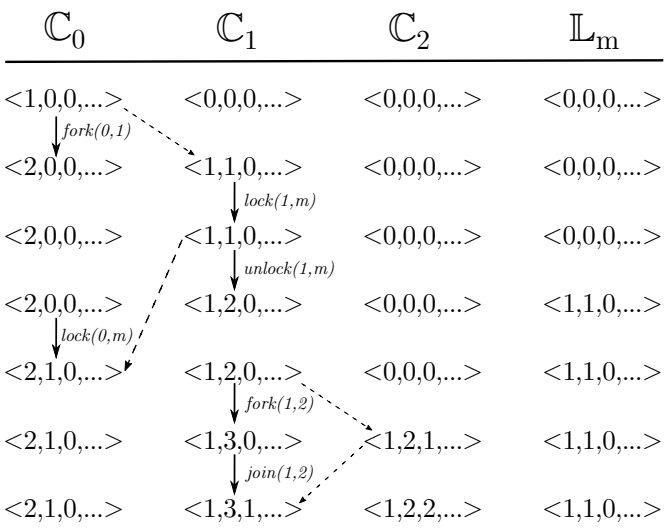
$$\begin{aligned} \mathbb{C}'_t &= \mathbb{C}_t \sqcup \mathbb{C}_u \\ \mathbb{C}'_u &= inc_u(\mathbb{C}_u) \end{aligned} \tag{3.8}$$

Na obrázku 3.2 jsou zobrazeny všechny jmenované operace a jejich vliv na vektorové hodiny vláken i zámku m. Přerušovaná šipka značí *hb*-relaci, která při synchronizačních událostech vzniká. *hb*-relace existuje také mezi vlákny T_0 a T_2 (pro přehlednost není v obrázku zaznačena), přestože mezi nimi nedošlo k přímé synchronizaci, ale byla zprostředkována vláknem T_1. Síla vektorových hodin spočívá v tom, že zachycují i tyto zprostředkované relace.

Využití vektorových hodin pro validaci kontraktů

Při dynamické analýze kontraktů se v paměti udržují informace o vektorových hodinách tak, jak bylo definováno v předchozí sekci:

- Pro každé vlákno $t \in \mathbb{T}$ existují vektorové hodiny $\mathbb{C}_t$.



Obrázek 3.2: Vliv synchronizačních operací na vektorové hodiny.

- Jednotlivé záznamy $\mathbb{C}_t(u)$ pro každé vlákno $u \in \mathbb{T}$ uchovávají informaci o poslední operaci v u, která se stala (*happens before*) před aktuální operací ve vlákně t.
- Pro každý zámek $l \in \mathbb{L}$ existují vektorové hodiny $\mathbb{L}_l$.
- Pro aktualizaci vektorových hodin platí pravidla definovaná dříve.

Navíc se pro každou událost $e \in \tau$ vykonanou vláknem $t \in \mathbb{T}$ uchovávají vektorové hodiny VC_e. Tyto hodiny zachycují stav $\mathbb{C}_t$ v okamžiku, kdy v monitorovaném programu nastala událost e. Pomocí nich lze potom určit, zda událost e_t vykonaná vláknem t nastala před událostí e_u vykonanou vláknem u, tedy zda jsou tyto dvě události v *hb*-relaci. Platí, že $e_t \prec_{hb} e_u$ pokud $VC_{e_t}(t) \leq VC_{e_u}(t)$. Jinými slovy se ověří, že logický čas vlákna t v momentě, kdy nastala událost e_t, je menší nebo roven logickému času poslední události vlákna t, která je v *hb*-relaci s událostí e_u.

Vektorové hodiny jsou velmi důležitou částí analýzy kontraktů, jelikož poskytují informace o synchronizaci vláken. Samotný algoritmus dynamické analýzy neprodukuje falešná chybová hlášení, pokud ovšem vektorové hodiny správně reflektují, co se děje v monitorovaném programu. Pokud program používá jiné mechanismy synchronizace, než byly popsány, je nutné tomu uzpůsobit i implementaci vektorových hodin.

3.4 Validace kontraktů za běhu programu

Článek [3] popisuje metodu, kterou je možno použít pro dynamickou analýzu kontraktů bez znalosti celé stopy programu. Ačkoli není možné dynamickou analýzou dokázat, že v programu nikdy nedojde k porušení kontraktu, popsaná metoda garantuje jeho odhalení pokud se nachází v analyzované stopě. Teoretický algoritmus pro analýzu byl reálně implementován jako analyzátor Contract-validator v prostředí ANaConDA a tento byl použit jako výchozí framework pro implementaci rozšíření podporující kontrakty s parametry.

3.4.1 Okno stopy

Okno stopy (z angl. *trace window*) umožňuje zachytit pouze určitou část analyzovaného programu, je pohyblivé a nemá fixní velikost. Může obsahovat i celou stopu programu, ale cílem je, aby okno bylo co nejmenší. Metoda zavádí optimalizace, které bez ztráty přesnosti zahazují nepodstatné informace, snižují velikost okna a tím i paměťovou a časovou náročnost celé analýzy.

Formálně je okno v část sekvence stopy τ. $[\varrho]^v$ značí množinu všech instancí cíle $\varrho \in \mathbb{R}$ v okně v a $[\mathbb{R}]^v = \cup_{\varrho \in \mathbb{R}}[\varrho]^v$ množinu všech instancí všech cílů z $\mathbb{R}$ v okně v. Podobně je $[\sigma]^v$ množina všech instancí spoileru $\sigma \in \mathbb{S}$ v okně v a $[\mathbb{S}]^v = \cup_{\sigma \in \mathbb{S}}[\sigma]^v$ je množina všech instancí všech spoilerů z $\mathbb{S}$ v okně v.

Jakmile se objeví v monitorovaném programu událost, je ihned přesunuta do okna v. Pomocí operace $v \to e$ je možno událost e z okna v odstranit. Operace $v \to r$ odstraní z okna v všechny události, které patří do instance cíle $r \in [\mathbb{R}]^v$. Musí ovšem platit, že tyto události nejsou součástí žádné jiné monitorované instance cíle nebo spoileru (ani nedokončené), formálně:

$$\forall e_i \in r : v \to e_i \iff (\forall x \in [\mathbb{R}]^v \cup [\mathbb{S}]^v, x \neq r : e_i \notin x) \wedge$$
$$(\forall x \in [\mathbb{R}]^\tau \cup [\mathbb{S}]^\tau, start(x) \in v \wedge end(x) \notin v : e_i \notin x)$$

Podobně je definována operace $v \to s$, která z okna v odstraní všechny události spoileru $s \in [\mathbb{S}]^v$. Těchto operací se využívá pro redukci velikosti okna.

3.4.2 Odstranění instancí cílů a spoilerů

Článek [3] dokazuje, že při zachování přesnosti analýzy je nutné uchovávat v okně pouze poslední dokončenou instanci cíle či spoileru pro každé vlákno. Je zaručeno, že pokud dojde k porušení kontraktu, pak bude reportováno alespoň jednou pro každý cíl a každý spoiler, kdy k porušení došlo.

Předpokládá se, že události se do okna v přidají ihned poté, co se objeví ve stopě τ. Dalším předpokladem je, že jakmile se v okně v objeví instance r cíle ϱ, tedy platí $start(r) \in v \wedge end(r) \in v$, instance r se ověří na porušení kontraktu vůči všem instancím s všech spoilerů $\sigma \in \mathbb{C}(\varrho)$, se kterými je daný cíl v konfliktu a které jsou aktuálně zachyceny v okně v. Stejně tak musí platit, že instance spoileru se ověří vůči všem instancím cílů, které jsou momentálně v okně, jakmile se tato instance spoileru objeví v okně v.

Odstranění spoileru

Za dodržení daných předpokladů platí, že pro každé vlákno $t \in \mathbb{T}$ a pro každý spoiler $\sigma \in \mathbb{S}$ je potřeba v okně v uchovávat pouze poslední dokončenou instanci spoileru σ ve vláknu t.

Odstranění cíle

Při odstraňování cílů je situace trochu odlišná. Zde je potřeba uchovávat jednu poslední instanci cíle ϱ ve vláknu t stejně jako v případě spoilerů. Ta bude sloužit pro kontrolu porušení vůči spoilerům, jejichž začátek ještě v okně není. Dále je potřeba jedna instance pro každé vlákno, ve kterém je běžící (nedokončená) instance spoileru.

3.4.3 Optimalizace vektorových hodin

V předchozí sekci bylo řečeno, že při analýze kontraktů se uchovávají vektorové hodiny všech událostí, které v programu nastaly. To je ovšem zbytečné a postačuje uchovávat informace pouze o první a poslední události jednotlivých instancí.

Článek [3] nabízí také další optimalizaci vektorových hodin. Jedná se zejména o problém uchovávání instancí cílů pro běžící instance spoilerů. Navrhuje strukturu, díky které není nutné uchovávat informace o začátku a konci takových instancí r ve všech vláknech t. Pro porušení kontraktu je nutné splnit dvě podmínky. Proto je potřeba:

- Pamatovat si všechna vlákna u, ve kterých běží instance s, pro kterou je splněna první podmínka porušení kontraktu: $start(s) \not\prec_{hb} start(r)$. Jenom takové instance má smysl kontrolovat při jejich dokončení. Pokud podmínka splněna není, nemá smysl evidovat výskyt takové instance a kontrolovat druhou podmínku.

- Pamatovat si, kdy skončila exekuce instance r (tedy $VC_{end(r)}(t)$), jelikož tato informace bude potřebná pro kontrolu druhé části podmínky pro porušení kontraktu: $end(r) \not\prec_{hb} end(s)$.

Obě tyto informace lze uchovat pomocí mapování *possible violations* (angl. originál zde nebude přeložen) $PV_t^{\varrho,\sigma} : \mathbb{T} \to \mathbb{N}$ pro vlákno $t \in \mathbb{T}$, cíl $\varrho \in \mathbb{R}$, jehož instancí je r, a spoiler $\sigma \in \mathbb{S}$, jehož instancí je s. Tato struktura pro každé vlákno u, v němž běží instance s splňující první podmínku pro porušení kontraktu, obsahuje na indexu u čas, kdy skončila exekuce instance r, tedy $PV_t^{\varrho,\sigma}(u) = VC_{end(r)}(t)$. Pokud ve vláknu tato instance neexistuje nebo podmínku nesplňuje, záznam $PV_t^{\varrho,\sigma}$ je nastaven na hodnotu 0.

S využitím této optimalizace se při ukončení exekuce instance spoileru s ve vláknu t zkontrolují $PV_u^{\varrho,\sigma}(t)$ pro všechna vlákna u různých od t. Pokud je hodnota různá od 0, znamená to, že pro spoiler již byla splněna první podmínka porušení kontraktu, a provede se kontrola druhé podmínky. Jestliže platí $PV_u^{\varrho,\sigma}(t) \leq VC_{end(s)}(u)$, pak $end(r) \prec_{hb} end(s)$ a k porušení kontraktu nedošlo. V opačném případě je chyba reportována.

3.4.4 Algoritmus analýzy

Algoritmus 1 využívá popsané optimalizace a zahrnuje:

- průběžné odstraňování instancí cílů a spoilerů,
- mapování $PV_t^{\varrho,\sigma}$,
- kontroly porušení kontraktů mezi dokončenými instancemi cílů i spoilerů a
- kontroly částečně uložených instancí cílů vůči nově dokončeným instancím spoilerů.

Většina těchto kontrol se u neparametrického analyzátorů provádí při ukončení exekuce metody (rozdíl u parametrického analyzátoru bude popsán v kapitole 5). Při vstupu do metody se pomocí konečného automatu určuje, kterého cíle či spoileru je daná metoda součástí. Zároveň se zde vytváří nové instance, pokud pro daný cíl/spoiler ještě neexistuje, provádí se přechody v nedokončených automatech, pokud je to možné, a zahazují se existující instance, které nemohou provést přechod, ale daná metoda patří do abecedy cíle/spoileru příslušné instance (dále bude tento krok nazýván jako zneplatnění instance). Při výstupu z metody se provede algoritmus 1.

Nejdříve se zkontroluje (řádky 1 a 18), zda existují dokončené instance cíle či spoileru (konečný automat se nachází v koncovém stavu). Pokud existuje ukončená instance cíle, všechna ostatní vlákna se zkontrolují, zda obsahují dokončené instance spoilerů a případně se provede kontrola na porušení kontraktu (řádek 3). Pokud v okně existují běžící, ale nedokončené instance spoilerů, zkontroluje se první podmínka porušení kontraktů (řádek 7). Je-li splněno, že instance spoileru započala až po začátku aktuálně ukončené instance cíle, nebude se dít nic, jelikož o této instanci budeme mít uložené veškeré informace (čas začátku i konce), takže není potřeba sestavovat mapování $PV_t^{\varrho,\sigma}$. Pokud ale první podmínka porušení splněna není, je potřeba provést kontrolu vzhledem k předchozí ukončené instanci cíle (řádek 8) a při jejím splnění uložit konec této instance. Teprve poté je možné tuto starší instanci odstranit (řádky 14 a 15). Pokud se v okně nachází dokončená instance spoileru, odstraní se předchozí instance (řádky 19 a 20). Poté se provede kontrola dokončených instancí cílů, které může daný spoiler porušit, v ostatních vláknech (řádek 23). Nakonec se zkontrolují instance cílů, pro které byla splněna první podmínka porušení kontraktu (řádek 26, viz 3.4.3).

Algoritmus 1 Detekce porušení kontraktu při výstupu z metody.

