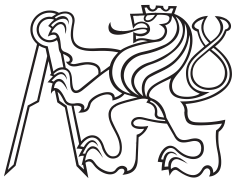
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Nástroj pro generování jednotkových testů pro .NET pomocí model checkeru

Bc. Benjamin Hejl

Vedoucí: Ing. Karel Frajták, Ph.D.
Obor: Otevřená informatika
Studijní program: Softwarové inženýrství
Leden 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hejl Benjamin** Personal ID number: **466023**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

II. Master’s thesis details

Master’s thesis title in English:

Unit tests generation tool for .net powered by model checker

Master’s thesis title in Czech:

Nástroj pro generování unit testů pomocí model checkingu pro .net

Guidelines:

Unit testing legacy code is a time consuming task. The system can be tested on a symbolic level without executing it or on a concrete level when the system is executed and the tool explores all possible paths through the system. This type of testing is called concolic testing (concrete + symbolic). Unit tests can be created using a tool called Pex. However, in the real world, this tool runs into problems caused by the complexity of the tested code that calls a database or uses configuration objects. Another tool, Microsoft Fakes, can help with this problem, the tool has capabilities to create detours. With detours the behaviour of the system can be specified (similarly to how mocks work). For example a call to get the current date/time value can be "detoured" to return a specific value.
The goal of this project is to create a tool that will be able to generate unit tests for a given method using artificially generated mocks with a help provided by the developer:
1. The tool will be given a method that should be tested with a configuration setting of the method input parameters
2. The tool will use the concolic engine to cover all possible paths in the tested method and discover all possible combinations of input parameters that will achieve the highest coverage of the tested method.
3. The tool will generate a set of unit tests (their code) that will, when executed, cover the tested method. An appropriate templating engine should be chosen to generate those tests. These tests will be added to the original project to test the legacy system.
4. Certain limitations of the solver can be bypassed with the help of the developer/tester - for example accessing the database or working with variables of an interface type. The tool will provide a mechanism that will let the developer/tester define how the system should behave in these situations, for example mocks can be used when working with variables of an interface type.
Demonstrate the tool on various sample methods with parameters of value type, reference and interface type and on methods with code where the developer/tester' hint will help.

Bibliography / sources:

Pex – White Box Test Generation for .NET
- https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/
Isolate code under test with Microsoft Fakes
- https://docs.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes?view=vs-2019
Concolic and symbolic testing
- http://zbchen.github.io/Papers_files/icse2018-2.pdf
- https://www.cs.cmu.edu/~aldrich/courses/17-355-17sp/notes/lec-concolic-sen.pdf
- http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security/ConcolicExecution.pdf

Name and workplace of master's thesis supervisor:

Ing. Karel Frajták, Ph.D. System Testing IntelLigent Lab FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **02.02.2022** Deadline for master's thesis submission: **10.01.2023**

Assignment valid until: **30.09.2023**

Ing. Karel Frajták, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Poděkování

Chtěl bych poděkovat svému vedoucímu práce, Karlu Frajtákovi, za trpělivost při revizi kódu a v nepřiměřeně velkých dávkách a své rodině a přátelům, kteří mě v mých studiích dlouhodobě podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně pod vedením Ing. Karla Frajtáka, Ph.D., a že veškerá použitá literatura je uvedena.

V Praze, dne 6. ledna 2023

Abstrakt

Jednotkové testy slouží jako základ testování softwaru a zdroj informací o fungování testovaného kódu, což je důležité pro jeho další rozvoj a vylepšování. Manuální formulace testů, které plně pokryjí testovanou metodu a ukáží všechny její vlastnosti je náročná a k chybám náchylná práce.

Nástroj *dnWalker* využívá konkolickou exekuci – kombinaci symbolické a konkrétní exekuce – k automatickému nalezení testovacích případů pro kód napsaný v .NET jazycích. Z testovacích případů vygeneruje soubor testů, který zajistí vysoké pokrytí kódu. Pro procházení a odhalení podmínek pro všechny cesty v kódu využívá *dnWalker* kombinaci teorií splnitelnosti a separační logiky.

Kromě konkrétních dat je *dnWalker* schopný pracovat i s abstraktními objekty. Pro izolaci jednotkových testů pak používá vygenerované testové dvojníky, přičemž uživatel může specifikovat konkrétní izolační framework.

Třebaže *dnWalker* je nástroj s omezenou funkcionalitou, experimentální výsledky ukazují, že při práci s jak primitivními tak i komplexními daty dosahuje srovnatelných anebo lepších výsledků než zavedené a komerční nástroje.

Klíčová slova: jednotkové testy, konkolická exekuce, testování softwaru, .NET, ověřování modelu, verifikace softwaru

Abstract

Unit tests are the base of software testing, part of the software verification and validation process, and a source of information about functionality of the code under test, an important element for its refactorization. Manual formulation of test cases which cover the software under test and demonstrate all of its properties is hard and error-prone work.

The tool *dnWalker* uses concolic execution – a combination of symbolic and concrete execution – for automatic test cases discovery for code written in .NET languages. From these test cases it generates a test suite which ensures high code coverage. In order to test and discover all conditions for all paths through the code, *dnWalker* uses a combination of satisfiability modulo theories and separation logic.

Apart from concrete data *dnWalker* is capable of working with abstract objects. Isolation of the unit tests is achieved using generated test doubles while using concrete isolation framework provided by the user.

Although *dnWalker* isn't a full-featured tool, experimental results show that when working with both primitive and dynamic data *dnWalker* achieves comparable or better results than matured and commercial tools.

Keywords: unit testing, concolic execution, software testing, .NET, model checking, software verification

Title translation: Unit Tests Generation Tool for .NET Powered by Model Checker

Obsah

Obrázky

Tabulky

Programy

Algoritmy

Kapitola 1

Úvod

Jednotkové testy jsou základním kamenem pro testování softwaru a zajištění jeho kvality. Kromě ověření fungování nejmenších částí zdrojového kódu poskytují dobře zformulované testy cenné informace o chování testovaného softwaru, což je nezbytné pro jeho údržbu a vývoj.

Nicméně psaní jednotkových testů, které vhodně pokryjí testovaný kód, je náročná a zdlouhavá práce. Tester musí do hloubky analyzovat zdrojový kód, aby objevil všechna jeho zákoutí a stavy a poté správně zformulovat vstupní a očekávaná výstupní data. Kromě vysoké pravděpodobnosti přehlédnutí chování (typickým případem je neošetření prázdné reference – `null`[27]) nemusí být zdrojový kód vůbec k dispozici, například při práci s legacy kódem, nebo je zastaralý a úkolem je jej vhodně refaktorovat. Jednotkové testy jsou pak zárukou zachování funkcionality.

Manuálního tvorba testů je v kontextu těchto problémů příliš neefektivní. Širokou škálu technik a metod pro automatizovanou analýzu softwaru nabízí verifikace a validace. Mezi ně patří *ověřování modelu* (*model checking*), který ověřuje vlastnosti softwaru vůči jeho modelu, *symbolická exekuce* (*symbolic execution*), která prozkoumává dostupné stavy programu pomocí symbolických vstupů a omezujících podmínek, které vznikají na rozhodovacích bodech v programu, a *konkolická exekuce* (*concolic execution*), která symbolickou exekuci kombinuje s konkrétní exekucí, pomocí jejíž výsledků zefektivňuje proces symbolické exekuce a poskytuje další možnosti.

Princip symbolické a konkolické exekuce je velmi jednoduchý (první výzkum symbolické exekuce byl publikován již v roce 1976 [55]), ale provedení čelí mnoha výzvám. Ty jsou spojené s komplexitou programů (velké množství větvení, cykly a rekurze) a schopnostmi vyjádření a ověření jeho vlastností (nelineární operace a vlastnosti dynamických struktur v paměti na haldě). Nicméně s rozvojem *SMT solverů*[1], nástrojů na rozhodnutí splnitelnosti matematické formule, vznikla od začátku 21. století řada nástrojů symbolické

[1] Vzhledem k absenci rozumného českého ekvivalentu je použit termín *solver*.

a konkolické exekuce, které cílí na různé programovací jazyky nebo ekosystémy, například *CUTE*[80], *JDART*[62], *SAGE*[43] nebo *Pex* (*IntelliTest*)[91].

Jednou ze zásadních výzev pro uvažování nad stavem programu je vyjádření omezení nad tvarem, velikostí a obsahem dynamických dat. Taková omezení a zejména rozhodování o nich musí brát v potaz mimo jiné *aliasing ukazatelů* (*pointer aliasing*) a mutaci dat v průběhu exekuce. Za účelem formulace těchto omezení vzniklo několik teorií, mezi které patří například HEX (*heap exploration logic*)[16] nebo *separační logika*(*separation logic*)[77].

V této práci je představen nástroj *dnWalker*[104], který využívá konkolickou exekuci automatické generování jednotkových testů pro .NET programy.

Nástroj *dnWalker* má splňovat následující základní požadavky:

1. Vstupem bude testovaná metodu společně s konfigurovatelnými parametry metody.

2. Pomocí konkolického průzkumu nástroj prohledá všechny průchody testovanou metodou a objeví kombinace vstupních parametrů pro zajištění nejvyššího pokrytí testované metody.

3. Výstupem bude zdrojový kód souboru jednotkových testů, jejichž exekuce pokryje testovanou metodu. Vhodné šablony budou použity pro generování testů.

4. Určité limitace nástroje mohou být řešeny za pomoci testera, například přístup k databázi přes rozhraní. Nástroj poskytne mechanismus pomocí kterého může tester nastavit, jak se má nástroj chovat v takových případech, například pomocí mockování rozhraní.

Kromě základních požadavků byl vytyčen také následujíc cíl:

5. Vygenerované testy nebudou závislé na specifickém testovacím frameworku a dalších testovacích knihovnách. Tester bude mít možnost použít kombinaci testovacího frameworku a knihoven dle svých preferencí. Zároveň nebudou závislé na nástroji samotném.

Organizace textu

Zbytek textu je organizován následujícím způsobem. V následující kapitole (2) je rozvedena problematika testování softwaru se zaměřením na jednotkové testy. V kapitole 3 je ukázán motivační příklad, který ukazuje, jak nástroj vytvořený v rámci této práce pomáhá testerovi při formulaci jednotkových testů.

Kapitola 4 se věnuje verifikaci softwaru. Uvádí metody důležité v kontextu této práce společně s úvodem do logiky používané těmito metodami. V kapitole 5 je představen stávající vývoj v těchto oblastech.

V kapitole 6 je popsána implementace nástroje *dnWalker*. Představení schopností nástroje *dnWalkeru* a jejich srovnání s podobnými nástroji je v kapitole 7. Zhodnocení práce a nástin některých dalších kroků ve vývoji nástroje jsou v závěrečné kapitole 8.

Kapitola 2

Testování softwaru

Testování softwaru je „proces, nebo série procesů, navržených k zajištění toho, že software provádí to, k čemu byl navržen, a zároveň neprovádí nic, k čemu nebyl zamýšlen“[67]. Pokud software nesplňuje některou z těchto podmínek, znamená, že v něm je nějaká chyba. Testování má za úkol tyto chyby odhalit.

Testování ale nemusí všechny chyby odhalit, protože „testování programu lze použít pro dokázání přítomnosti chyb, ale nikdy pro dokázání jejich nepřítomnosti“[33].

Testování softwaru je náročný proces, který od testera vyžaduje vhodný přístup. Podle [67] se často projevují mylné přístupy k testování, které spočívají v „dokázání, že v programu nejsou chyby“, „ukázání, že program provádí očekávané funkce správně“ nebo „ujištění, že program provádí to, k čemu je určen“.

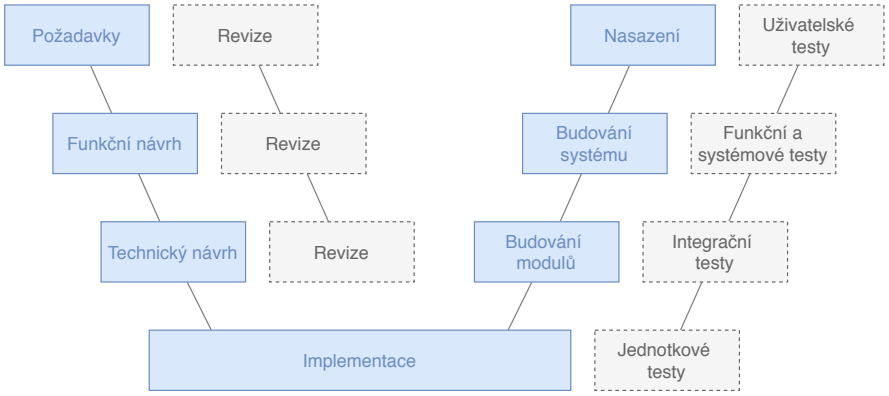
Oproti tomu staví přístup založený na tvrzení „program obsahuje chyby“. Testování je potom „proces spouštění programu s úmyslem najít chyby“[67].

Testování softwaru je součástí *verifikace* a *validace* softwaru[86]. Podle [11] se jedná o odpověď na dvě otázky:

1. Budujeme správný produkt? (*validace*)
2. Budujeme produkt správně? (*verifikace*)

Testování softwaru pomáhá odpovědět na obě tyto otázky[86].

Testování je nezbytnou součástí vývoje softwaru, ať už v sekvenčním (viz obrázek 2.1) nebo agilním vývoji (viz obrázek 2.2). Podle zaměření můžeme testy dělit do skupin (viz tabulka 2.1).



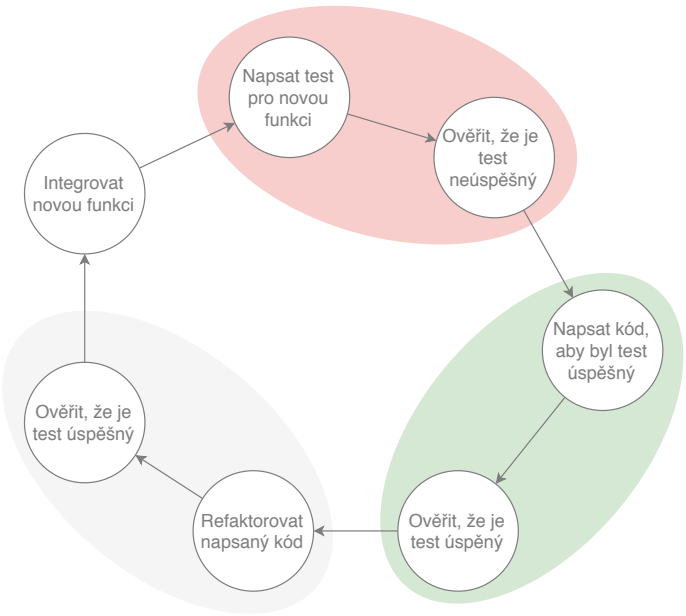
Obrázek 2.1: Sekvenční vývoj softwaru: W-model.

Název	Popis
Akceptační / uživatelské testy	Ověřují naplnění požadavků softwaru z hlediska koncového uživatele.
Systémové testy	Ověřují naplnění požadavků softwaru.
Funkční testy	Ověřují vnější specifikaci softwaru.
Integrační testy	Ověřují návrh systému a strukturu softwaru (e.i. integrace modulů).
Jednotkové / modulové testy	Ověřují specifikaci a korektnost jednotek kódu (modul, třída, procedura).

Tabulka 2.1: Dělení testů dle [67]

Testy pak můžeme poskládat to tzv. *testovací pyramidy* (viz obrázek 2.3)[99]. Na obrázku testovací pyramidy jsou naznačené 2 škály: *izolace* vs *integrace* a *rychlost*. Testy na nižší úrovni jsou zpravidla jednodušší, jejich průběh je rychlejší a je jich větší množství. Zároveň se testují malé části kódu v izolaci. Testy na vyšší úrovni jsou naopak komplikovanější a vyžadují inicializaci více nebo všech částí softwaru. Jejich průběh je proto pomalejší a bývá jich méně a jsou náchylnější na rozbití.

Podle Boehmova zákona podloženého empirickými daty roste cena opravy chyby exponenciálně s fází nalezení[10]. Naproti tomuto fenoménu stojí pravděpodobnost zanesení chyby a její nalezení, která klesá, respektive stoupá, s fází vývoje (viz obrázek 2.4)[10]. Jinak řečeno, nejvíce chyb se zanese v počátečních fázích vývoje softwaru, ale odhalí se až v konečných fázích, kdy poškodí koncového uživatele a jejich oprava je nejdražší. Také hrozí, že oprava chyby způsobí nové chyby, neošetří všechny případy nebo se znovu objeví dříve opravená chyba (regrese).



Obrázek 2.2: Agilní vývoj softwaru: TDD-model (*test driven development*).

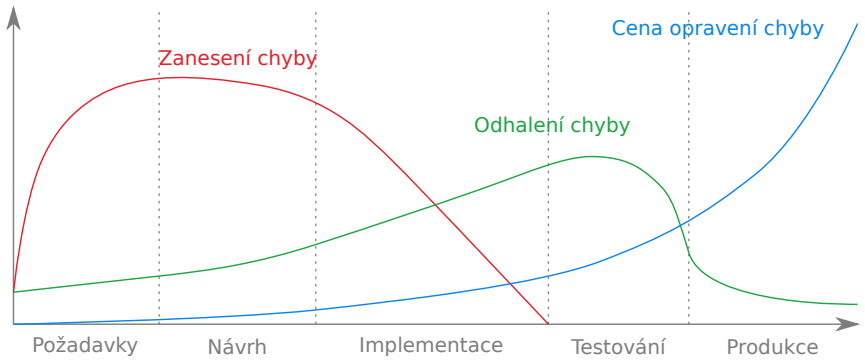
2.1 Softwarové testy

Základní jednotkou testování softwaru je testovací případ. Ten obsahuje informace pro provedení a ohodnocení (chyba byla, respektive nebyla, nalezena) jednoho testu. Testovací případ musí obsahovat 3 základní prvky:

1. vstupní podmínky - předdefinovaný stav systému před testem, testovací data,
2. testovaný proces - ověřovaná sekvence kroků v testovaném softwaru,
3. výstupní podmínky - očekávaný stav na konci testu, očekávaný výstup

Jednou z výzev testování softwaru je odhalení co největšího množství chyb s ohledem na všechny možné situace a ekonomickou proveditelnost[67]. Existují dva základní přístupy k testování, které vychází z pohledu na testovaný software.

Obrázek 2.3: Testovací pyramida podle [99] a [67].



Obrázek 2.4: Boehmův zákon a vývoj zavedení a odhalení chyby v čase[10].

Black-box testování. S programem se pracuje jako s černou skříňkou, jejíž vnitřní strukturu či stavy tester nevidí, pouze vstupy a výstupy. Primárním zdrojem informací je proto specifikace softwaru. Do této testovací strategie spadají uživatelské testy, analýza hraničních hodnot nebo kauzální grafy[67].