Data: okno v, událost $e \in \mathbb{E}$ vygenerována vláknem $t \in \mathbb{T}$

1: **if** $\exists \varrho \in \mathbb{R}, r \in [\varrho]_t^v : e = end(r)$ **then**
2: **for** $\sigma \in \mathbb{C}(\varrho), u \in \mathbb{T} : u \neq t$ **do**
3: **if** $\exists s \in [\sigma]_u^v : start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ **then**
4: instance r je porušena instancí s;
5: **end if**
6: **if** $\exists s \in [\sigma]_u^\tau : start(s) \in v \wedge end(s) \notin v$ **then**
7: **if** $start(s) \prec_{hb} start(r)$ **then**
8: **if** $\exists r' \in [\varrho]_t^v : r' \neq r \wedge start(s) \not\prec_{hb} start(r')$ **then**
9: $PV_t^{\varrho,\sigma}(u) = VC_{end(r')}(t)$;
10: **end if**
11: **end if**
12: **end if**
13: **end for**
14: **if** $\exists r' \in [\varrho]_t^v : r' \neq r$ **then**
15: $v \to r'$;
16: **end if**
17: **end if**
18: **if** $\exists \sigma \in \mathbb{S}, s \in [\sigma]_t^v : e = end(s)$ **then**
19: **if** $\exists s' \in [\sigma]_t^v : s' \neq s$ **then**
20: $v \to s'$;
21: **end if**
22: **for** $\varrho \in \mathbb{C}(\sigma), u \in \mathbb{T} : u \neq t$ **do**
23: **if** $\exists r \in [\varrho]_u^v : start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ **then**
24: instance r je porušena instancí s;
25: **end if**
26: **if** $PV_u^{\varrho,\sigma}(t) \neq 0 \wedge PV_u^{\varrho,\sigma}(t) \not\leq VC_{end(s)}(u)$ **then**
27: cíl ϱ je porušen instancí s;
28: **end if**
29: **end for**
30: **end if**

Kapitola 4

Návrh parametrického analyzátoru

Analýza kontraktů bez parametrů trpí dvěma problémy. V některých případech může být příliš přísná a odhalovat porušení kontraktu i tam, kde sémanticky kvůli hodnotám parametrů nemá smysl (příklad byl uveden v sekci 3.2.2). Dalším problémem ale může být i zneplatnění probíhající instance, pokud je proložena voláním metody z abecedy daného cíle/spoileru. Oba tyto problémy lze vyřešit s využitím parametrů. V kapitole 3 bylo navrženo rozšíření specifikace kontraktů o spoiler a o parametry. Cílem této práce bylo implementovat analyzátor s podporou cílů, spoilerů, parametrů a jejich omezení a tato kapitola se zabývá jeho návrhem.

Formát specifikace kontraktů

Pro příklady v této kapitole bude použit následující zápis pro specifikaci kontraktů:

```
{ X=fncA() fncB(_,X)  ← bar(Y)}
Y > X
```

Na prvním řádku se nachází specifikace cíle a spoileru včetně parametrů. Druhý řádek je nepovinný a jedná se o tzv. omezení parametrů (*constraint*). Pomocí omezení je možné ještě více zpřesnit výsledky analýzy. K porušení kontraktu je nyní zapotřebí splnit obě podmínky popsané dříve, a navíc musí hodnoty parametrů splňovat daná omezení.

4.1 Rozdíly parametrické analýzy

Článek [3] definuje kontrakty a další pojmy s nimi spojené pro analýzu bez parametrů. Pro parametrické kontrakty podobné definice zatím neexistují. Při návrhu tedy nejdříve bylo zvažováno, jakým způsobem budeme definovat instanci a kdy bude instance zneplatněna výskytem události z abecedy cíle/spoileru. Poté následovaly pokusy navrhnout algoritmus pro analýzu včetně zvážení možných rizik a situací, kdy by daný postup nefungoval. V této kapitole budou popsány tři možnosti pro analýzu parametrických kontraktů:

- Nejobecnější metoda se snahou podporovat veškeré teoretické situace.
- Parametry pouze jako dodatečné omezení, kdy se skutečně jedná o porušení kontraktu a kdy ne.
- Kompromis mezi dvěma předchozími přístupy – Analýza s deterministicky definovanými hodnotami parametrů.

4.2 Metody pro parametrickou analýzu kontraktů

Parametry výrazně ovlivní kroky, které se provádí při volání metody:

- provedení přechodu v konečném automatu u běžících instancí,
- zneplatnění instance při výskytu události z abecedy cíle/spoileru,
- vytvoření nové instance a
- rušení dokončených instancí.

Také přibude nutnost kontrolovat omezení. Algoritmus pro detekci porušení kontraktů se ale v podstatě nezmění, a proto mu při popisu jednotlivých metod nebude věnována pozornost.

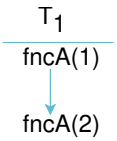
4.2.1 Obecný přístup k parametrické analýze

V kapitole 3 byla instance cíle, resp. spoileru definována pouze na základě názvů metod a v každém okamžiku mohla být v rámci jednoho vlákna pouze jedna běžící instance každého cíle, resp. spoileru. Tato metoda zahrnuje do instance i hodnoty jednotlivých parametrů, což mimo jiné znamená, že pro zneplatnění instance je potřeba, aby nejen název metody patřil do dané abecedy, ale musí se shodovat i hodnoty všech parametrů.

Budeme-li uvažovat specifikaci kontraktu

$$\texttt{\{ fncA(A) fncB(A) } \leftarrow \texttt{ fncC() \}}$$

a stopu vlákna T_1 zobrazenou na obrázku 4.1 můžeme vidět, že jsou ve vláknu dvě běžící instance cíle, které se liší hodnotou parametru `A`. Na dalších příkladech bude ukázáno, proč s sebou tento rozdíl v instancích přináší velký nárůst objemu dat podstatných pro analýzu, což může teoreticky vést i k nepoužitelnosti parametrické analýzy.



Obrázek 4.1: Stopa vlákna zachycující dvě běžící instance cíle.

Provedení přechodu v konečném automatu

Uvažujme specifikaci kontraktu

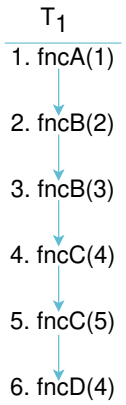
$$\texttt{\{ fncA(A) fncB(B) fncC(C) fncD(C)} \leftarrow \texttt{ fncE() \}}$$

a stopu vlákna T_1 zobrazenou na obrázku 4.2. V této krátké stopě lze vidět 2 dokončené instance cíle:

- `fncA(1) fncB(2) fncC(4) fncD(4)`,
- `fncA(1) fncB(3) fncC(4) fncD(4)`

a 5 běžících instancí:

- `fncA(1)`,
- `fncA(1) fncB(2)`,
- `fncA(1) fncB(3)`,
- `fncA(1) fncB(2) fncC(5)`,
- `fncA(1) fncB(3) fncC(5)`.



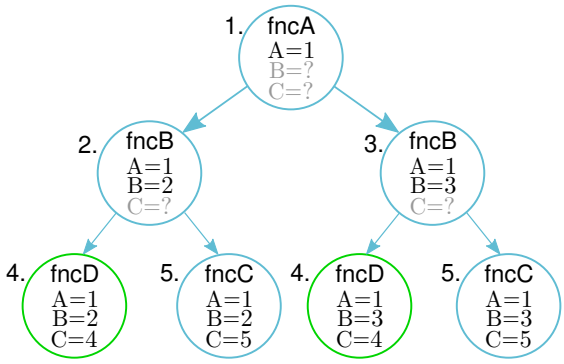
Obrázek 4.2: Stopa vlákna zachycující dvě dokončené a pět běžících instancí cíle.

Z příkladu je patrné, že při úspěšném přechodu v konečném automatu není možné pouze upravit stávající instanci, ale bude potřeba vytvořit její kopii a teprve poté přechod provést. Navíc je potřeba mít uloženy veškeré hodnoty sledovaných parametrů. Přestože se z paměťového hlediska jedná o značnou zátěž v porovnání s analýzou bez parametrů, implementačně je problém poměrně snadno řešitelný využitím hierarchie zachycené ve stromové struktuře. Jeden n-ární strom reprezentuje běžící i dokončené instance konkrétního cíle či spoileru. Uzly reprezentují jednotlivé instance, které se liší hodnotami parametrů. Vždy, když je pro instanci možné určit hodnotu dalšího parametru, vytvoří se pro ni nový synovský uzel s příslušnou hodnotou parametru. Tvorba nových uzlů je potřebná pouze pokud v instanci existuje parametr s nedefinovanou hodnotou. Jakmile jsou známy hodnoty všech parametrů (k tomuto dochází v listových uzlech), může se pouze provádět přechod v konečném automatu. V uvedeném příkladu se pouze přechod bez vytváření nového potomka provede po volání funkce `fncD`. Jednotlivé instance by v paměti mohly být uloženy podobně jako na obrázku 4.3.

Zneplatnění instance

Uvažujme stejný příklad. Pokud v této situaci dojde k volání funkce `fncB(2)` bude potřeba zahodit všechny běžící instance, u kterých se jedná o volání metody ze stejné abecedy. Při využití stromové struktury by se nabízelo zrušit uzel včetně všech jeho potomků, ale takové řešení je chybné. Pokud by se funkce `fncB(2)` volala znovu, došlo by k opětovnému vytvoření uzlu, přestože taková instance není platná. Navíc by mohlo dojít ke zrušení platných dokončených instancí.

Problém lze vyřešit pomocí příznaku, kterým by se označovaly neplatné uzly. Takový přístup ovšem povede k udržování neplatných instancí v paměti, což zvyšuje paměťovou



Obrázek 4.3: Jedna z možností, jak ukládat a klonovat běžící instance. Název uzlu značí stav konečného automatu instance, šedě jsou označeny parametry bez definované hodnoty a zelený kroužek značí dokončenou instanci. U každého uzlu je napsáno číslo korespondující s pořadím události z obrázku 4.2, která vedla k jeho vzniku.

i časovou náročnost analýzy, jelikož se při snaze provést přechod v konečném automat bude procházet velké množství uzlů.

Se zneplatněním instancí se pojí ještě jedna komplikace. Uvažujme kontrakt

$$\texttt{\{ fncA(X) fncB(Y) fncC(Z) fncD(U)} \leftarrow \texttt{fncE() \}}$$

a stopy vláken T_1 a T_2 jako na obrázku 4.4.

T_1	T_2
1. fncA(0)	
	1. fncA(0)
2. fncB(1)	
	2. fncB(1)
3. fncD(4)	
	3. fncD(3)
4. fncC(3)	
	4. fncC(3)
5. fncD(4)	
	5. fncD(4)

Obrázek 4.4: Stopa programu pro demonstraci problematiky zneplatnění instancí.

V obou vláknech se jako 3. událost zavolá funkce `fncD`, která patří do abecedy cíle běžících instancí. Bylo řečeno, že ke zneplatnění dochází pouze tehdy, pokud se rovnají i hodnoty parametrů, což v tomto momentu není možné určit. Zda došlo ke zneplatnění instance se zjistí až při 5. události. Ve vláknu T_1 se skutečně bude muset běžící instance zrušit, kdežto ve vláknu T_2 se instance cíle korektně ukončí. Pro řešení takových situací by bylo zapotřebí udržovat v paměti informace o minulých volání funkcí včetně jejich argumentů, které by teoreticky mohly instanci zneplatnit, což by mohlo způsobit nepoužitelnost analyzátoru na reálné projekty. Jedná se o hlavní důvod, proč tato metoda nakonec nebyla implementována.

Vytvoření nové instance

Vytváření nových instancí by neměl být problém, vytvoří se nový strom nebo uzel v závislosti na konkrétní implementaci. Pouze je potřeba zjistit, zda již funkce nebyla volána se stejným argumentem, protože potom musí samotnému vytváření předcházet zneplatnění již probíhajících instancí.

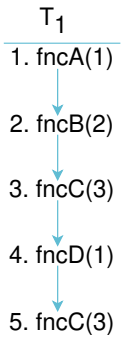
Rušení dokončených instancí

Podstatná optimalizace algoritmu pro analýzu bez parametrů spočívala v zachovávání pouze vektorových hodin první a poslední události dokončené instance. Tato metoda ovšem podobnou optimalizaci využívat nemůže. Pokud by po dokončení instance došlo ke zrušení uzlu a zachování pouze vektorových hodin, mohlo by dojít k následující chybě.

Uvažujme kontrakt

$$\texttt{\{ fncA(X) fncB(Y) fncC(Z) fncD(X)} \leftarrow \texttt{fncE() \}}$$

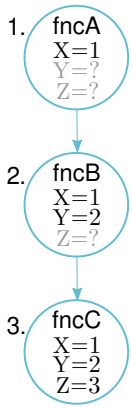
a stopu vlákna T_1 zobrazenou na obrázku 4.5. Na obrázku 4.6 je ukázka stromu instancí po události č. 3. Událost č. 4 ukončí instanci (uzel s názvem `fncC`). Pokud by v tomto momentu došlo k odstranění uzlu ze stromu, událost č. 5 by vedla k vytvoření stejného stromu jako po 3. události, přestože taková instance není platná. Pro vyřešení problému je potřeba udržovat v paměti i informaci o dokončených instancích, včetně hodnot parametrů.



Obrázek 4.5: Stopa vlákna pro demonstraci problému při odstraňování dokončených instancí.

Shrnutí metody

Při snaze sestavit algoritmus pro obecnou analýzu parametrických kontraktů byly nacházeny stále další protipříklady, kdy nebude fungovat. Doposud navržené kroky se zdály paměťové i časově příliš náročné pro využití na reálných projektech a problémů stále přibývalo, navíc bylo těžké najít reálné specifikace kontraktů, kdy by takové problematické kombinace skutečně dávaly smysl. Protože cílem práce bylo implementovat použitelný analyzátor a ne teoretický algoritmus, který nebude možné nasadit na reálné projekty, bylo rozhodnuto obecnou metodu opustit a vydat se směrem, kdy bude upraveno samotné chápání instancí a výsledný algoritmus by mohl být jednodušší a hlavně rychlejší. Tyto metody samozřejmě mají svá omezení a na velmi specifické situace fungovat nebudou. Pokud ale budeme znát reálnou specifikaci takového kontraktu včetně očekávaného chování analyzátoru, bude možné navržené algoritmy dodatečně upravit tak, aby vyhovovaly požadavkům uživatele.



Obrázek 4.6: Ukázka stromu instancí po události č. 3 ze stopy programu na obrázku 4.5 spolu s informací, po kterých událostech byly jednotlivé uzly vytvořeny.

4.2.2 Rozšíření bezparametrické analýzy o parametry

Tato metoda parametry v kontraktech využívá pouze pro omezení hlášení o porušení kontraktu. Zachovává princip pouze jedné běžící instance cíle/spoileru tak, jak tomu bylo u bezparametrické analýzy. Instance jsou zde chápány stejně jako v předchozí metodě, tedy zahrnují nejen název funkce, ale i hodnoty parametrů. K jejich zneplatnění ovšem stačí pouze výskyt události, kdy její název patří do abecedy cíle/spoileru, ale parametry nemusí mít shodné hodnoty.

Uvažujme kontrakt

$$\texttt{\{ fncA(X) fncB() } \leftarrow \texttt{fncC(X) \}}$$

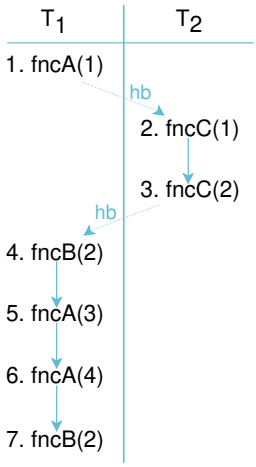
a stopu programu z obrázku 4.7. Daný příklad by vyprodukoval hlášení o porušení kontraktu pro instanci cíle danou událostmi 1 a 4 instancí spoileru danou událostí 2. Událost 3 neporuší kontrakt, jelikož se nerovnají hodnoty parametrů. Událost 5 by vytvořila novou běžící instanci cíle. Událost 6 ji zruší, jelikož se jedná o funkci ze stejné abecedy, ale zároveň vytvoří novou instanci, která je dokončena událostí 7. Instance 1+4 a 6+7 jsou považovány za odlišné a po jejich dokončení budou uloženy nejen vektorové hodiny začátku a konce, ale i hodnoty jejich parametrů.

Provedení přechodu v konečném automatu

Jelikož v jednom vláknu může v jedné chvíli běžet pouze jedna nedokončená instance každého cíle/spoileru, implementace ani algoritmus nemusí být příliš odlišný od bezparametrické analýzy. Přibude zde pouze krok s ověřením rovnosti hodnot argumentů, příp. jiných podmínek, pokud jsou zadána omezení. To, jak se zachovat v případě, kdy je možné přechod z hlediska názvu funkce provést, ale neplatí podmínky omezující hodnoty argumentů, záleží na reálných požadavcích. Buď je možné volání ignorovat nebo běžící instanci invalidovat.

Zneplatnění instance

Pokud v konečném automatu na základě názvu funkce není možné přechod, ale jedná se o součást abecedy cíle/spoileru, dojde ke zneplatnění instance bez ohledu na hodnoty parametrů. Tento krok metodu oproti té předchozí výrazně zjednodušuje.



Obrázek 4.7: Stopa programu obsahující dvě dokončené instance cíle, z nichž jedna byla porušena instancí spoileru.

Shrnutí metody

Implementačně i algoritmicky se analýza nebude příliš lišit od analýzy bez parametrů. Je potřeba přidat dva kroky – kontrolovat, zda parametry splňují specifikovaná omezení (pokud nějaká jsou) a při detekci porušení kontraktu zkontrolovat hodnoty parametrů. Dále může být pro jeden cíl/spoiler více dokončených instancí, které se navzájem liší hodnotami argumentů. Metodu jsme se rozhodli neimplementovat, protože příliš omezuje použití parametrů.

4.2.3 Analýza s předem stanovenými hodnotami parametrů

Tato metoda řeší problémy metody pro obecnou analýzu a zároveň neomezuje využití parametrů až v takovém rozsahu jako druhá metoda. Instance a jejich invalidace jsou chápány stejně jako v první metodě. Rozdíl spočívá v tom, že hodnoty veškerých parametrů použitých v cíli nebo spoileru musí být stanoveny při volání první metody. Tato skutečnost velmi zjednoduší výsledný algoritmus i paměťovou a časovou náročnost analýzy.

Omezení způsobí, že například kontrakt

```
{ X=fncA() Y=fncB(X) fncC(Y)  ← fncD() }
```

nebude možné analyzovat. Příklady tohoto typu lze řešit buďto rozkladem kontraktu na několik klauzulí:

```
{ X=fncA() fncB(X)  ← fncD() }
{ Y=fncB() fncC(Y)  ← fncD() }
```

nebo pomocí výrazů, které jednoznačně určí hodnoty argumentů:

```
{ X=fncA() Y=fncB(X) fncC(Y)  ← fncD() }
Y = X + 1
```

Přestože lze vymyslet teoretické kontrakty, které pomocí této metody nebude možné analyzovat, nepovedlo se nalézt příklad reálného kontraktu, pro který by nebyla použitelná. Pokud by se takový případ vyskytl, bylo by možné na základě konkrétních požadavků na chování analyzátoru metodu upravit nebo navrhnout jinou.

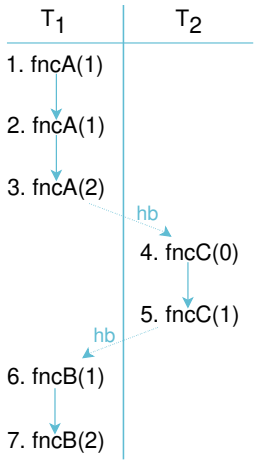
Provedení přechodu v konečném automatu

Díky tomu, že hodnoty všech argumentů jsou stanoveny při volání první metody cíle/spoileru, tedy při vzniku nové instance, odpadá nutnost vytvářet kopie existujících instancí. Je však možné mít několik běžících instancí jednoho cíle/spoileru, které se budou lišit hodnotami argumentů.

Uvažujme kontrakt

```
{fncA(X) fncB(Y) ← fncC(X)}
Y = X + 1
```

a stopu programu z obrázku 4.8. Při události 1 se vytvoří instance r_1 s parametry $X = 1$ a $Y = 2$. Událost 2 je volání metody ze stejné abecedy cíle i se stejnými hodnotami parametrů, takže dojde k invalidaci běžící instance r_1 a založení nové instance r_2. Událost 3 vytvoří novou instancí r_3 s parametry $X = 2$ a $Y = 3$, takže aktuálně existují dvě běžící instance cíle. Události 4 a 5 vytvoří dvě odlišné instance spoileru s_1 a s_2 a uloží se vektorové hodiny i hodnoty parametrů. Při události 6 dojde k pokusu o přechod v konečných automatech instancí r_2 a r_3. V obou případech by na základě názvu metody bylo možné přechod provést, ale hodnota Y je odlišná, a proto se nestane nic. 7. událost však způsobí přechod a dokončení instance r_2. V tomto okamžiku se zkontroluje porušení kontraktu vzhledem k instancím spoileru s_1 a s_2. Podmínky porušení vyplývající z *hb*-relace splňují obě instance, ale hodnota parametru X se rovná pouze u s_2, a proto bude zahlášeno pouze toto jedno porušení kontraktu.



Obrázek 4.8: Stopa programu obsahující jedno porušení kontraktu. Instance cíle tvořená událostmi 2 a 7 je porušena instancí spoileru tvořenou událostí 5.

Zneplatnění instance

Při znalosti hodnot parametrů z principu nemůže docházet k problémům popsaným u první metody, kdy při volání metody z abecedy cíle/spoileru nebylo možné rozhodnout, zda bude instance validní či nikoli.

Rušení dokončených instancí

S dokončenými instancemi lze nakládat podobně jako u bezparametrické analýzy, pouze je potřeba si kromě vektorových hodin pamatovat i hodnoty argumentů.

Shrnutí metody

Tato metoda pro analýzu parametrických kontraktů zavádí omezení na použití parametrů, jejichž hodnoty musí být pro konkrétní instanci stanoveny při volání první metody cíle/spoileru. Omezení výrazně zjednodušuje analýzu parametrických kontraktů při zachování jejích výhod. Umožňuje specifikovat sémantické podmínky, kdy má smysl kontrolovat porušení kontraktu, a zároveň dovoluje mít několik běžících instancí cíle či spoileru, které se vzájemně liší právě hodnotami parametrů. Proto byla tato metoda implementována jako rozšíření prostředí ANaConDA (viz kapitola 5).

4.3 Výsledný algoritmus pro analýzu kontraktů s parametry a jejich omezeními

Bezparametrická analýza kontraktů byla rozdělena na dva kroky, jeden se provedl při vstupu do metody (přechody v konečných automatech) a druhý při výstupu z ní (detekce porušení kontraktu). Při parametrické analýze se oba tyto kroky vykonávají až při výstupu z metody, protože mezi sledovanými parametry může být i návratová hodnota funkce, která by při vstupu samozřejmě ještě nebyla známá. Nejdříve se provede algoritmus 2, který slouží k obsluze instancí, a poté se provede detekce porušení kontraktu podle algoritmu 1. Pokud budou splněny podmínky pro porušení kontraktu, zkontrolují se ještě navíc hodnoty parametrů vůči daným omezením.

Algoritmus 2 Část algoritmu pro parametrickou analýzu kontraktů.

Data: okno v, metoda m s názvem n_m a argumenty A vygenerovaná vláknem t

1: **for** $\varrho \in \mathbb{R}$ **do**
2: **for** $r \in [\varrho]_t^\tau : start(r) \in v$ **do**
3: **if** $n_m \in \Sigma_\varrho$ **then**
4: **if** $n_m = next(r)$ **then**
5: **if** $A \iff \mathbb{P}_r$ **then**
6: proveď přechod v instanci r
7: **end if**
8: **else**
9: **if** $A \iff \mathbb{P}_r$ **then**
10: invaliduj instanci r
11: **end if**
12: **end if**
13: **end if**
14: **end for**
15: **if** $n_m = first(\varrho)$ **then**
16: **if** $\nexists r \in [\varrho]_t^\tau : start(r) \in v \wedge A \iff \mathbb{P}_r$ **then**
17: **if** argumenty A splňují definovaná omezení **then**
18: vytvoř instanci r' s parametry $\mathbb{P}'_r \iff A$
19: **end if**
20: **end if**
21: **end if**
22: **end for**

Algoritmus 2 popisuje kroky pouze pro cíle daného kontraktu. Vše se musí zopakovat ještě jednou pro spoilery, ale jelikož je tato část totožná, není již v algoritmu uvedena. V algoritmu se vyskytuje několik pojmů, které ještě nebyly definovány. Pro abecedu konkrétního cíle je použito označení Σ_ϱ. Jedná se o množinu názvů všech metod, které se v sekvenci daného cíle vyskytují. Dále je pro každou běžící instanci r cíle ϱ definována událost $next(r)$, která označuje další očekávanou událost, která je součástí instance a umožňuje provedení přechodu v jejím konečném automatu. Podobně je pro každý cíl ϱ definována událost $first(\varrho)$, která označuje první událost daného cíle. $\mathbb{P}_r$ je množina všech parametrů a jejich hodnot instance r. Všechny uvedené pojmy je možné obdobně definovat také pro spoiler.

Kapitola 5

Implementace analyzátoru Contract-validator-with-params

Na základě návrhu byl v jazyce C/C++ implementován analyzátor Contract-validator-with-params jako rozšíření prostředí ANaConDA. V tomto prostředí již existuje analyzátor Contract-validator pro kontrakty rozšířené o cíle a spoilery, ale nepodporuje parametry. Jeho součástí jsou také zdrojové kódy pro okno stopy, vektorové hodiny a konečný automat. Jejich autorem je Ing. Jan Fiedor, Ph.D. Implementace parametrického analyzátoru z těchto kódů vychází a rozšiřuje je o další části nutné pro zpracování parametrů. Většina nového kódu se tak provádí při samotném zpracování konfiguračního souboru a dále při provádění algoritmu 2.

5.1 Formát specifikace kontraktu

Specifikaci kontraktu je možné zapsat do konfiguračního souboru `contracts.conf`, který se může nacházet buďto v adresáři, ze kterého bude spouštěna analýza (tedy pro každý program jiný), nebo ve výchozím adresáři pro konfiguraci prostředí ANaConDA (cesta `anaconda/framework/conf/contracts.conf`). Soubor se bude zpracovávat po řádcích a na každém očekává jednu z následujících možností:

- klauzule kontraktu,
- omezení typu **datový typ**,
- omezení typu **podmínka**,
- omezení typu **přiřazení hodnoty** nebo
- komentář, což je řádek uvozený znakem `#`.

Klauzule se zapisuje tak, jak je specifikováno níže pomocí rozvinuté Backusovy-Naurovy formy (EBNF):

```
klauzule              ::= '{' cíl '<-' spoiler (';' spoiler)* '}'
cíl                   ::= funkce+
spoiler               ::= funkce+
funkce                ::= (parametr '=')? identifikátor_funkce '('
                          seznam_parametrů? ')'
seznam_parametrů      ::= parametr (',' parametr)*
parametr              ::= identifikátor | '_'
identifikátor_funkce  ::= identifikátor ('::' identifikátor)?
identifikátor         ::= písmeno (písmeno | číslo | '_')*
```

Konkrétně by specifikace klauzule mohla vypadat například takto:

```
{A=fnc_bool() foo(A) <- bar(_,X)}
```

Datový typ musí být specifikován zvlášť pro každý parametr, který se v klauzuli objeví. Syntaxe tohoto omezení je:

```
omezení_datový_typ  ::= parametr ':' datový_typ
datový_typ          ::= 'int' | 'float' | 'double' | 'char*' | 'bool' |
                        'string' | 'void*'
```

Pro každý datový typ musí být v prostředí ANaConDA implementována podpora v podobě funkcí pro převod hodnot, porovnávání a výpočty výrazů, a proto je podporována pouze určitá množina datových typů. Pokud bude potřeba specifikovat parametr pro monitorování objektu, je možné pro něj využít datový typ `void*`.

Podmínky jsou výrazy, jejichž výsledkem je hodnota typu `bool`. Lze pomocí nich omezit hodnoty parametrů, pro které budou vytvářeny instance a pro které nikoli. Mezi operátory, které je pro zápis podmínek i dalších výrazů možno využít, patří `or`, `and`, `not`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `+`, `-`, `*`, `/`, `%`, `(`, `)` a jejich priorita je stejná jako u odpovídajících operátorů v jazyce C. Podporovány jsou také literály typu celé číslo, desetinné číslo, řetězec a pravdivostní hodnota, které se zapisují stejně jako v jazycích C/C++. Ve výrazech lze samozřejmě využívat i názvy parametrů. Veškeré operátory a literály je nutné oddělovat pomocí mezer. Specifikace podmínky pro parametr A typu `float` může vypadat například takto:

```
not ( A == 5.0 and ( A != 0.0 or A < 10.0 ) )
```

Přiřazením hodnoty lze pomocí výrazu jednoznačně určit hodnotu parametru. Syntaxe tohoto omezení je:

```
přiřazení_hodnoty ::= parametr '=' výraz
```

Pro výrazy platí podmínky popsané výše. Konkrétní zápis přiřazení může vypadat například takto:

```
B = ( ( ( A + 2 ) - 10 ) * 3 ) / 1
```

Je nutné, aby veškeré parametry a literály použité ve výrazu měly stejný datový typ, konkrétně zde `int`.