White-box testování. Opakem black-box přístupu je analýza vnitřní struktury. White-box testování vychází přímo z testovaného programu a snaží se otestovat všechny jeho stavy[67].

Zatímco white-box testování se zaměřuje na odhalení chyb způsobených nekorektním kódem – *verifikace* – (například vyhození neočekávané výjimky), pro nalezení chyb vycházející z nesplnění požadavků – *validce* – (například chybějící funkcionalita) je nutné použít black-box testování.[67]

Gray-box testování. Některé metody kombinují white-box a black-box přístup a říká se jim proto metody *gray-box* testování. Jsou založené na částečné znalosti implementace (použité algoritmy, datové struktury atp.) a znalosti požadavků[96]. Mezi ně patří třeba *maticové testování* (*matrix testing*), které upravuje množství proměnných či entit v softwaru, aby odpovídalo požadavkům, čímž snižuje jeho komplexitu[96].

2.2 Jednotkové testy

Jednotkové testy, které jsou předmětem této práce, leží v samém základě testovací pyramidy a tvoří tak jednu ze základních komponent testování softwaru. Cílem jednotkového testu je ověřit nejmenší možnou jednotku kódu. Zpravidla je jednotkových testů mnoho, aby dostatečné otestovalo testovaný kód a zároveň mají být velmi rychlé. V případě procedurálního programování se jedná o proceduru, v případě objektově orientovaného programování je občas vhodné jednotku rozšířit na celou třídu, nebo její část, protože data a metody třídy mohou být silně provázané[86].

2.2.1 Struktura jednotkových testů

Jednotkový test se, podobně jako jiné testy, skládá ze tří[86] (čtyř[66]) fází: uspořádání (*arrange*), jednání (*act*), ověření (*assert*, *verify*) a demontáž/úklid (*tear down*). Této struktuře se, podle anglických názvů, říká AAA.

Uspořádání. Nastaví a inicializuje testovaný software, včetně vstupů a prostředí[66]. Zpravidla se jedná o nejkomplikovanější část testu. Zejména pokud má testovaný software nějaké závislosti, pro které se pak musí připravit testoví dvojníci[66] (viz sekce 2.2.2). V příkladě v programu 2.1 se vytvoří testový dvojník pro rozhraní `IBar` pomocí třídy `Mock<IBar>`.

Vykonání. Nejjednodušší část testu - spustí se testovaný software a získá se jeho výstup. Výstup může být návratová hodnota, ale také změna stavu vstupů nebo vyhozená výjimka[66]. V příkladě v programu 2.1 se uloží návratová hodnota testované metody.

Ověření. Druhá komplikovaná část testu, protože se musí správně interpretovat výsledky z předchozí fáze – porovnání návratové hodnoty či vyhozené výjimky s očekávanými výstupy anebo ověření změny stavu. Společně s první fází je nejvíc náchylná na chyby. Důležitou součástí ověření je i zpětná vazba. Pokud test selhal, je nutné předat informaci proč[66].

Úklid. Tato fáze není nutná, pokud předchozí fáze nějakým způsobem neovlivní širší prostředí. Sem patří uvolnění zdrojů atp. Při psaní testu je nutné zařídit, aby tato fáze proběhla v případě úspěšného i neúspěšného průběhu testu[66]. Zpravidla se provádí na úrovni testovacího frameworku, proto příklad 2.1 tuto fázi neobsahuje.

```
1 public void FooReturnValueSchema_2()
2 {
3     // Arrange input model heap
4     SimpleMethod simpleMethod1 = new SimpleMethod();
5     simpleMethod1.Value = 0;
6     Mock<IBar> iBar1_mock = new Mock<IBar>();
7     IBar iBar1 = iBar1_mock.Object;
8     iBar1_mock
9         .Setup(o => o.GetValue())
10        .Returns(0);
11
12    // Arrange method arguments
13    SimpleMethod @this = simpleMethod1;
14    IBar bar = iBar1;
15
16    int result = @this.Foo(bar);
17    Assert.Equal(5, result);
18
19 }
```

Program 2.1: Příklad jednotkového testu vygenerovaného *dnWalkerem*.

2.2.2 Testoví dvojníci

Důležitým faktorem při navrhování jednotkových testů je izolace testovaného kódu[66]. Jednotkový test má za úkol ověřit vlastnosti daného modulu nezávisle na jiných modulech[67]. Pro zajištění této nezávislosti používáme tzv. *testové dvojníky* (*test doubles*)[66]. Testový dvojník napodobuje pozorovatelné chování ostatních modulů, které je však přesně definované podle požadavků na testování. Kromě naprosté kontroly nad chováním ostatních komponent, se odeberou nechtěné vedlejší efekty[66]. Například při testování komponenty rezervačního systému je ve většině jednotkových testů nechtěné, aby systém poslal konfirmační email. Testoví dvojníci se mohou implementovat manuálně,

nebo použít *izolační framework* (*isolation framework*), pomocí kterého je lze definovat dynamicky[37].

Dummy object. Minimální možná implementace nutná pro bezchybnou kompilaci testovacího kódu. Představuje pouze náhražku bez jakékoliv funkcionality[66]. Program 2.2 ukazuje tento typ pomocí implementace `DummyFoo`, jehož metoda `Bar` pouze zavolání vyhodí výjimku.

Test stub. Metody mají danou návratovou hodnotu bez vedlejších efektů[66]. V programu 2.2 se jedná o `StubFoo`, která v tomto případě vrací číslo 5. Také se může jednat o vyhození dané výjimky atp.

Test spy. Rozšiřuje Test Stub, přičemž přidává funkcionalitu, která umožňuje sledovat, jak se s objektem zacházelo – které metody a s kterými argumenty se volaly[66]. V programu 2.2 se argumenty ukládají do seznamu, který je možné získat pomocí nové metody `SpyFoo::GetBarInvocations`.

Mock object. Rozšiřuje Test spy. Kromě specifikace chování a sledování použití také obsahuje ověření, že byl použit správně[66]. V příkladě v programu 2.2 se jedná o třídu `MockFoo`, která očekává, že metody `IFoo::Bar` bude zavolána právě dvakrát s argumenty 3 a -1.

Fake object. Představuje zjednodušenou implementaci[66]. V příkladě v programu 2.2 vrací implementace `FakeFoo::Bar` výraz $x(x-1)$. V praxi se jedná například o *in–memory* databázi, která nahrazuje reálnou SQL databázi.

V praxi se pojem testový dvojník nepoužívá, a výše popsané techniky se slučují do jedné – *mock*[37], protože poskytuje funkcionalitu všech ostatních testových dvojníků. Pouze v případě *Fake Object* mohou nastat komplikace, protože umožňuje chování s vedlejšími efekty. Nicméně rozšířené implementace izolačních frameworků to zpravidla umožňují.

```
 1 interface IFoo {
 2   int Bar(int x);
 3 }
 4
 5 class DummyFoo : IFoo {
 6   public int Bar(int x) => throw new NotImplementedException();
 7 }
 8
 9 class StubFoo : IFoo {
10   public int Bar(int x) => return 5;
11 }
12
13 class FakeFoo : IFoo {
14   public int Bar(int x) => x*(x - 1);
15 }
16
17 class SpyFoo : IFoo {
18   List<int> _barArgs = new();
19
20   public int Bar(int x) {
21     _barArgs.Add(x);
22     return 5;
23   }
24
25   public IEnumerable<int> BarInvocations => _barArgs;
26 }
27
28 class MockFoo : SpyFoo {
29
30   public bool Verify() => BarInvocations.SequenceEquals(3, -1);
31 }
```

Program 2.2: Příklad manuálně implementovaných testových dvojníků.

Implementace

Pro .NET (viz část 6.1.1) je implementace těchto knihoven založená na 2 základních způsobech: vytváření *dynamických proxy objektů* a ovlivnění běhu aplikace přes CLR (*common language runtime*) API. První způsob je jednodušší, ale umožňuje nahradit pouze virtuální a abstraktní metody – v případě náhrady abstraktní třídy – nebo metody rozhraní (*interface*) – v případě náhrady rozhraní. A v případě náhrady třídy je potřeba poskytnou argumenty pro konstruktor. Knihovna vygeneruje nový typ, který přetěžuje či implementuje potřebné metody tak, jak nadefinoval uživatel. Jedná se například o knihovny jako *moq*[110] nebo *FakeItEasy*[106].

Druhý způsob pracuje přímo s implementací .NET, respektive CLR, který

provádí mimo jiné JIT (*just in time*) kompilaci. Knihovna odchytí požadavky na volání metod (vzniklé instrukcemi `call` a `callvirt`) a vloží místo nich falešné odbočky, které se chovají dle potřeb nastavených testerem. Jedná se o mocnější nástroj, protože je možné upravit chování metod, které nelze přetížit nebo jsou statické. Díky tomu je možné nastavit chování i nedeterministických metod vracející, například aktuální datum a čas či náhodná čísla. Zároveň jsou ale závislé na konkrétní implementaci CLI (.NET). Tento přístup využívají knihovny jako *Isolator*[108] nebo *MSFakes*[111].

Některé nástroje také vyrábí testové dvojníky pomocí technologie *remoting proxy* (*remote proxy*). Ta umožňuje vytvořit proxy objekt, který komunikuje s reálnou instancí pomocí zpráv, přičemž reálná instance může existovat v jiném procesu nebo někde v síti. Nástroj tuto komunikaci odchytí a použije jí k zajištění požadovaného chování[112].

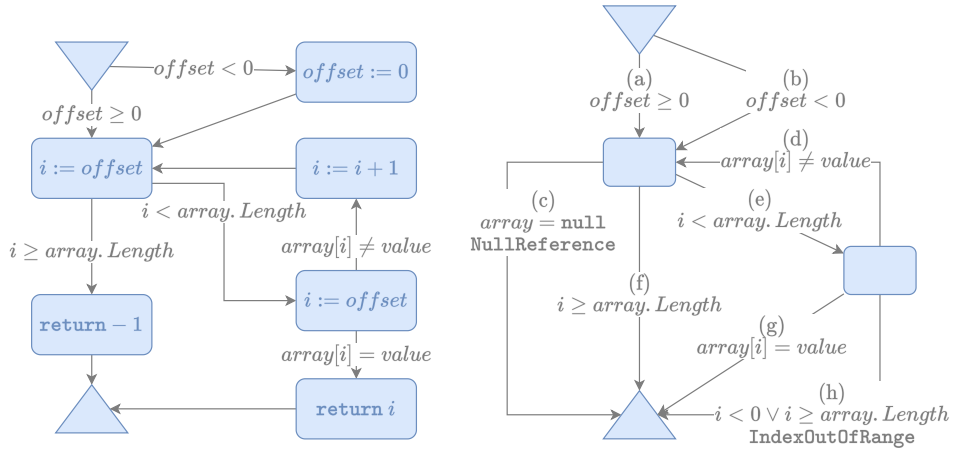
2.2.3 Metrika testování

Podle čeho může tester usoudit, že má vhodně vytvořené jednotkové testy? Respektive že nalezené testovací případy dostatečně ověří vlastnosti softwaru? Jedním z nejpoužívanějších měřítek „protestování“ kódu, je jeho pokrytí. Nabízí se sledovat pokrytí instrukcí, větvení a průchodů. Počet průchodů programem ale roste exponenciálně s jeho složitostí (větvení, cykly, rekurze atp), což zásadně komplikuje práci testera a dělá úplné pokrytí (všechny možné průchody programu) stěží neproveditelné.

Graf toku řízení (CFG)

Pro analýzu struktury metody se využívá tzv. *graf toku řízení* (*control flow graph*, CFG), který ilustruje vztahy mezi jednotlivými instrukcemi a rozhodovací body v metodě. Problémy pokrytí kódu (instrukce, větvení, průchody) lze pak převést na problémy pokrytí grafu (uzle, hrany, cesty) a pro jejich řešení můžeme využít teorii grafů.

CFG je orientovaný multigraf se vstupním a výstupním uzlem (počet vstupních, respektive, výstupních hran je 0). Každá instrukce je asociovaná s uzlem v CFG. Hrana (s_1, s_2) existuje právě tehdy, když instrukce s_2 může být spuštěna okamžitě po instrukci s_1. Navíc existuje hrana mezi vstupním uzlem n_{in} a uzlem pro první instrukci metody. A poté mezi každým uzlem pro instrukci, která způsobí ukončení metody a výstupním uzlem n_{out}[44].



(a) : Graf toku řízení (CFG) bez výjimek. **(b) :** Rozhodovací graf s výjimkami, nerozlišuje chybové a normální koncové stavy.

Obrázek 2.5: CFG a rozhodovací graf s výjimkami pro program 2.3.

CFG bývá rozšiřován, jako například v této práci, i o hrany které popisují efekt vyhození a zachycení výjimek (více v části 6.2.1). Také bývá naopak zjednodušován odebráním uzlů, které mají pouze jednu vstupní a jednu výstupní hranu, protože pro analýzu CFG nejsou důležité, čímž vzniká tzv. *rozhodovací graf* (*decision graph*), kde každý uzel představuje rozhodovací bod[44]. Ukázka těchto procesů pro metodu v programu 2.3 ukazuje obrázek 2.5.

Touto metodou obecně vzniká multigraf. Pokud například metoda obsahuje instrukci `switch` s více případy, které míří na stejnou instrukci, nebo až při zjednodušení CFG, jak ukazuje obrázek 2.5.

```
 1 int IndexOf(int value, int[] values, int offset) {
 2   if (offset < 0) {
 3     offset = 0;
 4   }
 5
 6   for (int i = offset; i < values.Length; ++i) {
 7     if (values[i] == value) {
 8       return i;
 9     }
10   }
11   return -1;
12 }
```

Program 2.3: Metoda `IndexOf` s ofsetem.

Z hlediska pokrytí CFG je také důležitá *základní cesta* (*prime path*), což je jednoduchá cesta (nenavštíví jeden uzel vícekrát, ledaže v něm začíná a končí), která není obsažena v jiné základní cestě (je maximálně

Název	Popis
	Pro množinu cest skrz CFG platí:
Uzly	každý uzel se nachází na alespoň jedné cestě.
Hrany	každá hrana se nachází na alespoň jedné cestě ($TDL = 1$).
TDL= n, (úroveň hloubky testování)	pro každý rozhodovací bod D a každou sekvence n hran (S_D), takovou, že první hrana je vstupní hrana do D, existuje cesta p taková, že obsahuje sekvenci S_D.
Základní cesty	každá základní cesta je obsažena v alespoň jedné cestě.
Cesty	všechny cesty jsou obsažené v množině cest.

Tabulka 2.2: Kritéria pokrytí CFG , od nejsilnějšího po nejslabší. Silnější kritéria zajistí slabší kritéria.

dlouhá) a nemusí obsahovat vstupní a výstupní uzel. Pro CFG na obrázku 2.5 je množina základních cest (vyjádřených jako posloupnost hran) $\{ac, af, bc, bf, ed, de, dc, df, aeg, aeh, beg, beh\}$.

Tabulka 2.2 pak obsahuje základní kritéria pokrytí CFG.

Pokrytí podmínek a rozhodnutí

Větvení v programu se tvoří pomocí tzv. *rozhodovacích bodů* (*decision point*). Jedná se o instrukce, ze kterých se řízení může přesunout na více míst v metodě, přičemž přesun se volí na základě logického výrazu spojené s rozhodovacím bodem. Pro pokrytí všech větvení stačí poměrně málo testů (zpravidla se jedná o binární větvení, tudíž postačí 2 testy). Nicméně v případě, že je výraz komplexní, je vhodné zvolit testovacích případů více.

Uvažujme rozhodovací bod, jehož podmínka je $(\alpha \vee \beta)$. Pokrytí všech hran, respektive rozhodnutí (viz níže) je zajištěno dvěma testy, například $(\alpha = \textbf{false}, \beta = \textbf{true})$ a $(\alpha = \textbf{false}, \beta = \textbf{false})$. V implementaci nicméně může být chyba, která spočívá ve vlivu tvrzení α na výsledek tohoto rozhodnutí a v takto zvolených případech se jeho efekt neprojeví kvůli sémantice disjunkce.

Proto se pracuje s variantami pokrytí *(atomických) podmínek* (primitivní logický výraz, který neobsahuje logický operátor, například $x \leqslant 5$) a *rozhodnutí* (logický výraz, který se skládá z minimálně jedné podmínky a jednoho či více logických operátorů, například $x \leqslant 5 \vee y > 3$).

Vlastnost	DC	CC	C/DC	MC/DC	MCC
Každý vstupní a výstupní bod je spuštěn alespoň jednou	✓	✓	✓	✓	✓
Pro každé rozhodnutí bylo dosaženo všech možných výsledků alespoň jednou.	✓		✓	✓	✓
Pro každou podmínku v rozhodnutí bylo dosaženo všech možných výsledků alespoň jednou.		✓	✓	✓	✓
Pro každou podmínku v rozhodnutí bylo ukázáno jak nezávisle upravuje výsledek rozhodnutí[a].				✓	✓[b]
Každé kombinace výsledků podmínek bylo dosaženo alespoň jednou.					✓

[a] Existují logické výrazy, ve kterých každá podmínka nemá nezávislý efekt[46].
[b] Je zajištěno implicitně pokrytím všech možných kombinací.
DC: *decision coverage*
CC: *condition coverage*
C/DC: *decision/condition coverage*
MC/DC: *modified decision/condition coverage*
MCC: *multiple condition coverage*

Tabulka 2.3: Kritéria pokrytí podmínek a rozhodnutí, převzato z [46].

V tabulce 2.3 je popsáno 5 standardních kritérií založených na podmínkách a rozhodnutích. Kritérium MC/DC je třeba používané v systémech NASA[46], zejména kvůli dobrému poměru síle pokrytí a množství testů. Pro rozhodnutí o n podmínkách stačí v MC/DC $n + 1$ testovacích případů – na rozdíl od nejsilnějšího MCC, které vyžaduje 2^n testovacích případů.

Nástrahy pokrytí kódu

Pokrytí kódu nese informaci o neotestovaném kódu, což může být pro testera užitečná, avšak často zavádějící informace. Z praxe je prokázané, že řada chyb není odhalena, třebaže kód, který ji obsahuje, je pokryt testy[39, 64]. Příklad takové chyby může být chybějící člen v podmínce `if`, použití špatné proměnné či operátoru ($<$ místo $\leqslant$) či neinicializovaná data[39].

Kapitola 3

Motivační příklad

V předchozích kapitolách byl popsán význam jednotkových testů a problematika jejich tvorby. V této kapitole bude tento proces představen na konkrétní metodě (viz program 3.1).

Předpokládejme, že se jedná o součást nějakého systému, který entity v databázi reprezentuje pomocí tříd dědících ze základní třídy `DataObject`. Ta obsahuje důležitou metodu `DataObject::Read`, která z databáze reprezentované rozhraním `IDatabase` získá všechny záznamy týkající se dané entity a nastaví své vnitřní hodnoty podle nich. Pokud při čtení dojde k chybě (například záznamy popisují neplatnou entitu nebo po nastavení hodnot nebude odpovídat kontrolní součet) tak metoda vrací `false`, v opačném případě `true`.