5.2 Rozšíření prostředí ANaConDA o podporu parametrů

V kapitole 2.4 bylo zmíněno, že základem analyzátorů jsou zpětná volání, díky kterým získávají informace o událostech v monitorovaném programu. ANaConDA poskytuje sadu funkcí pro registraci zpětných volání, která při požadované události také získají důležité informace například o identifikaci vlákna, zámku nebo o hodnotách argumentů. V současné době ANaConDA umožňuje zaregistrovat zpětné volání pro funkci s konkrétním názvem a získat hodnotu jejího argumentu, případně i návratové hodnoty. Pro parametrický analyzátor je ovšem nutné získávat hodnoty více argumentů najednou a tato podpora v prostředí dosud chyběla. Proto byla před samotnou implementací analyzátoru ANaConDA rozšířena o možnost registrovat dvě zpětná volání, přičemž první získá hodnoty požadovaných argumentů specifikovaných pomocí indexu a druhé návratovou hodnotu.

Dále bylo třeba implementovat podporu pro generické zpracování hodnot různých datových typů bez jejich znalosti při implementaci analyzátoru. Framework ANaConDA byl tedy rozšířen o různé konverzní a komparační funkce (soubory `framework/src/utils/convert`, `framework/src/utils/compare` a `framework/src/utils/functions`). Nabídka těchto funkcí je lehce rozšiřitelná i o další datové typy.

5.3 Schéma analyzátoru

Schéma analyzátoru je zobrazeno na obrázku 5.1. Contract-validator-with-params se skládá z několika komponent (některé jsou společné pro analýzu bez parametrů i s nimi):

- *Analyser* (soubor `contract-validator-with-params`) je hlavní kód analyzátoru, který řídí průběh analýzy na základě výskytu událostí v monitorovaném programu. Zajišťuje komunikaci s prostředím ANaConDA a obsluhu zpětných volání. Provádí operace nad vektorovými hodinami a posuny logických časů vláken při výskytu synchronizačních událostí a extrahuje informace o funkcích a jejich argumentech pro potřeby algoritmů pro samotnou analýzu. Detaily budou popsány v části 5.3.1.

- *Contract* (soubor `contract`) slouží především k načtení a uložení informací o kontraktu z konfiguračního souboru. Detaily budou popsány v části 5.3.2.

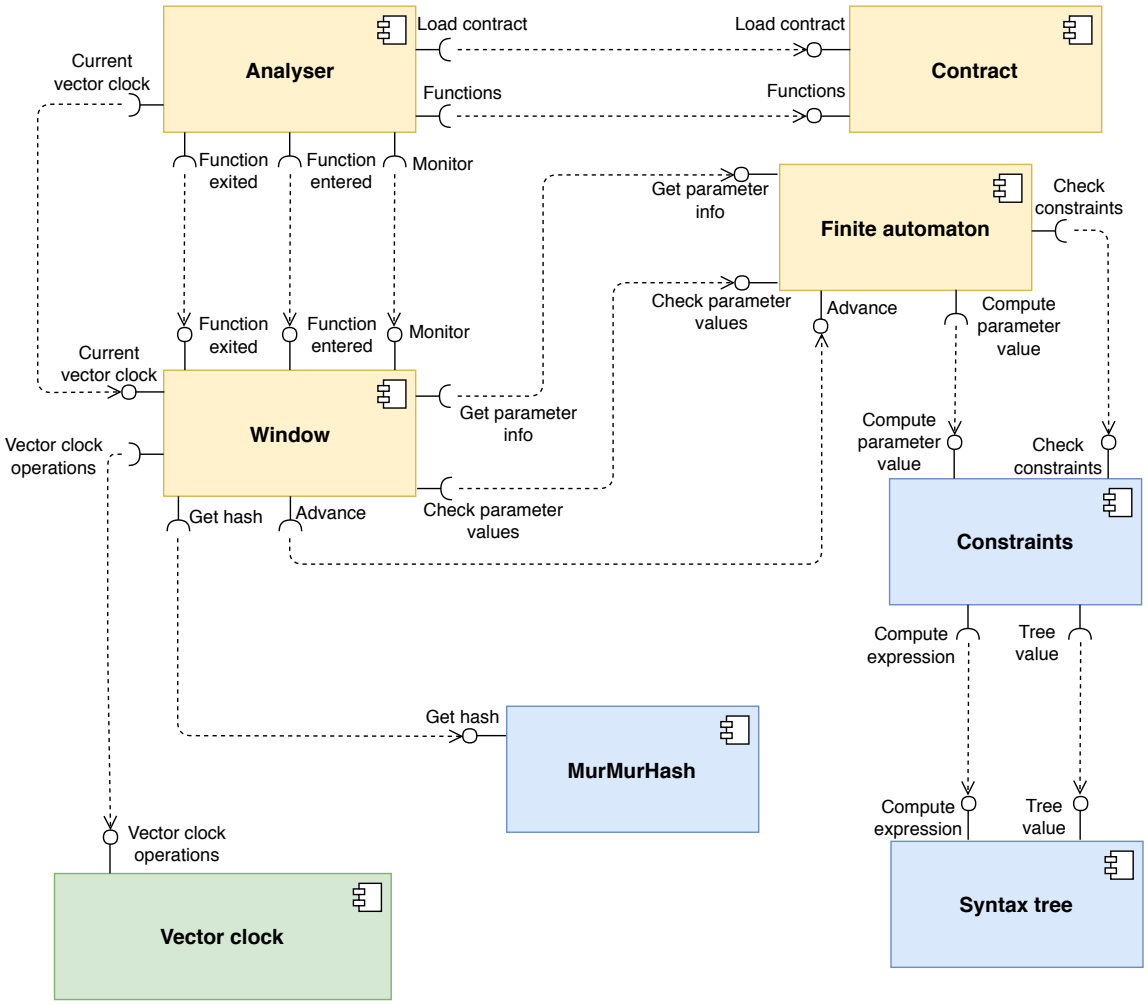
- *Window* (soubor `window`) realizuje okno stopy a udržuje informace o logickém času vlákna a také o instancích cílů a spoilerů ve vlákně. Implementuje metody pro obsluhu běžících instancí a pro detekci porušení kontraktu, které odpovídají dříve popsaným algoritmům 1 a 2. Detaily budou popsány v části 5.3.3.

- *Vector clock* (soubor `vc`) implementuje strukturu pro realizaci vektorových hodin a operací nad nimi včetně ověření *hb*-relace. Detaily budou popsány v části 5.3.4.

- *Finite automaton* (soubor `fa`) poskytuje struktury a třídy pro reprezentaci instancí cílů a spoilerů pomocí konečného automatu. Drží informace o hodnotách parametrů a poskytuje metody pro provádění přechodů a vytváření nových instancí. Detaily budou popsány v části 5.3.5.

- *MurMurHash* (soubor `MurMurHash`) je implementace nekryptografické hashovací funkce. Pro analyzátor byla využita implementace MurMurHash3 v C++ z veřejného kódu dostupném na GitHub[1] pod licencí MIT, ale původně funkci vytvořil a sdílel

[1] https://github.com/logrhythm/MurMurHash



Obrázek 5.1: Schéma komponent analyzátoru Contract-validator-with-params a jejich rozhraní. Zeleně jsou označeny komponenty společné pro parametrický i bezparametrický analyzátor, žlutě komponenty analyzátoru Contract-validator, které byly pro potřeby analýzy s parametry změněny, a modře komponenty, které musely být nově vytvořeny.

Austin Appleby[2]. V analyzátoru se pomocí ní vytváří hash argumentů funkce, aby se snadno dala ověřit existence instance pro stejnou kombinaci argumentů.

- *Constraints* (soubor `constraints`) poskytuje třídu, která drží informace o omezeních pro parametry, umožňuje vyhodnocovat jejich platnost a také vypočítat hodnoty parametrů pomocí výrazů. Detaily budou popsány v části 5.3.6.
- *Syntax tree* (soubor `syntax-tree`) provádí syntaktické a sémantické kontroly výrazů a sestavuje z nich binární strom, který poté používá pro jejich vyhodnocování. Detaily budou popsány v části 5.3.7.

Jednotlivé komponenty spolu komunikují pomocí rozhraní a to takto:

- *Analyser* – *Contract* komunikace probíhá především před samotným zahájením analýzy. Analyzátor zavolá metodu pro načtení informací o kontraktu z konfiguračního

[2] https://github.com/aappleby/smhasher/wiki

souboru a poté získá potřebné informace o funkcích a jejich argumentech, pro které je potřeba zaregistrovat zpětná volání. Získané informace o kontraktu dále budou předávány oknu stopy vždy při vzniku nového vlákna.

- *Analyser – Window* komunikace probíhá v průběhu celé analýzy. Začíná voláním metody `monitor` při vzniku nového vlákna, která slouží k uložení informací o kontraktu do okna stopy. Pokud dojde k synchronizační operaci nad zámkem, je potřeba v analyzátoru zjistit logický čas vlákna, který je taktéž uložen v jeho okně stopy. Metoda `functionEntered`, resp. `functionExited` se volá při vykonávání monitorované funkce pro provedení algoritmu 2, resp. 1.

- *Window – Vector clock* komunikace probíhá vždy, když je potřeba provádět operaci nad vektorovými hodinami a také při detekci porušení kontraktu pro kontrolu *hb*-relace.

- *Window – Finite automaton* komunikace slouží k obsluze instancí, které konečný automat reprezentuje. Zahrnuje metodu pro provedení kroku v automatu a pro získání informací o parametrech a jejich hodnotách. Při kontrole, zda došlo k porušení kontraktu, je také potřeba ověřovat, zda platí omezení pro kombinace parametrů cíle a spoileru, k čemuž komponenta *Finite automaton* také poskytuje metodu.

- *Window – MurMurHash* komunikace probíhá při vykonávání řádku 16 algoritmu 2, tedy když je potřeba ověřit, zda pro danou kombinaci hodnot již existuje instance.

- *Finite automaton – Constraints* komunikace se provádí při vzniku instance, kdy je potřeba pomocí výrazů dopočítat hodnoty nedefinovaných parametrů a také ověřit platnost omezení, a při provádění algoritmu 1 znovu pro ověření platnosti omezení.

- *Constraints – Syntax tree* komunikace probíhá při vyhodnocování výrazů. *Constraints* vyhodnocuje platnost všech omezení a pro výpočet jednotlivých výsledků využívá metody komponenty *Syntax tree*.

Důležité datové typy použité v implementaci analyzátoru

Obrázek 5.2 ilustruje datové typy, strukturu a výčtový typ, které se používají napříč všemi komponentami v analyzátoru:

- Výčtový typ `ArgumentType` reprezentuje podporované datové typy parametrů.

- Datový typ `Type` se používá pro identifikátor cíle a spoileru.

- Datový typ `Arguments` slouží k reprezentaci hodnot argumentů funkce na určitých indexech.

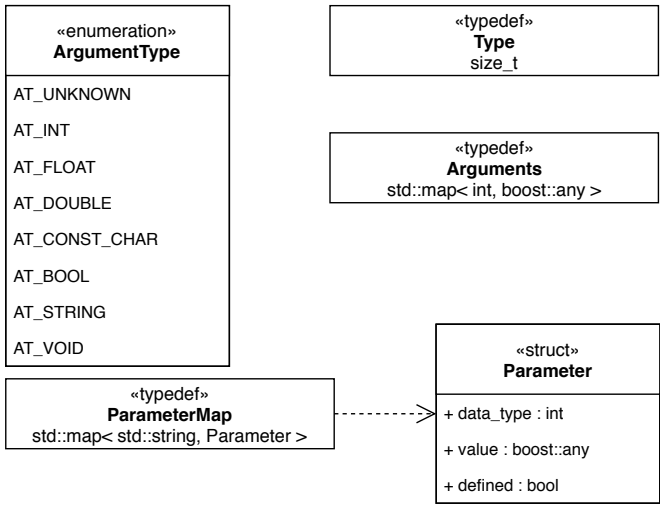
- Struktura `Parameter` udržuje informace o konkrétním parametru – jeho datový typ, informaci o tom, zda má definovanou hodnotu a příp. i samotnou hodnotu.

- Datový typ `ParameterMap` slouží k udržení informací o parametrech na základě jejich názvu.

V implementaci je třeba rozlišovat mezi datovými typy `Arguments` a `ParameterMap`. `Arguments` lze vytvořit hned při volání metody, je pouze potřeba zjistit datové typy argumentů, aby mohly být korektně přetypovány a uloženy do datového typu `boost::any`.

`ParameterMap` je možné vytvořit pouze pro konkrétní stav konkrétního konečného automatu, pro řešení situací, kdy se stejná funkce vyskytne v sekvenci cíle či spoileru vícekrát, ale s různými parametry:

```
{ A=foo() foo(A) <- bar() }
```



Obrázek 5.2: Důležité datové typy používané v různých komponentách analyzátoru Contract-validator-with-params.

5.3.1 Obsluha zpětných volání

V hlavním kódu analyzátoru tvoří největší část obsluha zpětných volání. Při spuštění analýzy se zaregistrují obslužné funkce pro požadované události a při jejich výskytu se provádí akce popsané v tabulce 5.1.

Tabulka 5.1: Popis jednotlivých akcí, které se v analyzátoru vykonají při výskytu události v monitorovaném programu. Význam zkratek: VH – vektorové hodiny, LČ – logický čas, NH – návratová hodnota. Algoritmy u události **Konec exekuce funkce** jsou uvedeny v pořadí, ve kterém se vykonají.

Událost	**Akce**
Obdržení zámku	Aktualizace VH vlákna (viz 3.5)
Uvolnění zámku	Aktualizace VH zámku a inkrementace LČ vlákna (viz 3.6)
Operace *fork*	Aktualizace VH a LČ vláken (viz 3.7)
Operace *join*	Aktualizace VH a LČ vláken (viz 3.8)
Začátek vlákna	Vytvoření okna stopy a načtení informací o kontraktu
Začátek exekuce funkce	Uložení hodnot argumentů funkce
Konec exekuce funkce	Získání argumentů a NH, vykonání algoritmů 2 a 1

Struktury potřebné pro fungování analýzy zahrnují:

- soukromá data vláken (ANaConDA poskytuje tzv. *Thread Local Storage*),

- mapu pro uložení hodnot argumentů a jejich předání mezi zpětným voláním při začátku a konci exekuce funkce a
- kontejner obsahující vektorové hodiny zámků.

Soukromá data vláken jsou přístupná na základě klíče generovaného pro unikátní identifikaci vlákna, takže není potřeba používat synchronizaci pro výlučný přístup. Mapa argumentů před přístupem k datům provádí zamknutí kritické sekce. Úprava vektorových hodin zámků může být narušena přepnutím kontextu, takže je potřeba zajistit výlučný přístup pomocí zámků.

Identifikace vlákna získaná nástrojem Pin nemusí být unikátní, a proto v kódu analyzátoru ještě dochází k vygenerování unikátní identifikace, která taktéž musí být pomocí zámků chráněna před chybami souběžného přístupu.

5.3.2 Kontrakt

Komponenta kontrakt obsahuje implementaci struktur pro cíl a spoiler a třídy pro samotný kontrakt. Vazby mezi nimi jsou zobrazeny na diagramu tříd 5.3. Struktury pro cíl i spoiler obsahují atributy:

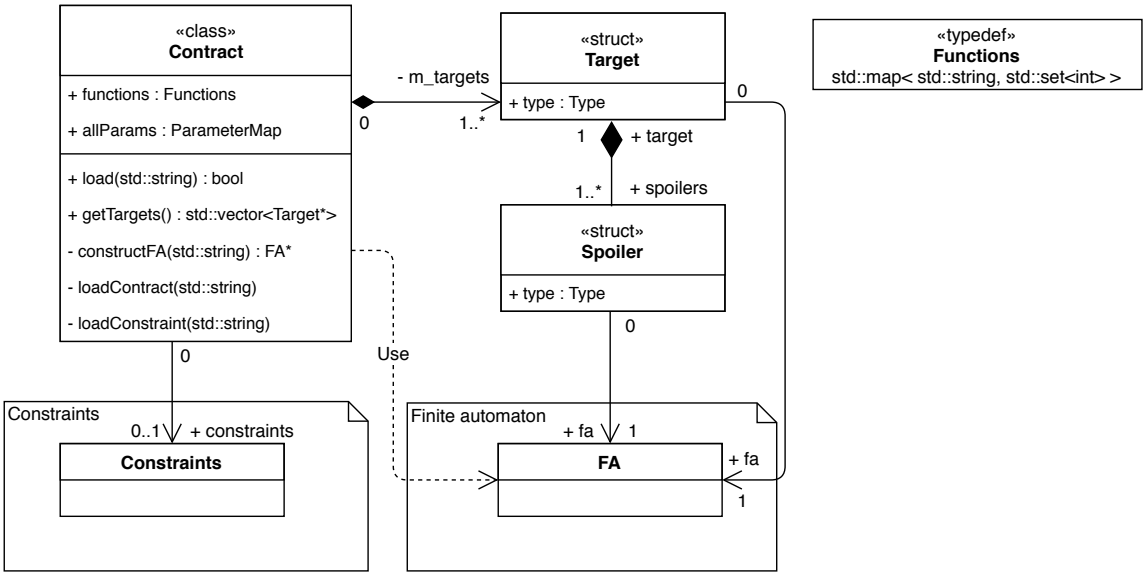
- `type` – identifikace cíle, resp. spoileru pro okno stopy,
- `fa` – konečný automat vyjadřující sekvenci cíle, resp. spoileru (viz 5.3.5),
- `spoilers`, resp. `target` – odkaz na spoilery, které mohou target porušit, resp. odkaz na target, který může spoiler porušit.

Kontrakt obsahuje atributy:

- `functions` – názvy všech monitorovaných funkcí včetně množiny obsahující indexy monitorovaných argumentů,
- `allParams` – informace o všech parametrech, které se ve specifikaci kontraktu vyskytly,
- `m_targets` – seznam všech cílů (resp. klauzulí) v kontraktu,
- `constraints` – reprezentace omezení parametrů (viz 5.3.6)

a metody:

- `load()` – veřejná metoda pro načtení informací z konfiguračního souboru a vytvoření objektu obsahující všechna specifikovaná data,
- `getTargets()` – veřejná metoda pro přístup k množině cílů,
- `constructFA()` – soukromá metoda pro vytvoření konečného automatu z řetězce, který popisuje cíl, resp. spoiler,
- `loadContract()` – soukromá metoda pro zpracování řádku, který obsahuje specifikaci klauzule kontraktu,
- `loadConstraint()` – soukromá metoda pro zpracování řádku, který obsahuje specifikaci omezení libovolného typu.



Obrázek 5.3: Diagram zobrazující vazby mezi třídami v komponentě kontrakt.

5.3.3 Okno stopy

Třídní diagram pro komponentu *Window* se nachází na obrázku 5.4. Základem je třída `Window`, která obsahuje atributy:

- `m_tid` – identifikace vlákna, kterému dané okno náleží,
- `m_targets` – množina všech monitorovaných cílů a jejich instancí vzájemně odlišných hodnotami parametrů,
- `m_spoilers` – množina všech monitorovaných spoilerů a jejich instancí vzájemně odlišných hodnotami parametrů,
- `cvc` – aktuální logický čas vlákna,
- `m_windows` – seznam všech oken stop pro všechna vlákna v programu

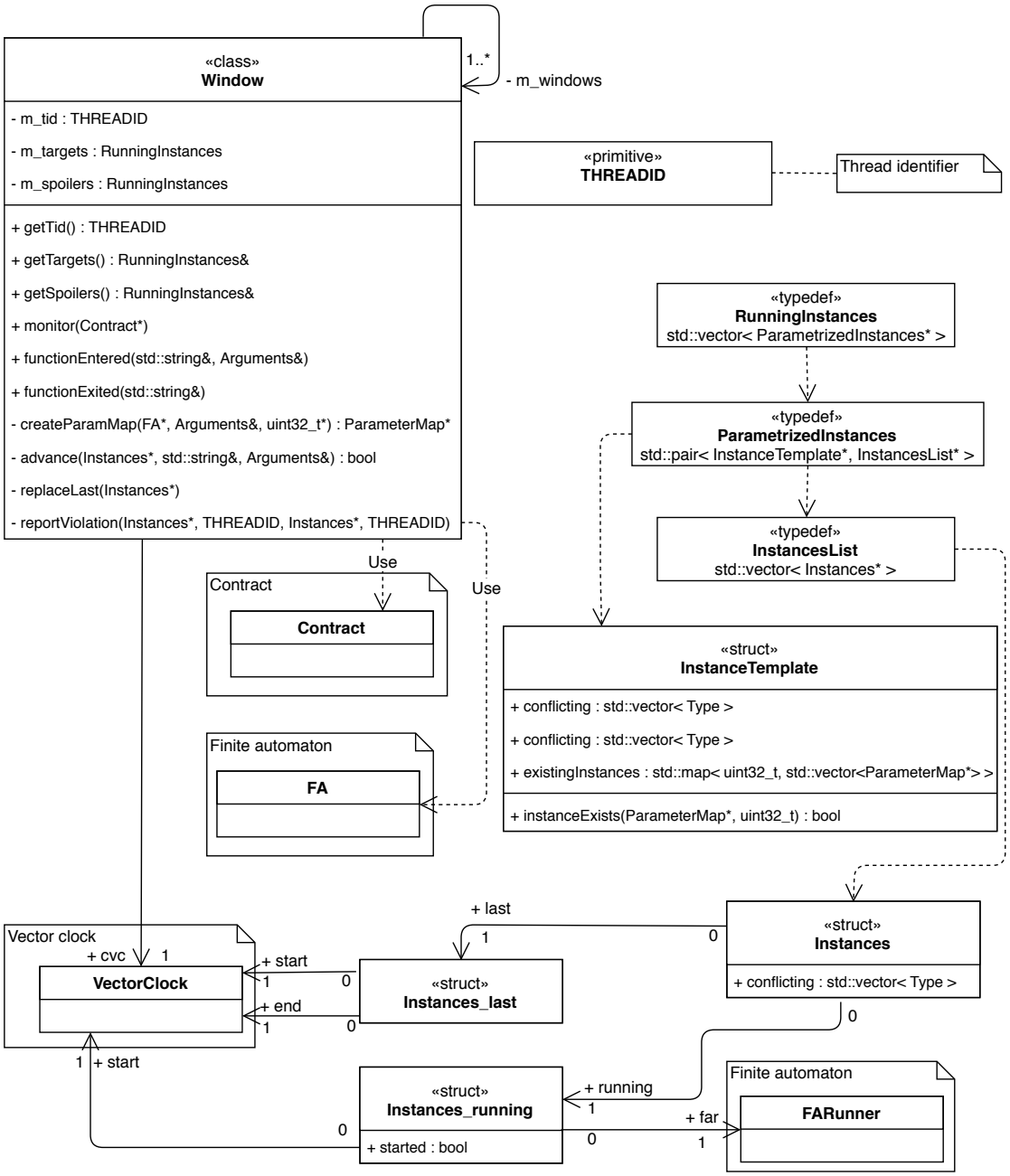
a metody:

- `getTid()` – veřejná metoda pro přístup k soukromé identifikaci vlákna,
- `getTargets()` – veřejná metoda pro přístup k soukromé množině cílů,
- `getSpoilers()` – veřejná metoda pro přístup k soukromé množině spoilerů,
- `monitor()` – veřejná inicializační metoda pro okno stopy, kdy se z objektu třídy `Contract` (viz 5.3.2) vytvoří příslušné šablony pro budoucí instance (tedy objekty stuktury `InstanceTemplate`),
- `functionEntered()` – veřejná metoda, která na základě názvu funkce a jejích argumentů provede přechody v konečných automatech instancí a případně vytvoří nové instance (jedná se o implementaci algoritmu 2, na obrázcích 5.5 a 5.6 se nachází vývojový diagram),

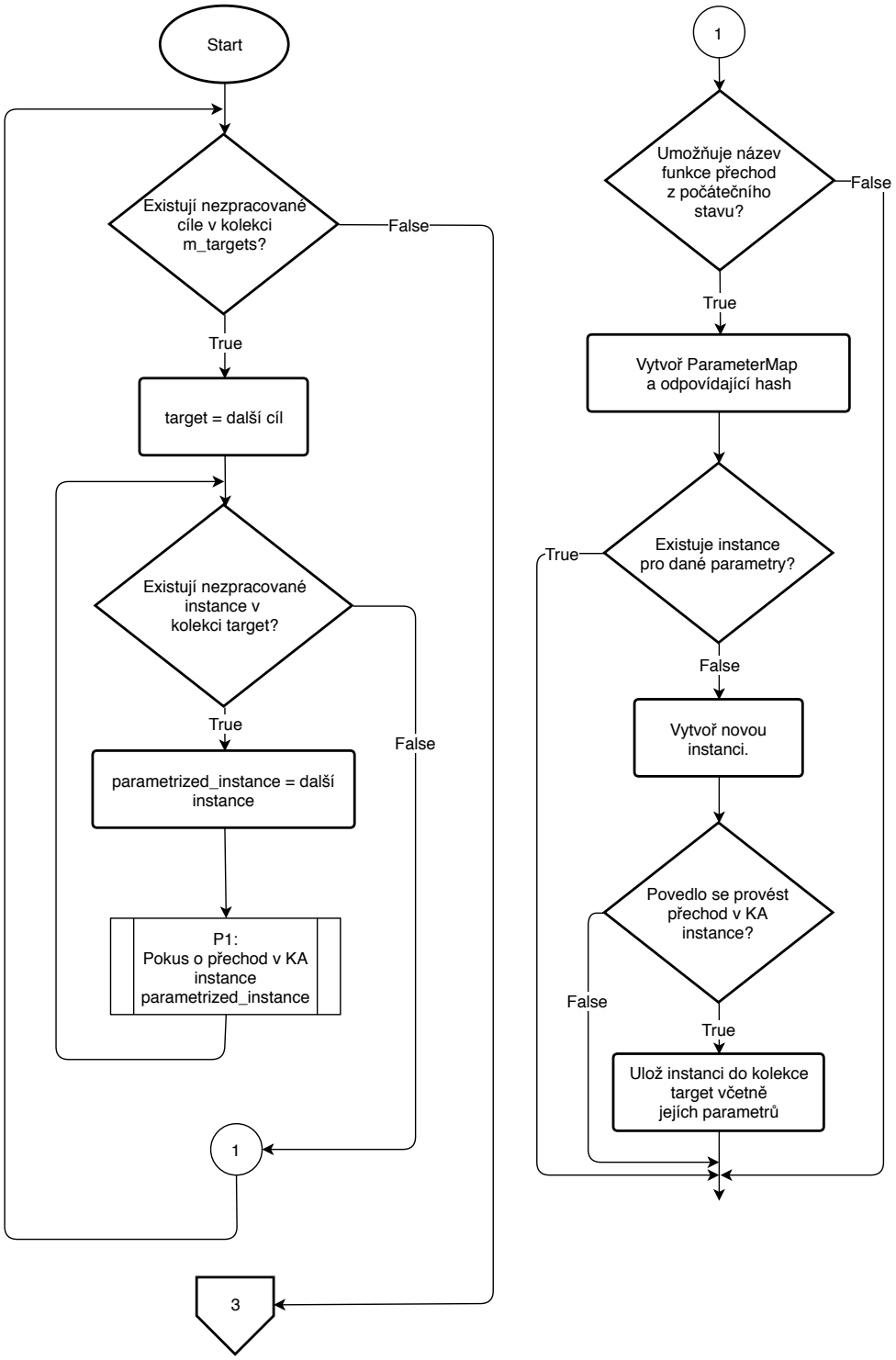
- `functionExited()` – veřejná metoda, která provede kontroly, zda došlo k porušení kontraktu (jedná se o implementaci algoritmu 1),
- `createParamMap()` – soukromá metoda, která ze získaných argumentů funkce vytvoří mapu parametrů potřebnou pro kontroly, zda již pro danou kombinaci hodnot existuje instance (metoda `instanceExists()` struktury `InstanceTemplate`),
- `advance()` – soukromá metoda, která se pokouší provést přechod v konečném automatu instance a v závislosti na výsledku této akce může invalidovat instanci (vývojový diagram pro popis metody je zobrazen na obrázku 5.7),
- `replaceLast()` – soukromá metoda, která vymaže z paměti poslední dokončenou instanci a nahradí ji novou dokončenou instancí,
- `reportViolation()` – soukromá metoda, která vypíše chybovou hlášku o porušení kontraktu spolu s identifikací vláken obsahujících instance cíle a spoileru, příp. i s hodnotami parametrů, které porušení způsobily.