Třída `DataObject` a metoda `Read` je ale problematická. Například kontrola platnosti záznamů z hlediska hodnot `LastAccess` a `Created` obsahuje chybu. Pokud instance není inicializovaná (obě mají vlastnosti hodnotu 0), tak pomocí metody `Read` nelze tyto vlastnosti nastavit (část podmínky na řádku 19 nebude nikdy splněná, protože všechna kladná čísla jsou větší než 0).

Pro zpřehlednění by se nastavení vybraných 3 vlastností mohlo převést do metody `SetValue`, která by již nebyla abstraktní, ale virtuální. Před touto refaktorizací je ale nutné zformulovat soubor jednotkových testů, které prokáží zachování funkcionality, čímý se předchází regresi.

Manuální přístup

Program 3.2 ukazuje jeden z jednotkových testů. Ten ověří chování, pokud se z databáze načte prázdná kolekce záznamů a stav objektu se proto nezmění a zároveň kontrolní součet v databázi odpovídá kontrolnímu součtu určenému stavem objektu.

```
 1 public partial abstract class DataObject {
 2
 3   public bool Read(IDatabase database) {
 4     DataRecord[] records = database.GetRecords(Id);
 5
 6     for (int i = 0; i < records.Length; ++i) {
 7       DataRecord dr = records[i];
 8       if (dr.Name == "LastAccess") {
 9         if (LastAccess < dr.IntValue &&
10           dr.AsInt() >= Created) {
11           LastAccess = dr.IntValue;
12           continue;
13         }
14         else
15           return false;
16       }
17       if (dr.Name == "Created") {
18         if (Created < dr.IntValue &&
19           LastAccess >= dr.IntValue) {
20           Created = dr.IntValue;
21           continue;
22         }
23         else
24           return false;
25       }
26       if (dr.Name == "Author")  {
27        Author = dr.StringValue;
28         continue;
29       }
30       if (!SetValue(dr))
31         return false;
32     }
33
34     if (GetCheckSum() == database.GetCheckSum(Id)) {
35         return true;
36     }
37     else {
38         return false;
39     }
40   }
41
42   protected int GetCheckSum() {
43       return (Id * 1671941 + Created * 8329 +
44               LastAccess * 4909 + 54629)
45           % (1 << 30);
46   }
47
48   public abstract bool SetValue(DataRecord record);
49 }
```

Program 3.1: Metoda `DataObject::Read`.

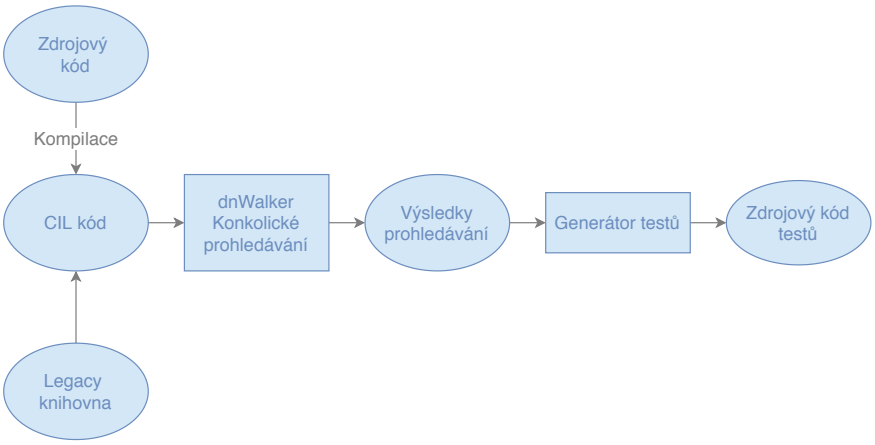
```
 1 public void ReadDataReturnValueSchema()
 2 {
 3   // Arrange input model heap
 4   DataRecord[] dataRecordArr1 = new DataRecord[0];
 5   Mock<DataObject> dataObject1_mock = new Mock<DataObject>();
 6   DataObject dataObject1 = dataObject1_mock.Object;
 7   dataObject1.Id = 55;
 8   dataObject1.Created = 113;
 9   dataObject1.LastAccess = 225;
10   dataObject1.Author = "John Doe";
11   Mock<IDatabase> iDatabase1_mock = new Mock<IDatabase>();
12   IDatabase iDatabase1 = iDatabase1_mock.Object;
13   iDatabase1_mock
14     .Setup(o => o.GetCheckSum(It.IsAny<int>()))
15     .Returns(3717698);
16   iDatabase1_mock
17     .Setup(o => o.GetRecords(It.IsAny<int>()))
18     .Returns(dataRecordArr1);
19
20   // Arrange method arguments
21   DataObject @this = dataObject1;
22   IDatabase database = iDatabase1;
23
24   bool result = @this.ReadData(database);
25   Assert.Equal(true, result);
26 }
```

Program 3.2: Jednotkový test pro metodu `DataObject::Read`, program 3.1, vygenerovaný pomocí nástroje *dnWalker*.

Další testy, pro tuto metodu by měly ověřit chování v závislosti na obsah samotných záznamů `DataRecord`. V poli navráceném databází se mohou nacházet prázdné hodnoty (`null`) nebo mohou záznamy mít špatnou kombinaci dat a názvu. K tomu je potřeba ošetřit problémy vzniklé z pořadí záznamů. Výsledek lokálních proměnných `LastAccess` a `Created` je závislé jak na stavu objektu přes zavoláním metody `Read`, tak na pořadí záznamů v databázi.

Tyto všechny případy vedou na formulaci mnoha testovacích dvojníků a jednotkových testů. I při použití izolačního frameworku se jedná o zdlouhavý proces, ve kterém je snadné udělat chybu, ať už kvůli formulaci špatných testovacích případů nebo kvůli přehlédnuté cestě exekuce skrz metodu.

Existují metody, pomocí kterých se zanalyzuje graf toku řízení (CFG) a naleznou se vhodné průchody, tak aby se dosáhlo pokrytí programu (viz 2.2.3) a zároveň se minimalizoval počet testů. Pro pokrytí všech větvení v programu (hran v CFG) je možné pomocí *cyklomatické komplexity* ($M = E - N + 2P$, kde E je počet hran, N počet uzlů a P počet komponent CFG) určit minimální



Obrázek 3.1: Postup generování testů pomocí nástroje *dnWalker*.

nutný počet nezávislých průchodů, v tomto případě $20 - 13 + 2 = 9$. K těmto nalezeným průchodům je ale nutné zformulovat vstupní argumenty, které je zajistí.

Takto nalezené testy ale neprověří výše zmíněné situace týkající se pořadí záznamů v databázi a vztahu lokálních proměnných `LastAccess` a `Created`. To vyžaduje silnější pokrytí metody (viz část 2.2.3).

Zautomatizovaný přístup

Úkolem nástroje *dnWalker*, předmětu této práce, je tento proces výrazně usnadnit. Vstup do *dnWalkeru* je již zkompilovaný program pro CLI a specifikace od uživatele (mimo jiné omezení nad vstupními argumenty testované metody nebo nastavení chování některých objektů). Výstupem je spustitelný testovací kód pro zadanou metodu. Vygenerované testovací případy zajistí požadované pokrytí testovací metody (schéma postupu je zobrazeno na obrázku 3.1).

Ve zbytku kapitoly ale uvažujme metodu v programu 3.3, která je jednodušší a lépe se na ní ukáže fungování výsledného nástroje. Její průběh závisí na 3 informacích: hodnotě proměnné `this.Value`, návratové hodnotě prvního zavolání metody `bar.GetValue` a rovnosti reference proměnné `bar` s `null` (podmínky tohoto typu bývají často přehlížené). Metoda pak může skončit třemi způsoby: chybou - vyhozením výjimky `NullReferenceException` a výsledky 4 nebo 5.

Vstup	Výstup
`bar` = `null`	`NullReferenceException`
`bar` ≠ `null` ∧ `bar.GetValue()` = 0 ∧ `this.Value` = 0	`return` 5
`bar` ≠ `null` ∧ `bar.GetValue()` = 0 ∧ `this.Value` = -3	`return` 4

Tabulka 3.1: Testovací případy pro `Baz::Foo`

Konkolický průzkum nebo exekuce metody (blíže popsaná v kapitole 4.4) odhalí všechny cesty metodou a nalezne vstupní a výstupní podmínky (*precondition* respektive *postcondition*), pro každou z nich. Testovací případy pro tuto metodu jsou sepsané v tabulce 3.1. Generátor testů z nalezených vstupních a výstupních podmínek vytvoří jednotkové testy, v tomto případě opět 3 (vypsané v programu 3.4).

Ve fázi generování testů je možné, že pro jedny vstupní podmínky vznikne více testů, protože každý vygenerovaný test ověřuje pouze jeden aspekt výstupního stavu (například návratová hodnota, změna stavu objektu či pole atp.).

```
 1 class Baz {
 2   public int Value;
 3
 4   public int Foo(IBar bar) {
 5     if (2 * bar.GetValue() - 3 == Value)
 6       return 4;
 7
 8     return 5;
 9   }
10
11   ...
12 }
```

Program 3.3: Ukázková metoda pro konkolickou exekuci, `Baz::Foo` s dynamickými a abstraktními členy.