Struktura `Instances` reprezentuje jednu ukončenou a jednu běžící instanci cíle či spoileru s konkrétními hodnotami parametrů. U dokončené instance jsou uloženy vektorové hodiny jejího začátku a konce, u běžící instance příznak toho, zda již instance začala, vektorové hodiny jejího začátku a reference na objekt třídy `FARunner` (viz 5.3.5), který realizuje přechody v konečných automatech. Pro zachycení vazby mezi cílem a spoilerem, který ho může porušit, se používá atribut `conflicting`. Jedná se o seznam čísel, která identifikují daný cíl či spoiler (atribut `type` u struktur `Target` a `Spoiler` viz 5.3.2).

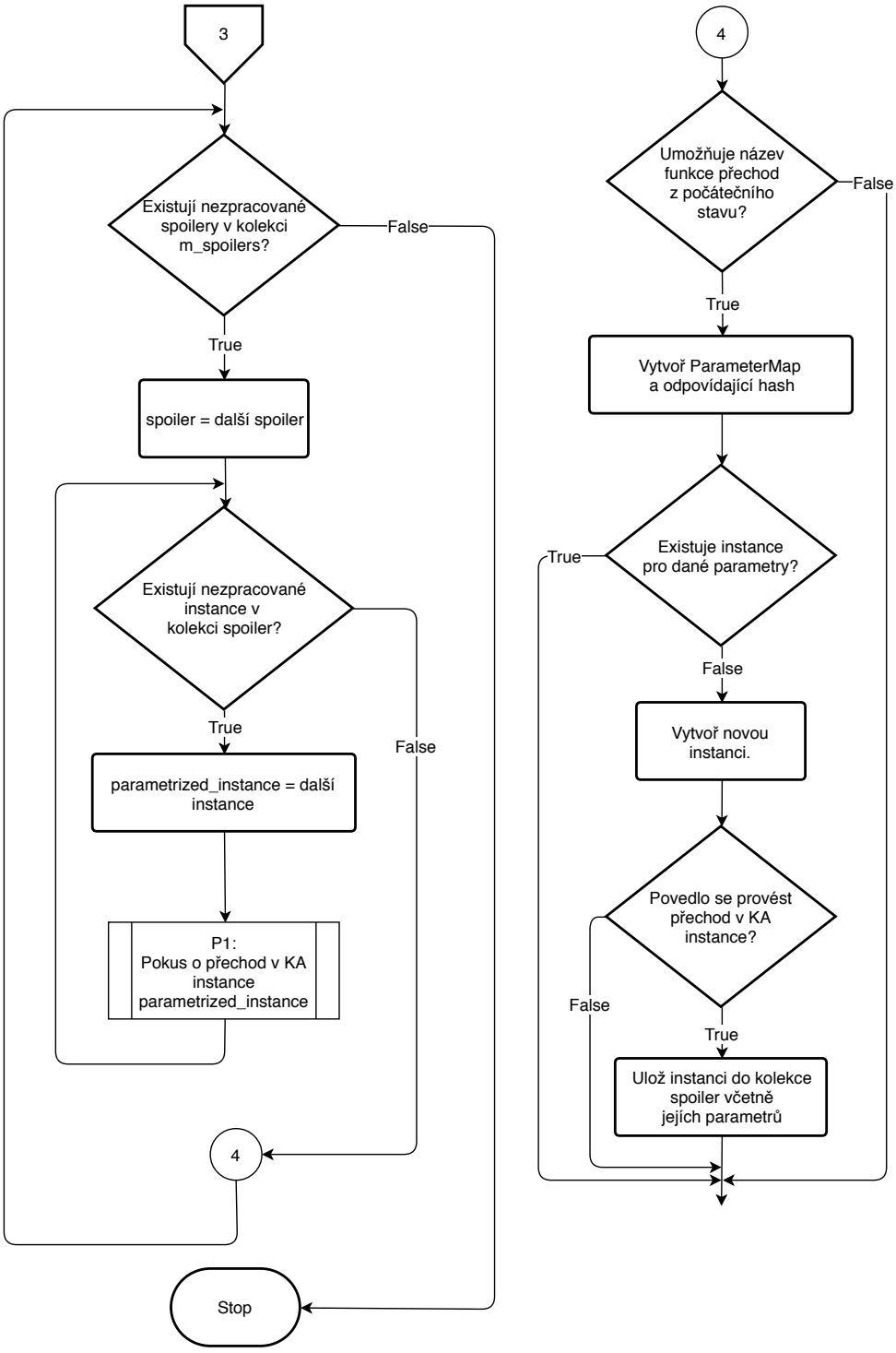
Pro jeden cíl či spoiler může existovat více instancí, proto je z nich vytvořen `InstancesList`. `InstanceTemplate` má dva účely – jednak obsahuje informace společné pro všechny instance daného cíle či spoileru, ale také si udržuje informace o všech existujících instancích a jejich parametrech. Pomocí metody `instanceExists()` se rozhoduje, zda pro funkci a její argumenty již existuje instance či je potřeba vytvořit novou.



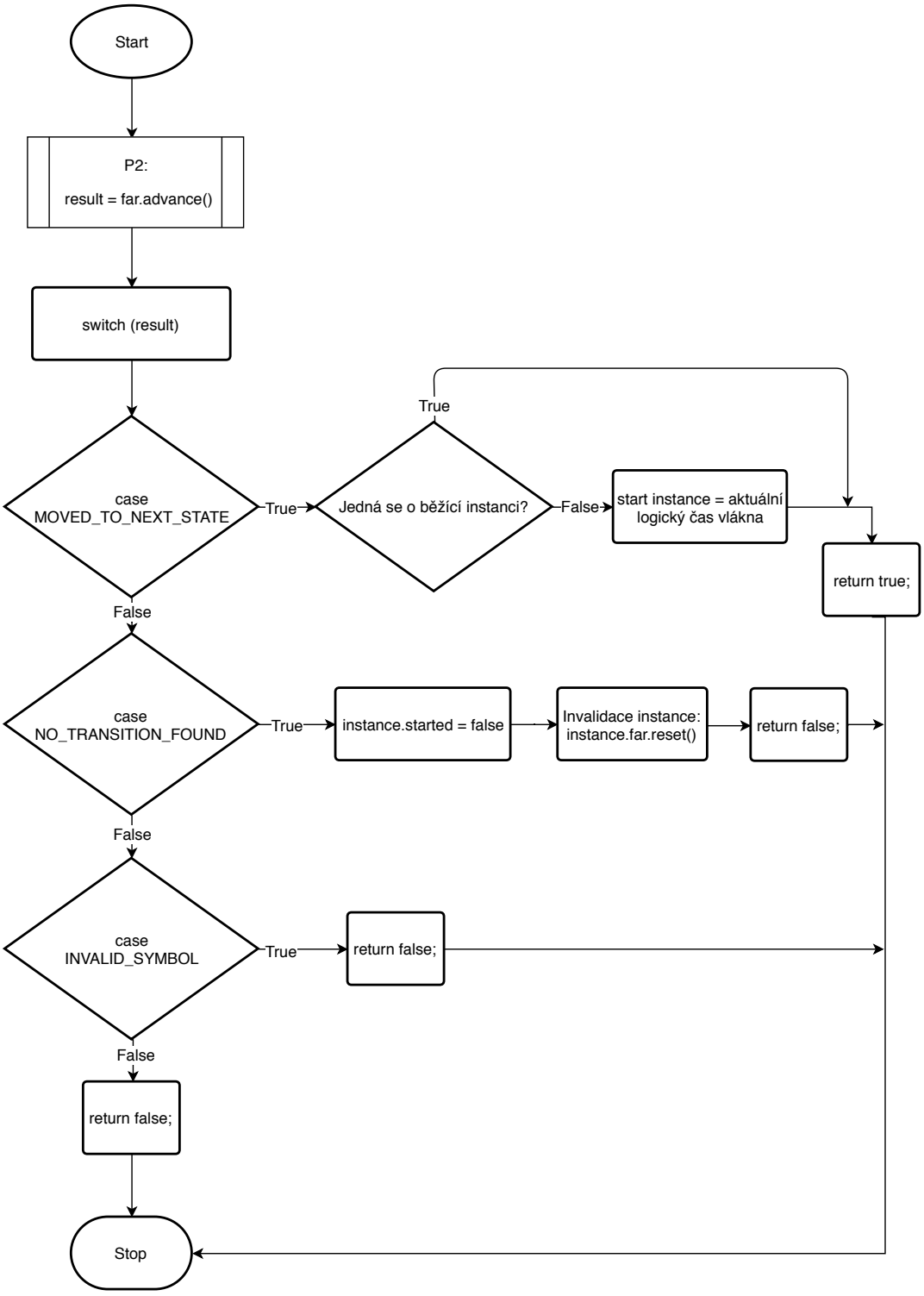
Obrázek 5.4: Diagram zobrazující vazby mezi třídami v komponentě pro realizaci okna stopy.



Obrázek 5.5: Vývojový diagram popisující část metody `functionEntered()` pro cíle. Zbývající část pro spoilery je zobrazena na obrázku 5.6. Podproces P1 je zobrazen na obrázku 5.7.



Obrázek 5.6: Vývojový diagram popisující část metody `functionEntered()` pro spoilery. Podproces P1 je zobrazen na obrázku 5.7.



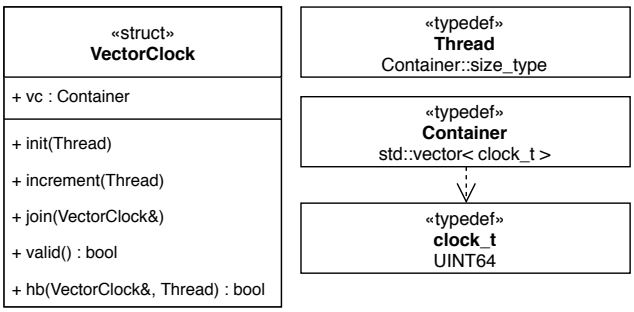
Obrázek 5.7: Vývojový diagram popisující metodu `advance()` ve třídě `Window`. Podproces P2 je zobrazen na obrázku 5.10.

5.3.4 Vektorové hodiny

Na obrázku 5.8 je zobrazeno schéma struktury `VectorClock` a použitých datových typů. Tato komponenta byla z analyzátoru Contract-validator převzata v nezměněné podobě. Základem je atribut `vc`, což je vektor obsahující logické časy vláken na odpovídajících indexech. Pro potřeby analýzy kontraktů a realizace *hb*-relace se na indexu daného vlákna t, kterému vektorové hodiny náleží, nachází jeho logický čas a na indexech ostatních vláken logický čas poslední události, která je v *hb*-relaci s aktuální událostí ve vláknu t.

Dále struktura `VectorClock` obsahuje implementaci operací:

- `init()` – Inicializace vektorových hodin vlákna t. Na všechny indexy různé od t je nastavena hodnota 0, index t je inicializován na hodnotu 1.
- `increment()` – Inkrementuje logický čas vlákna t (viz 3.4).
- `join()` – Spojí vektorové hodiny s jinými (viz 3.2).
- `valid()` – Metoda kontroluje, zda byly vektorové hodiny inicializovány.
- `hb()` – Kontroluje, zda se událost předaná argumentem metody stala před událostí, kterou reprezentuje objekt, nad kterým je metoda volána. Jinými slovy, kontroluje, zda `argument` $\prec_{hb}$ `this`.



Obrázek 5.8: Třídní diagram reprezentující komponentu vektorové hodiny.

5.3.5 Konečný automat

Komponentu pro realizaci konečného automatu, jejíž schéma se nachází na obrázku 5.9, tvoří dvě struktury a jedna třída. Struktura `FAState` reprezentuje jeden stav a obsahuje atributy:

- `paramIndices` – Na základě názvů parametrů určuje, na kterém indexu se nachází argument funkce/metody, jehož hodnoty parametr nabude. Například pro cíl `A=foo(_) foo(A)` budou existovat dvě instance struktury `FAState`, přičemž každá bude obsahovat jeden prvek v atributu `paramIndices`. V prvním případě se bude jednat o dvojici `(A,0)` a ve druhém o dvojici `(A,1)`.
- `accepting` – Příznak pro označení koncového stavu.
- `transitions` – Mapa přechodů. Klíč je typu `std::string` a obsahuje název funkce či metody, se kterým je možno provést přechod. Hodnotou je potom ukazatel na jinou instanci struktury `FAState`.

Struktura `FA` reprezentuje jeden konečný automat. Obsahuje atributy:

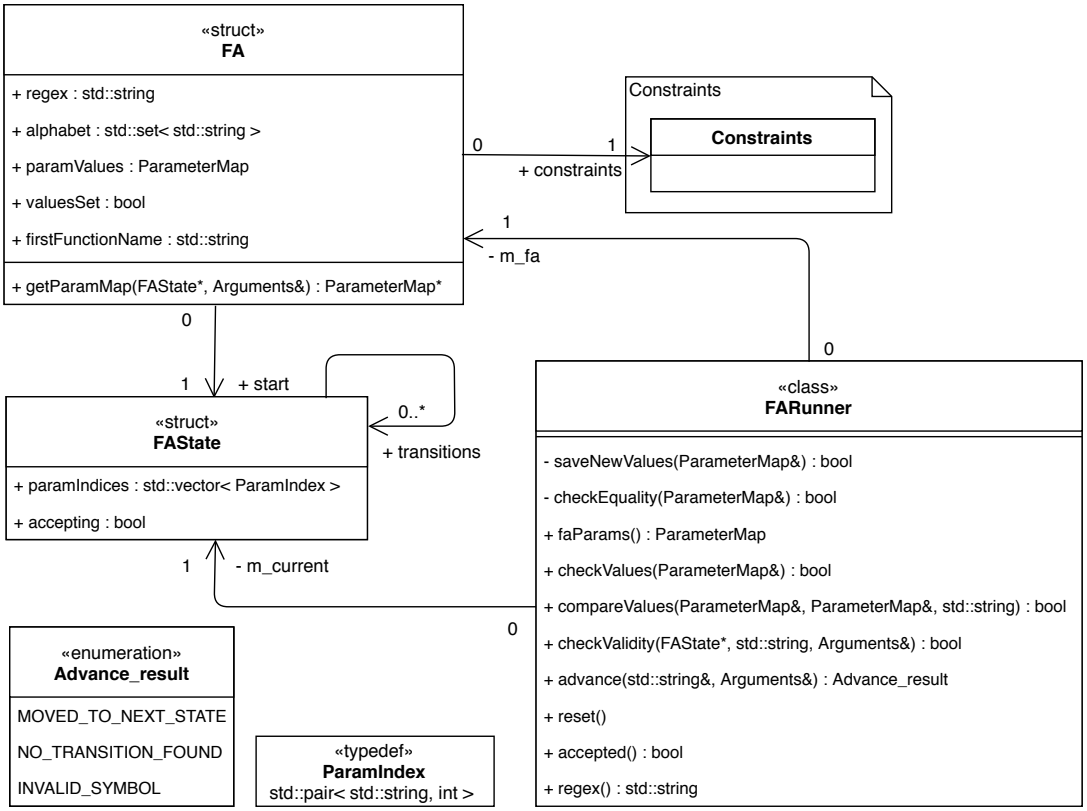
- `regex` – Hodnota typu `std::string` obsahující popis regulárního výrazu, který konečný automat reprezentuje. Atribut není pro analýzu podstatný, používá se pro informační výpisy.
- `alphabet` – Množina všech názvů funkcí/metod, které se v konečném automatu objevují (tedy abeceda konkrétního cíle či spoileru).
- `paramValues` – Informace o parametrech včetně jejich hodnot.
- `valuesSet` – Pravdivostní hodnota označující, zda má konečný automat již definované hodnoty parametrů.
- `firstFunctionName` – Název první funkce/metody, se kterou je možné provést přechod z počátečního stavu.
- `start` – Počáteční stav.
- `constraints` – Ukazatel na instanci třídy `Constraints` (viz 5.3.6), která obsahuje potřebné metody pro kontrolu, zda hodnoty parametrů splňují specifikovaná omezení.