Testy vygenerované tímto způsobem pochopitelně samy o sobě nestačí pro vhodné otestování softwaru. Jedná se ale o cennou informaci o testovaném kódu. Získá se *soubor testů* (*test suite*) , který pokryje celou metodu a tester tak získá přehled o tom, co a za jakých podmínek software přesně dělá, což je důležité pro refaktorizaci kódu (jak je zmíněno v úvodu kapitoly).

```
 1 class BazTests {
 2   public void FooExceptionSchema_1() {
 3     // Arrange input model heap
 4     SimpleMethod simpleMethod1 = new SimpleMethod();
 5     // Arrange method arguments
 6     SimpleMethod @this = simpleMethod1;
 7     IBar bar = null;
 8     Action foo = () => @this.Foo(bar);
 9     Assert.Throws<NullReferenceException>(foo);
10   }
11   public void FooReturnValueSchema_2() {
12     // Arrange input model heap
13     SimpleMethod simpleMethod1 = new SimpleMethod();
14     simpleMethod1.Value = 0;
15     Mock<IBar> iBar1_mock = new Mock<IBar>();
16     IBar iBar1 = iBar1_mock.Object;
17     iBar1_mock
18         .Setup(o => o.GetValue())
19         .Returns(0);
20     // Arrange method arguments
21     SimpleMethod @this = simpleMethod1;
22     IBar bar = iBar1;
23     int result = @this.Foo(bar);
24     Assert.Equal(5, result);
25   }
26   public void FooReturnValueSchema_3() {
27     // Arrange input model heap
28     Mock<IBar> iBar1_mock = new Mock<IBar>();
29     IBar iBar1 = iBar1_mock.Object;
30     iBar1_mock
31         .Setup(o => o.GetValue())
32         .Returns(0);
33     SimpleMethod simpleMethod1 = new SimpleMethod();
34     simpleMethod1.Value = -3;
35     // Arrange method arguments
36     SimpleMethod @this = simpleMethod1;
37     IBar bar = iBar1;
38     int result = @this.Foo(bar);
39     Assert.Equal(4, result);
40   }
41 }
```

Program 3.4: Jednotkové testy pro metodu `Baz::Foo`, program 3.1, vygenerované pomocí nástroje *dnWalker*.

Kapitola 4

Verifikace softwaru

V ideálním světě je vývoj softwaru úzce propojen s matematickým důkazem jeho korektnosti, respektive neschopnosti udělat chybu[25].

Základem pro verifikaci softwaru je schopnost vyjádřit, formalizovat jeho vlastnosti a rozhodnout jejich naplnění. S tím se pojí pojmy *formální specifikace* a *formální verifikace*. Formální specifikace je formalizovaný popis vlastností softwaru. Mezi používané jazyky patří *Z-notace* nebo SDL (*specification and description language*)[70]. Formální verifikace je pak ověření daného softwaru vůči jeho formální specifikaci. Spočívá v nalezení důkazu či kontradikce o naplnění daných vlastností[86]. Software je ale možné formálně verifikovat pouze vůči konkrétní formální specifikaci.

Pro reálné softwarové systémy je ale velice obtížné důkaz o korektnosti najít manuálně. Tato kapitola se věnuje třem technikám verifikace softwaru, spadající to skupiny *počítačově podporovaných* (*computer aided*) verifikačních technik: ověřování modelu, symbolická exekuce a konkolická exekuce. Před představením těchto technik je část textu věnována logice využívané těmito technikami.

4.1 Logika pro verifikaci softwaru

Existuje více přístupů pro verifikaci softwaru a každý z nich potřebuje vhodný nástroj pro rozhodnutí o (ne)naplnění vlastností softwaru. V této sekci jsou představeny čtyři rozšíření logiky používané pro uvažování nad softwarem či jeho modely.

4.1.1 Satisfiability module theories (SMT)

SMT (*satisfibility modulo theories*) je generalizace problému splnitelnosti logické formule (SAT). Logické výrazy jsou rozšířeny o výrazy interpretovanými v rámci „modula“ určité formální *teorie* predikátové logiky rozšířenou o obligátní teorii rovnosti (T_ε)[8].

Základními prvky SMT formule jsou funkce s určenou aritou. Funkce může mít interpretaci danou teorií, která ji definuje, například binární funkce $=$ z teorie rovnosti (T_ε) má danou interpretaci jako rovnost jejích dvou argumentů.

Neinterpretované funkce jsou maximálně flexibilní a řešení pak popisuje jejich interpretaci, aby byla konzistentní s omezeními danými instancí.

Funkce nulové arity jsou konstanty (například 1 v teorii celých čísel T_Z) nebo proměnné, pakliže se jedná o interpretované, respektive neinterpretované funkce.

SMT formule také může obsahovat standardní kvantifikátory $\exists$ a $\forall$. Nicméně jejich přítomnost v některých teoriích zvyšuje komplexitu rozhodovatelnosti. Například komplexita rozhodnutí formule v teorii reálných čísel (T_R) s kvantifikátory je obecně *doubly-exponential*, zatímco bez kvantifikátorů je její komplexita pouze polynomiální[29]. Často se proto pracuje s *quantifier free* (qf) fragmentem teorií.

Teorie - podpis, model

Podpis (Σ) je množina symbolů interpretovaných funkcí a predikátů. Σ-formule je formule jejíž mimologické symboly jsou obsažené v podpise Σ. Teorie T je množina sentencí. Pro daný podpis Σ je Σ-teorie (podpis teorie) množina sentencí, kde každá z nich je Σ-formule. Tímto způsobem se dodává interpretace symbolům patřícím do konkrétní teorie[8].

Řešení SMT problému je nalezení důkazu splnitelnost dané formule v rámci daného podpisu. Proto je použitý termín *modulo theories*, které specifikuje rozhodování na modulo použitých teorií[8].

S tím souvisí pojem model (M) a jeho doména $\mathrm{dom}(M)$, což je množina všech hodnot, které jsou v modelu M použity. Model se pro daný podpis Σ skládá ze tří částí[8]:

1. Mapování všech konstantních symbolů: $c \in \Sigma \rightarrow c^M \in \mathrm{dom}(M)$.
2. Mapování všech funkčních symbolů arity n: $f \in \Sigma \rightarrow f^M : \mathrm{dom}(M)^n \rightarrow \mathrm{dom}(M)$.
3. Mapování všech predikátových symbolů arity n: $p \in \Sigma \rightarrow p^M \subseteq \mathrm{dom}(M)^n$ (relace na množinu $\mathrm{dom}(M)$).

Pro daný model M lze definovat funkci s, která každé neinterpretované funkci $f \notin \Sigma$ přiřadí prvek $\mathrm{dom}(M)$. Tvrzení, že model M a funkce přiřazení proměnných s splňují formuli ϕ píšeme jako $\models_M \phi[s]$[8].

Pro danou Σ-teorii T může Σ-formule ϕ být[8]:

1. T-platná, právě tehdy když $\forall M, s \ \models_M \phi[s]$
2. T-splnitelná, právě tehdy když $\exists M, s \ \models_M \phi[s]$
3. T-nesplnitelná, právě tehdy když $\forall M, s \ \not\models_M \phi[s]$

Například formule

$$f(x) = 5 \wedge f(y) = 3 \wedge x \geqslant 5 \wedge y < x \tag{4.1}$$

obsahuje 3 neinterpretované funkce: f s aritou 1 a x a y s aritou nula. Další mimologické symboly = a < jsou interpretované funkce v rámci teorií rovnosti T_ε a celých čísel T_Z. Symbol 3 je zkrácením zápisu $1 + 1 + 1$, což jsou opět symboly z T_Z. Symbol $\wedge$ je logický operátor a proto není nutné jej interpretovat pomocí nějaké teorie.

Přiřazení neinterpretovaných symbolů, které dokazuje splnitelnost této formule může vypadat třeba takto:

$$\begin{aligned} x &= 10 \\ y &= 9 \\ f(t) &= \begin{cases} 5 & t = 10 \\ 3 & t = 9 \\ 42 & \text{jinak} \end{cases} \end{aligned} \tag{4.2}$$

Vícedruhová predikátová logika

Komplexnost SMT formulí při zapojení více teorií vede k nutnosti zavedení typů hodnot. K tomu existuje tzv. vícedruhová predikátová logika. Oproti predikátové logice je navíc definovaná množina *druhů* (*sort*) a zobrazení, které ke každé funkci, predikátu, proměnné a konstantě přiřadí druh[34]. Typickými druhy jsou celá čísla, reálná čísla, logické hodnoty atp.

Druhy také mohou reprezentovat složitější struktury, například řetězce, pole nebo množiny, a zároveň být parametrické. Například druh množina obsahuje parametr, který udává druh prvků množiny[34].

SMT Solver

SMT (*satisfiability modulo theories*) solverje nástroj pro řešení instancí SMT problému. Existuje velké množství solverů, například *Z3*[30] nebo *CVC5*[7].

Důležité pro vývoj SMT solverů je standardizace SMT teorií a jazyka solverů. Za tím účelem vznikla iniciativa SMT-LIB[115], která pořádá soutěže solverů[114].

4.1.2 Temporální logika

Temporální logika je zformulovaná za účelem popisu světa, který se vyvíjí v čase. Diskrétní vývoj v čase je modelován posloupností stavů. Vzhledem k tomu, že z jednoho stavu bývá možné přejít do více následujících stavů, možný vývoj z konkrétního stavu obecně popisuje strom[25].

Formálně je temporální logika rozšíření modální logiky o kvantifikátory cest interpretovaných nad stavy a temporální operátory interpretované nad cestami (posloupnostmi stavů)[25]. Tyto kvantifikátory a operátory jsou vypsány v tabulce 4.1.

Nezávisle na sobě byly zformulovány dvě temporální logiky – CTL (*computation tree logic*) a LTL (*linear temporal logic*). K nim byla zformulována logika CTL*, již jsou obě fragmentem. CTL vyžaduje, aby kvantifikátor cesty

Značení	Význam
A	Pro každou nekonečnou cestu z tohoto stavu ...
E	Existuje alespoň jedna nekonečná cesta z tohoto stavu ...
Xp	... tvrzení p platí v následujícím stavu.
Fp	... tvrzení p platí v nějakém stavu v budoucnosti.
Gp	... tvrzení p platí ve všech budoucích stavech.
q**U**p	... tvrzení p platí v nějakém stavu v budoucnosti a tvrzení q platí ve všech stavech, dokud neplatí p

Tabulka 4.1: Kvantifikátory cest a temporální operátory CTL* (*computation tree logic**) temporální logiky z [25]

Výraz	e	::=	$n \mid x \mid -e \mid e_1 + e_2$
Logická podmínka	b	::=	$\mathbf{true} \mid e_1 = e_2 \mid e_1 \geqslant e_2 \mid \neg b \mid b_1 \wedge b_2$
Program	p	::=	$x := e \mid p_1; p_2 \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \mid \texttt{while}\ b\ p$
Tvrzení	s	::=	$\mathbf{true} \mid b \mid \neg s \mid s_1 \wedge s_2 \mid \forall v\ s$

Tabulka 4.2: Syntax imperativního a ověřovacího jazyka.

byl vždy následován temporálním operátorem. Lineární temporální logika zakazuje všechny kvantifikátory, kromě jednoho **A** na začátku formule[25].

Formule 4.3 ukazuje jednoduchý příklad použití LTL pro popis situace na letišti. Pro každou cestou (**A**) platí, že pro každý moment v budoucnosti (**G**) platí že, dojde-li ke ztrátě pasu nebo letenky, tak v následujícím momentě (**X**) nebude možný odlet.

$$\mathbf{A}\ \mathbf{G}((\neg pas \vee \neg letenka) \implies \mathbf{X} \neg odlet) \tag{4.3}$$

4.1.3 Floyd-Hoareova logika

Zatímco SMT a temporální logika poskytuje nástroje, jak uvažovat nad obecnými systémy a modely, Floyd-Hoareova logika (formalizovaná v [48] mimo jiné na základě závěrů z [35]) nabízí systém pro obecnou formální verifikaci imperativních sekvenčních programů, odpovídající syntaxi v tabulce 4.2.

Provedení programu pracuje se stavem s, který odpovídá ohodnocení proměnných. Program splňuje tvrzení a, pakliže v daném okamžiku platí $s \models a$. Pro konstrukty programu jsou definována odvozovací pravidla (zapisovaná ve

tvaru $\frac{p_1 \ \cdots \ p_n}{c}$, kde p_i jsou předpoklady a c je závěr, pro axiomy je množina předpokladů prázdná), která popisují stav programu před a po provedení.

Základem těchto pravidel je tzv. Hoareova trojice:

$$\{P\}C\{Q\} \tag{4.4}$$

kde P je vstupní podmínka (*precondition*), C je program, a Q je výsledek či výstupní podmínka (*postcondition*). Konkrétní příklad Hoareovy trojice může být následující:

$$\{y > 2\}x := y + 1\{x > 3\} \tag{4.5}$$

Tato notace se čte: „za předpokladu, že je ve stavu s splněné tvrzení P ($s \models P$) předtím než je spuštěn program C ($\langle C \rangle s \rightsquigarrow s'$, program C změní stav s na s'), pak bude splněné tvrzení Q ($s' \models Q$) po jeho ukončení, nebo program C neskončí“[48].

1. Axiom prázdného programu:

$$\frac{}{\{P\} \ \texttt{skip} \ \{P\}} \tag{4.6}$$

2. Axiom přiřazení:

$$\frac{}{\{P[E/x]\} \ \texttt{x := E} \ \{P\}} \tag{4.7}$$

   kde výraz $P[E/x]$ popisuje podmínku, kde každý volný výskyt výrazu x je nahrazen výrazem E.

3. Pravidlo neměnnosti:

$$\frac{\{P\}C\{Q\}}{\{P \wedge R\}C\{P \wedge R\}} \tag{4.8}$$

   kde program C neupravuje volné proměnné formule R.

4. Pravidlo konjunkce:

$$\frac{\{P_1\}C\{Q_1\} \ , \{P_2\}C\{Q_2\}}{\{P_1 \wedge P_2\}C\{Q_1 \wedge Q_2\}} \tag{4.9}$$

5. Pravidlo disjunkce:

$$\frac{\{P_1\}C\{Q_1\} \ , \{P_2\}C\{Q_2\}}{\{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}} \tag{4.10}$$

6. Pravidlo následků:

$$\frac{P' \implies P \ , \{P\}C\{Q\} \ , Q \implies Q'}{\{P'\}C\{Q'\}} \tag{4.11}$$

7. Pravidlo kompozice:

$$\frac{\{P\}C_1\{Q_1\}\ ,\{Q_1\}C_2\{Q_2\}}{\{P\}C_1C_2\{Q_2\}} \tag{4.12}$$

8. Pravidlo iterace

$$\frac{\{P \wedge B\}C\{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ C\ \texttt{done}\ \{\neg B \wedge P\}} \tag{4.13}$$

9. Pravidlo podmínky

$$\frac{\{P \wedge B\}S\{P\}\ ,\{P \wedge \neg B\}T\{P\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T\ \{P\}} \tag{4.14}$$

Tyto axiomy a pravidla mají nedostatek v práci s vedlejšími efekty a chybí operace nad dynamickými daty – alokace, dealokace, čtení a zápis. Tomu se věnuje následující část textu.

4.1.4 Separační logika

V předchozí části byly popsány principy Floyd Hoareovy logiky. Následujícím text se věnuje jejímu rozšíření nazvaném *separační logika*, které přidává pravidla pracující se symbolickou haldou[76, 77].

Třebaže separační logika vznikla z uvažování nad sdílenými a proměnlivými datovými strukturami, je možné její abstrakci aplikovat i na jiné procesy, které sdílí zdroje[77]. Pro rozhodnutí splnitelnosti separační logiky existuje řada rozhodovacích procedur pro výrazy separační logiky[19, 20, 57, 58, 52, 75, 68, 89], nicméně separační logika není formalizovaná a nebývá standardní součástí standardních SMT solverů. Formalizaci usiluje iniciativa SL-COMP[113]. Z rozšířených solverů ji obsahuje solver *CVC4* či jeho novější verze *CVC5*[1], ale v této práci použitý solver *Z3* ji neobsahuje[2].

[1] `https://cvc5.github.io/docs/cvc5-1.0.2/theories/separation-logic.html`
[2] `https://github.com/Z3Prover/z3/issues/811`

Formulace separační logiky

Separační logika pracuje se dvěma objekty: $Store_V$ (nebo také $Stack_V$) a $Heap$[3]. $Store_V$ poskytuje mapování proměnných ($v \in V$ a V je konečná množina proměnných) na hodnoty ($u \in Val$) a lze jej přirovnat k registrům CPU. $Store_V$ tak existuje v kontextu konečné množiny proměnných V. Množina všech $Store_V$ je pak $Stores_V$. $Heap$ poskytuje mapování adres, nebo lokací, ($l \in Loc$) na struktury obsahující informace o neskalárních datech a aproximuje tak operační paměť. Stav programu je pak reprezentován dvojicí ($Store_V\ s, Heap\ h$). Množina všech stavů je značena jako $States_V \stackrel{\text{def}}{=} Stores_V \times Heaps$[77].

Existuje několik formulací separační logiky, které se liší v sémantice objektu $Heap$, respektive v popise struktur, které obsahují informace o neskalárních datech. Níže jsou představeny 3 z nich, a to: první verze separační logiky, formulovaná v [76], nejčastěji využívaná verze z [77] a formulace využita v této práci založená na [57]. Obrázek 4.1 ukazuje jejich rozdíly.

V následujícím textu je použit zápis #, který značí nezávislost či disjunknost dvou entit. Pro množiny platí $A \# B \iff A \cap B = \varnothing$. Pro zobrazení platí $A \# B \iff \mathrm{dom}(A) \cap \mathrm{dom}(B) = \varnothing$. Dále termín $Heap$ se používá pro popis vlastností komponenty separační logiky, termín *halda* pak v případě popisu vlastností části paměti vyhrazené pro dynamicky alokovaná data.

Obecně je sémantika definována pomocí množin:

Val	:	množina všech hodnot, kterých mohou nabýt proměnné
Loc	:	$\mathrm{dom}(\bigcup_{h \in Heaps})$, množina všech lokací na haldě
		zpravidla $Loc \subseteq Val$
$Atoms$	:	množina všech nečíselných skalárních hodnot
		zpravidla $Atoms \subseteq Val$ a $\mathbf{nil} \in Atoms$
N	:	množina všech číselných hodnot
		zpravidla $N \subseteq Val$

Halda jako množina vektorů. V této sémantice jsou množiny $Atoms$, Loc a N disjunktní. Struktury na haldě jsou reprezentovány pomocí konečné k-tice hodnot, kde každá z nich může nabývat libovolné skalární hodnoty $u \in Val$, kde $\mathbf{nil} \in Val$ nebo lokace $l \in Loc$ a tím „ukazovat“ na další část haldy[76, 77].

[3]Pro termíny spojené se separační logikou (názvy objektů, množina atp) jsou použity originální anglické výrazy z [77] a [57].