Struktura `FA` obsahuje pouze jednu metodu a to `getParamMap()`. Tato metoda převádí argumenty funkce/metody reprezentované datovým typem `Arguments` na datový typ `ParameterMap`. Jak již bylo zmíněno dříve, tento převod je možný pouze pro konkrétní stav konečného automatu. Při transformaci se využívají informace uložené v atributu `paramValues` (např. datový typ) a v atributu `paramIndices`.

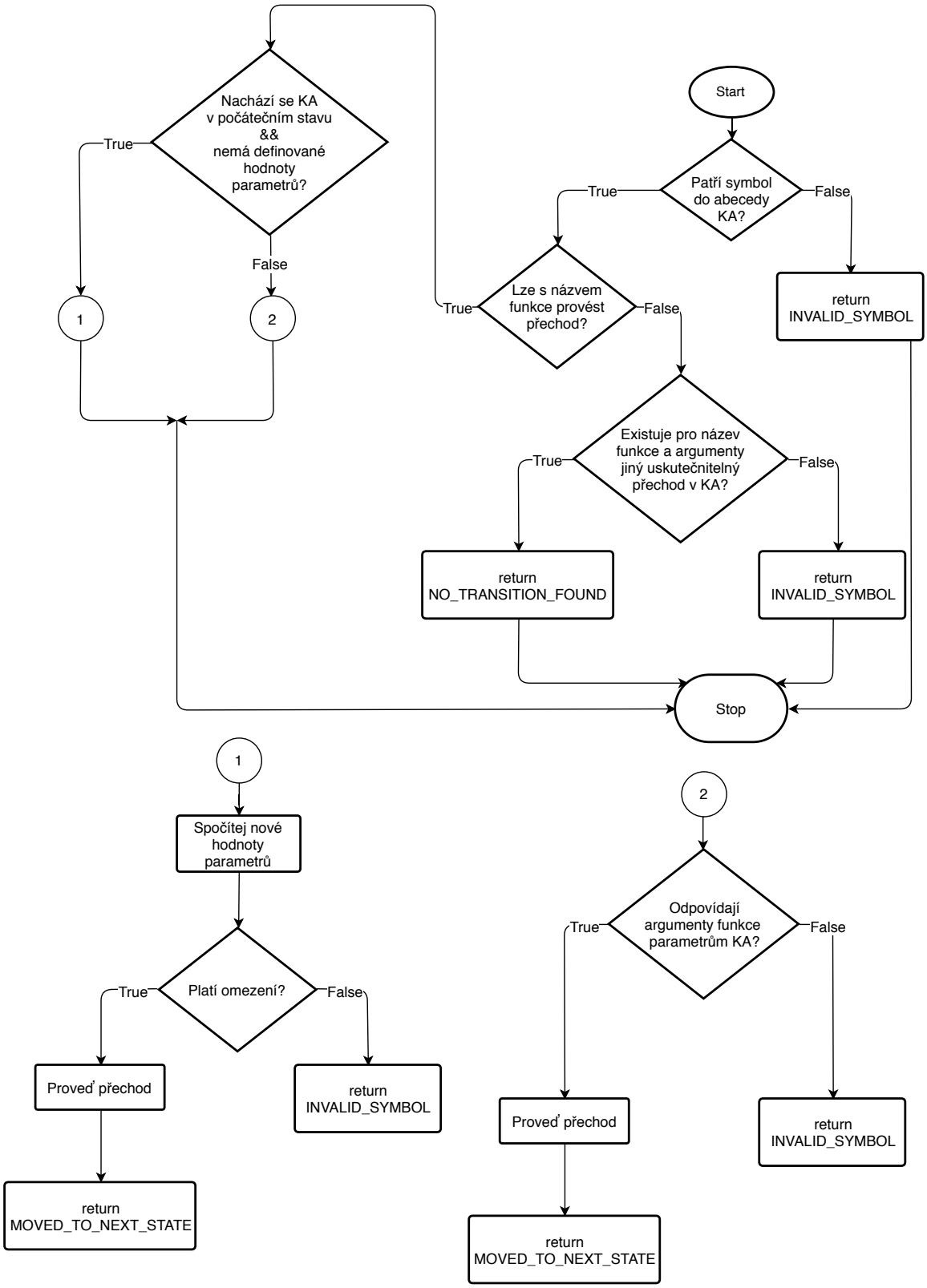
Třída `FARunner` reprezentuje jeden běh v konečném automatu `FA`. Obsahuje atribut `m_fa`, což je reference na konečný automat, a atribut `m_current` obsahující ukazatel na aktuální stav v konečném automatu. Dále poskytuje metody:

- `saveNewValues()` – Metoda dopočítá hodnoty parametrů podle specifikovaných výrazů, zkontroluje platnost omezení a pokud platí, uloží hodnoty do atributu `paramValues` konečného automatu `m_fa`.
- `checkEquality()` – Kontrola, zda se hodnoty parametrů předané metodě argumentem rovnají hodnotám uloženým v atributu `paramValues` konečného automatu `m_fa`.
- `faParams()` – Veřejná metoda pro přístup k atributu `paramValues` soukromé reference na konečný automat `m_fa`.
- `checkValues()` – Metoda kontroluje, zda parametry předané metodě argumentem splňují omezení uložená v atributu `constraints` konečného automatu `m_fa`.
- `compareValues()` – Kontrola, zda dva parametry mají stejnou hodnotu.
- `checkValidity()` – Metoda pro kontrolu, zda je daná instance validní. Pokud v konečném automatu existuje přechod pro danou kombinaci názvu funkce/metody a jejích argumentů, pak je považována za nevalidní.
- `advance()` – Metoda pro provedení přechodu v konečném automatu. Vývojový diagram se nachází na obrázku 5.10.
- `reset()` – Metoda pro návrat zpět do počátečního stavu.

- `accepted()` – Kontrola, zda se konečný automat `m_fa` nachází v koncovém stavu.
- `regex()` – Pomocná metoda pro výpis regulárního výrazu konečného automatu `m_fa`.



Obrázek 5.9: Diagram tříd a struktur pro implementaci konečného automatu.



Obrázek 5.10: Vývojový diagram pro popis metody `advance()`.

5.3.6 Omezení

Komponenta pro realizaci omezení parametrů, jejíž schéma se nachází na obrázku 5.11, implementuje strukturu, která ve svých atributech udržuje informace o jednotlivých výrazech a pomocí metod umožňuje jejich vyhodnocování. Pro uložení informací o datových typech se používají dvě reprezentace. Datový typ `ParamDataType` mapuje jméno parametru na jeho datový typ. Zpětné volání pro zpracování argumentů funkce obdrží pouze ukazatele do paměti, na kterých se nachází hodnoty, a aby bylo možné je korektně přetypovat, je potřeba pro každý argument znát jeho datový typ. K tomuto účelu slouží mapa `FunctionParams`, která pro monitorované funkce udržuje informaci o datovém typu argumentu na základě jeho indexu.

Atributy struktury `Constraints` jsou:

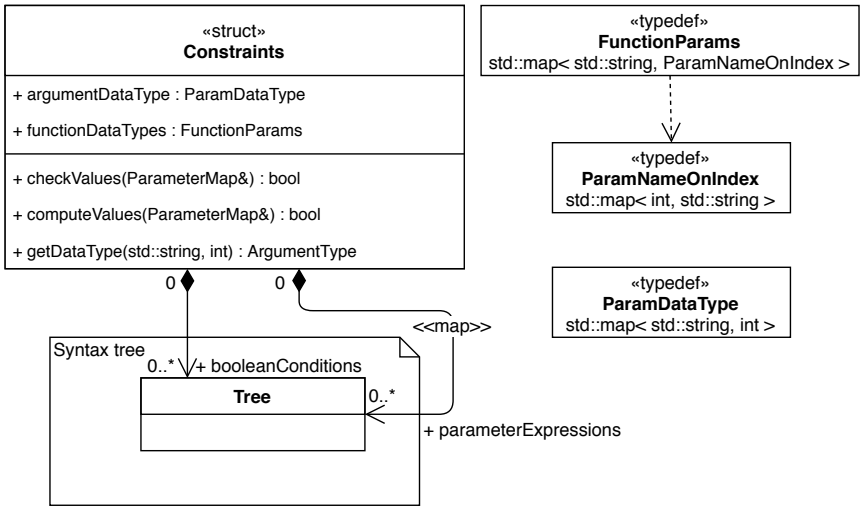
- `argumentDataType` – Atribut obsahuje informace o parametrech a jejich datových typech.
- `functionDataTypes` – Atribut obsahuje informace o datových typech argumentů funkcí.
- `booleanConditions` – Vektor ukazatelů na instance struktury `Tree` (viz 5.3.7), které reprezentují omezení typu podmínka.
- `parameterExpressions` – Pokud je v konfiguračním souboru uveden výraz pro přiřazení hodnoty parametru, bude uložen v tomto atributu. Jedná se o mapu, kde klíčem je název parametru a hodnotou instance struktury `Tree` realizující uvedený výraz.

Struktura `Constraints` poskytuje metody pro vyhodnocení omezení, konkrétně:

- `checkValues()` – Metoda kontroluje, zda pro zadané parametry platí všechna omezení typu podmínka. Pokud pro výpočet omezení nejsou definovány hodnoty všech potřebných parametrů, je považováno za platné. K tomuto může dojít například v situaci, kdy je zadána podmínka, ve které se porovnávají parametry cíle a spoileru.
- `computeValues()` – Metoda vyhodnotí všechny výrazy v atributu `parameterExpressions` a do vstup-výstupní mapy parametrů uloží jejich nové hodnoty.
- `getDataType()` – Metoda vrátí datový typ argumentu na základě názvu funkce a jeho indexu.

5.3.7 Syntaktický strom

Komponenta pro syntaktický strom, jejíž schéma se nachází na obrázku 5.12, implementuje především funkce nutné pro zpracování výrazů a samotné vytvoření binárního stromu pro jejich reprezentaci (struktura `Tree`). Zpracování výrazů probíhá pomocí syntaktické analýzy zdola nahoru s využitím precedenční tabulky. Z reprezentace výrazu pomocí `std::string` je nejdříve vytvořen vektor tokenů. Ten je poté transformován na oboustrannou frontu (`std::deque`), která realizuje postfixovou reprezentaci výrazu. Z ní je sestrojen samotný binární strom. Posledním krokem jsou sémantické kontroly datových typů, které zároveň do všech uzlů stromu doplní jejich datový typ.

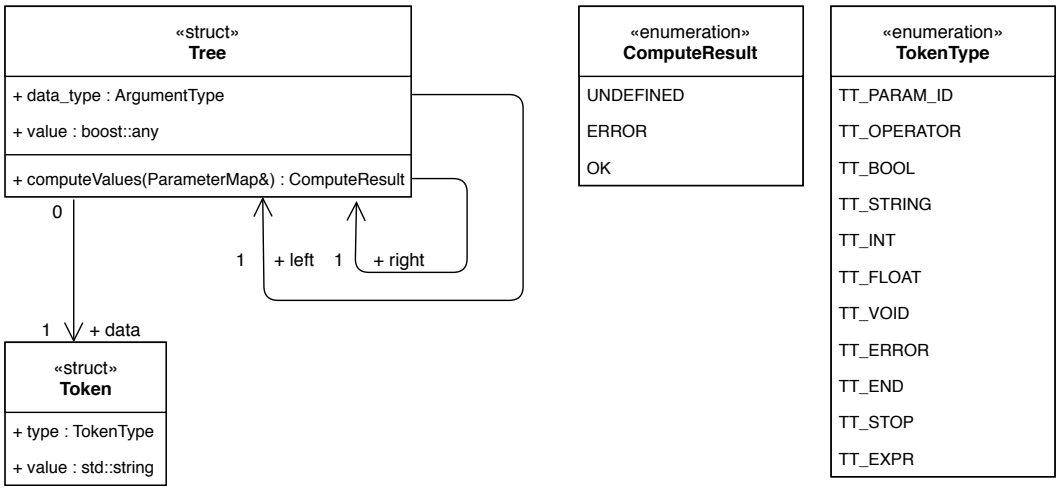


Obrázek 5.11: Diagram tříd pro popis komponenty realizující omezení pro parametry.

Struktura `Tree` reprezentuje binární strom (resp. jeden jeho uzel) a kromě ukazatelů na své dva potomky obsahuje atributy:

- `data_type` – Datový typ uzlu vyjádřený pomocí výčtového datového typu `ArgumentType` (viz 5.3).
- `value` – Hodnota uzlu, která je definována až po prvním vyhodnocení výrazu.
- `data` – Data uzlu reprezentována pomocí struktury `Token`.

K dispozici je pouze jedna veřejná metoda pro vyhodnocení výrazu a tedy výpočet hodnot všech uzlů stromu – `computeValues()`.



Obrázek 5.12: Diagram komponenty pro reprezentaci výrazů pomocí binárního stromu.

5.4 Testování implementace

Výsledný analyzátor byl podroben testování pomocí cca 100 automatizovaných testovacích případů zaměřených především na parametry a vyhodnocení výrazů, ale i na rozdíly mezi parametrickou a bezparametrickou analýzou. ANaConDA poskytuje skripty umožňující automatizaci testování a jednoduchou tvorbu nových testovacích případů, které navíc mohou být v budoucnu využity i pro nasazení průběžné integrace.

Jednotlivé testy jsou rozděleny do několika sad:

- Testovací případy pro různé kombinace datových typů parametrů a operátorů. Zahrnují:
  - `args_bool`,
  - `args_char`,
  - `args_double`,
  - `args_float`,
  - `args_int`,
  - `args_string` a
  - `args_void`

  přičemž název vždy určuje datové typy parametrů, které sada testuje. Pro každý datový typ je otestováno, zda funguje vyhodnocování podmínek, výrazů pro stanovení hodnoty parametru a nalezení instance se stejnými hodnotami. Pokrytí kombinací jednotlivými testovacími případy zobrazuje tabulka 5.2. Pro zjednodušení je vždy uvedeno pouze číslo, ale celý název testu je `args_<datový typ><číslo>` (např. `args_int11`).
- `param_contractsA` testuje vliv parametrů na detekci porušení kontraktu. Jednotlivé testovací případy vždy obsahují více porušení, ale jen některá z nich splňují stanovená omezení. Výsledkem je méně detekovaných porušení než při analýze bez parametrů.
- `param_contractsB` obsahuje dvě dvojice testů, které srovnávají výsledky analýzy bez parametrů a s nimi. Monitorovaný program je vždy stejný, liší se ale specifikace kontraktů. Pokud parametry nejsou uvedeny, není nalezeno žádné porušení kontraktu, při specifikaci parametrů k porušení dojde.
- `param_contractsC` testuje případy, kdy je specifikováno více kontraktů v jednom konfiguračním souboru.
- `param_contracts` sada pro speciální případy.

Testování pomohlo nejen v ladění implementace analyzátoru, ale odhalilo i několik problémů v prostředí ANaConDA, na jejichž opravě se nyní pracuje.

5.5 Experimenty nad reálnými programy

Analyzátor Contract-validator-with-params vznikl, aby pomáhal vývojářům při tvorbě nových a mnohdy i rozsáhlých paralelních programů. Proto bylo naším cílem podrobit ho nejen testování, ale i experimentům na reálných programech, aby bylo možné srovnat výsledky

Tabulka 5.2: Pokrytí datových typů a operátorů testovacími případy.

Operátor \ Typ	`bool`	`char`	`double`	`float`	`int`	`string`	`void`
`or`	6	7	7, 17	7, 17	7, 17	7	
`and`	5	9	9, 17	9, 17	9, 17	9	
`not`	2	8	8, 17	8, 17	8, 17	8	
`<`		1	1, 17	1, 17	1, 17	1	
`>`		2, 8	2, 8	2, 8	2, 8	2, 8	
`<=`		4, 9	9	9	9	4, 9	
`>=`		3, 9	3, 4, 9	3, 4, 9	3, 4, 9	3, 9	
`==`	3	5, 6, 7	5, 7, 17	5, 7, 17	5, 6, 7, 17	5, 6, 7	3
`!=`	4	6, 7, 9	6, 7, 17	6, 7, 17	6, 7, 17	6, 7, 9	1, 2
`+`			10, 15, 18	10, 15, 18	10, 18	11	
`-`			11, 16, 18	11, 16, 18	11, 16, 18		
`*`			12, 13, 18	12, 13, 18	12, 13, 18		
`/`			14, 18	14, 18	14, 18		
`%`					15		

parametrické a bezparametrické analýzy, časové a paměťové náročnosti a aby se ověřilo, že je implementace schopná analyzovat i náročné projekty. Původní Contract-validator byl spuštěn nad sadou programů obsahujících porušení atomicity a výsledky analýzy byly uvedeny v [3]. Pro experimenty popsané v této sekci byla zvolena stejná sada programů a byla zanalyzována oběma analyzátory na stejném stroji, aby bylo možné jejich výsledky srovnávat.

V tabulce 5.3 jsou uvedeny výsledky analýzy bez parametrů pro Contract-validator (CV) a Contract-validator-with-params (CVP). Kromě času a paměti jsou uvedeny počty klauzulí ve specifikaci kontraktu (sloupec T/S) a celkový počet nalezených porušení kontraktu (PK). Počet porušení je dle očekávání téměř totožný (u dynamické analýzy jsou mírné odchylky běžné), což dokazuje korektnost implementace parametrického analyzátoru. Časová náročnost se pro analyzátor Contract-validator-with-params zvýšila v průměru o 15 % a paměťová náročnost se téměř nezměnila.

Contract-validator-with-params byl také použit pro parametrickou analýzu příkladu NASA [1]. U ostatních příkladů bohužel nebylo možné parametry definovat. Z výsledků zobrazených v tabulce 5.4 je patrné, že se časová ani paměťová náročnost příliš nezvýšila (jedná se ovšem o poměrně jednoduchý příklad), ale počet detekovaných porušení byl snížen ze 100 na 1[3]. Tento experiment dokazuje, že pomocí specifikace parametrů lze snížit počet hlášení až o desítky situací, které nezpůsobují chyby, a umožní tak vývojářům soustředit se na nalezení a opravení skutečně závažných problémů.

[3] Jednalo se o 99 falešných alarmů, kdy skutečně došlo k porušení kontraktu vzhledem k jeho definici, ale detekované situace nezpůsobovaly chybu.

Tabulka 5.3: Výsledky analýzy kontraktů pomocí nástrojů Contract-validator a Contract-validator-with-params.

Test	**Čas (s)**	**Paměť (kB)**	**T/S**	**PK**
Account (CVP)	1.33	59 106	1	318
Account (CV)	1.14	58 524	1	318
Coord03 (CVP)	2.66	62 760	8	336
Coord03 (CV)	2.36	64 868	8	344
Coord04 (CVP)	1.51	61 200	4	21
Coord04 (CV)	1.31	60 116	4	20
Local (CVP)	1.37	60 924	4	2
Local (CV)	1.23	59 580	4	2
NASA (CVP)	1.67	61 508	1	99
NASA (CV)	1.44	59 096	1	100

Tabulka 5.4: Výsledek analýzy kontraktů v programu NASA s využitím parametrů.

Test	**Čas (s)**	**Paměť (kB)**	**T/S**	**PK**
NASA s parametry	1.68	62128	1	1

Tabulka 5.5: Srovnání analýzy programu Chromium-1.

Analyzátor	**Čas (s)**	**Paměť (kB)**	**T/S**	**PK**
CV	3:19.61	1873260	1	14
CVP bez parametrů	3:29.53	1911608	1	14
CVP s parametry	4:28.41	1912916	1	2

Výše zmíněná sada programů obsahuje pouze jednoduché příklady, pro otestování implementace nad rozsáhlým projektem byl zvolen program Chromium-1 [12], jehož zdrojový kód se skládá ze 7,5 milionu řádků. Jedná se o starší verzi prohlížeče Chrome, která obsahuje porušení atomicity způsobující pád programu. Chybu je možné popsat pomocí kontraktu a dokonce umožňuje použití parametru. Výsledky obou analyzátorů jsou zobrazeny v tabulce 5.5. Ačkoli rozdíl mezi počtem detekovaných porušení není až tak znatelný jako u příkladu NASA, experiment dokázal, že parametrickou analýzu je možné použít i na rozsáhlé projekty a získat tak přesnější výsledky za rozumnou cenu.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat metodu pro dynamickou analýzu kontraktů s podporou parametrů a jejich omezeními. Byla popsána problematika kontraktů a změny a problémy, které s sebou parametry přináší. Ze tří navržených metod pro analýzu byla vybrána ta, která se jeví jako nejpoužitelnější pro analýzu reálných projektů. Metoda byla implementována jako nový analyzátor Contract-validator-with-params jako rozšíření frameworku ANaConDA. V současné době je v recenzním řízení článek [6], jehož jsem spoluautorkou, o vlastnostech tohoto prostředí.

Analyzátor byl podroben testování pomocí automatizované sady testovacích případů, což nejen pomohlo odhalit chyby v implementaci, ale v budoucnu mohou jednotlivé případy fungovat i jako regresní testy či být použity pro průběžnou integraci. Contract-validator-with-params byl také spouštěn nad stejnou sadou programů, jako původní Contract-validator bez parametrů. Experimenty ukázaly, že za cenu mírného zvýšení časové a paměťové náročnosti je možno dosáhnout mnohem přesnějších výsledků analýzy a snížit počet chybových hlášení až v řádu desítek. Analýza programu Chromium-1 potvrdila, že implementaci s podporou parametrů bude možné využít i na rozsáhlé projekty.

Základní princip analýzy parametrických kontraktů, možnosti jejich využití a výsledky, kterých bylo dosaženo, byly prezentovány na studentské konferenci Excel@FIT [16]. Práce byla oceněna odborným panelem, jedním z partnerů i odbornou veřejností.

V budoucnu bychom se chtěli zaměřit především na další experimenty a nasazení analyzátoru do praxe, aby skutečně mohl pomáhat vývojářům při jejich práci. Dá se očekávat, že postupem času vyvstanou nové požadavky na analýzu či specifické případy, pro které bude potřeba analyzátor přizpůsobit, a proto bude třeba implementaci nadále udržovat a rozšiřovat.

Literatura

[1] Artho, C.; Havelund, K.; Biere, A.: High-Level Data Races. In *Journal on Software Testing, Verification and Reliability (STVR)*, ročník 13(4), 2003, ISSN 0960-0833, s. 207–227, doi:10.1002/stvr.281.

[2] Desnoyers, M.; McKenney, P. E.; Stern, A. S.; aj.: User-Level Implementations of Read-Copy Update. In *IEEE Transactions on Parallel and Distributed Systems*, ročník 23(2), Feb. 2012, ISSN 1045-9219, s. 375–382, doi:10.1109/TPDS.2011.159.

[3] Dias, R. J.; Ferreira, C.; Fiedor, J.; aj.: Verifying Concurrent Programs Using Contracts. In *2017 IEEE International Conference on Software Testing, Verification and Validation*, 2017, ISBN 978-1-5090-6031-3, s. 196–206, doi:10.1109/ICST.2017.25.

[4] Edelstein, O.; Farchi, E.; Goldin, E.; aj.: Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, ročník 15(3-5), 2003: s. 485–499, ISSN 1532-0626, doi:10.1002/cpe.654.

[5] Fiedor, J.; Křena, B.; Letko, Z.; aj.: A Uniform Classification of Common Concurrency Errors. *Lecture Notes in Computer Science*, ročník 2012, č. 6927, 2012: s. 519–526, ISSN 0302-9743.

[6] Fiedor, J.; Mužikovská, M.; Smrčka, A.; aj.: Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. Odesláno do recenzního řízení konference ISSTA 2018.

[7] Fiedor, J.; Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of 3rd International Conference on Runtime Verification—RV'12*, ročník 7687 of LNCS, Istanbul, Turkey, 2012, ISSN 0302–9743, s. 35–41, doi:10.1007/978-3-642-35632-2_5.

[8] Fiedor, J.; Vojnar, T.: Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1456-5, s. 36–46, doi:10.1145/2338967.2336813.

[9] Flanagan, C.; Freund, S.: FastTrack: efficient and precise dynamic race detection. In *Communications of the ACM*, ročník 53(11), 01 November 2010, ISSN 0001-0782, s. 93–101, doi:10.1145/1839676.1839699.

[10] 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, PASTE '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0082-7, s. 1–8, doi:10.1145/1806672.1806674.

[11] Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, London, UK, UK: Springer-Verlag, 2000, ISBN 3-540-41030-9, s. 245–264.

[12] Jalbert, N.; Pereira, C.; Pokam, G.; aj.: RADBench: A Concurrency Bug Benchmark Suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, Berkeley, CA, USA: USENIX Association, 2011, s. 2–2.

[13] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, ročník 21(7), 01 July 1978, ISSN 0001-0782, s. 558–565, doi:10.1145/359545.359563.

[14] Letko, Z.; Vojnar, T.; Křena, B.: AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-052-4, s. 7:1–7:10, doi:10.1145/1390841.1390848.

[15] McKenney, P. E.; Slingwine, J. D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems*, October 1998, s. 509–518.

[16] Mužikovská, M.: Dynamická analýza parametrických kontraktů pro paralelismus. In *Excel@FIT, Studentská konference*, 2018.

[17] Nethercote, N.; Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '07*, ročník 42, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-633-2, s. 89–100, doi:10.1145/1250734.1250746.

[18] Rinard, M. C.: Analysis of Multithreaded Programs. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, London, UK: Springer-Verlag, 2001, ISBN 3-540-42314-1, s. 1–19.

[19] 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*, SOSP '97, New York, NY, USA: ACM, 1997, ISBN 0-89791-916-5, s. 27–37, doi:10.1145/268998.266641.

[20] Šoková, V.: *Analýza práce s dynamickými datovými strukturami v C programech*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.

[21] Williams, A.: *C++ concurrency in action : practical multithreading*. New York : Manning Publications Co., 2012, ISBN 978-1-933988-77-1.

Příloha A

Obsah přiloženého paměťového média

Adresářová struktura:

- `anaconda/`
  - `analysers/contract-validator-with-params/` – Zdrojové kódy analyzátoru Contract-validator-with-params.
  - `framework/` – Zdrojové kódy frameworku ANaConDA.
  - `tests/framework/monitoring/` – Adresář se všemi testy pro framework ANaConDA spolu s testovací sadou vytvořenou pro Contract-validator-with-params.
- `demo/` – Jednoduché příklady pro demonstraci použití a výsledků analyzátoru.
- `diff/` – Výsledky programu `diff` nad změněnými a původními soubory frameworku.
  - `diff_list.txt` – Seznam souborů, které byly pro implementaci této práce do frameworku přidány, nebo byly upraveny.
- `doc/index.html` – HTML dokumentace analyzátoru automaticky vygenerovaná programem `doxygen`.
- `README` – Soubor s popisem obsahu média.
- `xmuzik05.pdf` – PDF této technické zprávy.
- `xmuzik05-src/` – Zdrojové kódy této technické zprávy pro LaTeX.