Pro $Store$, $Heap$ a s nimi spojené množiny platí následující definice:

$$\begin{aligned} Val &\stackrel{\text{def}}{=} N \cup Atoms \cup Loc \\ &\quad \text{kde } Atoms \# Loc,\ Loc \# N,\ Atoms \# N \text{ a } \mathbf{nil} \in Atoms \\ Stores_V &\stackrel{\text{def}}{=} V \to Val \\ &\quad \text{kde } V \text{ je konečná množina proměnných} \\ Heaps &\stackrel{\text{def}}{=} \bigcup_{L \subseteq^{\text{fin}} Loc} \left(L \to Val^k\right) \\ &\quad \text{kde } k \text{ je konečné přirozené číslo} \end{aligned} \tag{4.15}$$

Halda jako nekonečná posloupnost hodnot. Pomocí předchozí formulace nelze vyjádřit libovolnou aritmetiku ukazatelů (*pointer arithmetics*) (pouze rovnost či nerovnost). Proto vznikla formulace, ve které jsou lokace $l \in Loc$ podmnožinou přirozených čísel.

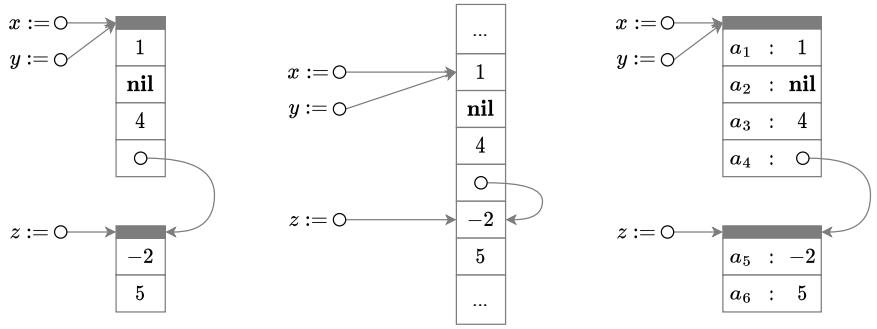
Všechny hodnoty jsou tedy čísla, z nichž nekonečně mnoho jsou adresy na haldě. $Heap$ potom připomíná paměť, ve které každá buňka může mít svou hodnotu a zároveň informaci, zda je aktivní. $\mathrm{dom}(h)$ je potom množina všech adres, které ukazují na aktivní buňky.

Také se předpokládá, že pro každé $n \geqslant 0$ obsahuje množina adres nekonečně mnoho na sebe navazujících sekvencí o délce n. Tím se zajistí možnost alokace neomezeně velkých struktur, které se nepřekrývají[77].

$$\begin{aligned} Val &\stackrel{\text{def}}{=} N \\ &\quad \text{kde } Loc \cup Atoms \subseteq N \text{ a } \mathbf{nil} \in Atoms \\ Stores_V &\stackrel{\text{def}}{=} V \to Val \\ &\quad \text{kde } V \text{ je konečná množina proměnných} \\ Heaps &\stackrel{\text{def}}{=} \bigcup_{L \subseteq^{\text{fin}} Loc} (L \to Val) \end{aligned} \tag{4.16}$$

Halda jako množina datových buněk. Předchozí formulace je výhodná pro popis nízkoúrovňových jazyků, jako například C/C++, které pracují s aritmetikou ukazatelů [77]. Pro účely této práce ale není příliš vhodná, protože cílí na vysokoúrovňové objektově orientované programovací jazyky, které tyto možnosti nevyužijí, ale naopak potřebují flexibilně popsat vlastnosti objektů (například C#). Proto je využita formulace založená na [57].

Element na haldě je reprezentován datovou buňkou ($r \in R$), která je



(a) : Halda jako množina n-tic.

(b) : Halda jako nekonečná posloupnost.

(c) : Halda jako množina datových buněk.

Obrázek 4.1: Příklady formulací separační logiky. V každém případě se zobrazuje situace, kde $x = y$ a x ukazuje na strukturu s hodnotami 1, **nil**, 4 a ukazatelem na strukturu s hodnotami -2 a 5, na kterou zároveň ukazuje proměnná z.

sjednocení zobrazení z konečné množiny atributů ($a \in A$) na množinu Val. Atribut představuje jednu vlastnost buňky. Pro účely této práce uvažujeme šest typů atributů, blíže popsané v části 6.2.5.

$$\begin{aligned} Val &\stackrel{\text{def}}{=} N \cup Atoms \cup Loc \\ &\quad \text{kde } Atoms \# Loc,\ Loc \# N,\ Atoms \# N \text{ a } \mathbf{nil} \in Atoms \\ Stores_V &\stackrel{\text{def}}{=} V \to Val \\ &\quad \text{kde } V \text{ je konečná množina proměnných} \\ Heaps &\stackrel{\text{def}}{=} \bigcup_{L \subseteq^{\text{fin}} Loc} (L \to (R \to A \to Val)) \end{aligned} \tag{4.17}$$

Objekt $Store_V$ je vždy definovaný nad konečnou množinou proměnných V. Ve zbytku textu toto označení bude zanedbané a předpokládá se, že všechny proměnné, které figurují v dané situaci jsou součástí konečné množiny proměnných V nad kterou je daný $Store$ definovaný.

Principy separační logiky

Zásadní výhodou separační logiky je možnost lokálního uvažování na malé části haldy, které je nezávislé na zbytku. To je umožněno zejména pomocí operátorů *separační konjunkce*, značeného jako $*$, který se čte „a separátně“ (*and separately*) a *separační implikace*, značeného jako $-\!\!*$ (*magic wand*) [77].

Separační konjunkce. $s, h \models \phi_1 * \phi_2$ platí právě tehdy když je možné $Heap$ h rozdělit na dvě nezávislé, disjunktní, části h_1 a h_2 ($h_1 \# h_2 \wedge \mathrm{dom}(h_1) \cup \mathrm{dom}(h_2) = \mathrm{dom}(h)$) takové, že $s, h_1 \models \phi_1$ a zároveň $s, h_2 \models \phi_2$.

Separační implikace. $s, h \models \phi_1 \mathrel{-\!\!*} \phi_2$ platí právě tehdy když $\exists h'$ $(s, h' \models \phi_1) \implies (s, h \cup h' \models \phi_2)$, kde $h' \# h$. Což znamená, že pokud $Heap$ h rozšíříme o disjunktní $Heap$ h', na které je splněna formule ϕ_1, pak na sjednocení h a h' je splněná formule ϕ_2.

K těmto operátorům jsou definovány 2 formy tvrzení: *ukazuje na* (*points to*), značené jako $\mapsto$ a prázdná halda, značené jako **emp** .

Ukazuje na. $s, h \models v \mapsto (\bar{a} : \bar{e})$ platí právě tehdy když $h = \{s(v) \rightarrow r\} \wedge r(a_i) = s(e_i)$, tedy hodnota proměnné v je lokace datové buňky r, jejíž atributy a_i mají hodnotu e_i (zápis $s(e)$ značí hodnotu výrazu e v rámci $Store_V$ s).

Prázdná halda. $s, h \models \mathbf{emp}$ platí právě tehdy když $\mathrm{dom}(h) = \varnothing$, tedy $Heap$ neobsahuje žádnou datovou buňku.

Pro ilustraci výše popsaných operátorů a tvrzení uvažujme situaci ukázanou na obrázku 4.1c, kterou lze popsat formulí

$$x \mapsto (a_1 : 1, a_2 : \mathbf{nil}, a_3 : 4, a_4 : z) \wedge x = y * z \mapsto (a_5 : -2, a_6 : 5) \tag{4.18}$$

Ta využívá vlastnosti separační konjunkce a vylučuje možnost rovnosti proměnných z a x. Podobně lze stejnou situaci popsat formulí

$$x \mapsto (a_1 : 1, a_2 : \mathbf{nil}) \wedge y \mapsto (a_3 : 4, a_4 : z) \wedge x = y \\ * z \mapsto (a_5 : -2, a_6 : 5) \tag{4.19}$$

ve které je stejná struktura na haldě popsána ze dvou různých přístupů a to přes proměnnou x a y, které jsou si rovné.

Avšak formule

$$x \mapsto (a_1 : 1) \wedge y \mapsto (a_1 : 4) \wedge x = y \tag{4.20}$$

ve které v jedné buňce, na kterou ukazují x a y jsou očekávané různé hodnoty (1 a 4) pro atribut a_1 a proto musí být nesplnitelná.

Výraz	e	::=	$\ldots \mid \mathbf{nil}$
Program	p	::=	$\ldots \mid x := \texttt{cons}(a_1 : e_1, \ldots) \mid x := e.a \mid x.a := e \mid \texttt{dispose}\ e$
Tvrzení	s	::=	$\ldots \mid \mathbf{emp} \mid e \mapsto (a_1 : e_1, \ldots) \mid s_1 * s_2 \mid s_1 \mathrel{-\!\!*} s_2$

Tabulka 4.3: Syntax rozšíření imperativního a ověřovacího jazyka z tabulky 4.2 pomocí separační logiky.

Pro příklad separační implikace předpokládejme, že existuje formule ϕ, která je splněná pro *Store* s a *Heap* h $(s, h \models \phi)$ takové, že $s \supseteq \{x : \alpha\}$ a $h \supseteq \{\alpha \mapsto (a : 16)\}$. Potom platí

$$s, h' \models (x \mapsto (a : 16)) \mathrel{-\!\!*} \phi \tag{4.21}$$

kde $\mathrm{dom}(h') = \mathrm{dom}(h) \backslash \alpha$, tedy pro stejný *Store* s, ale *Heap* h', která vznikne z původní h odebráním buňky na lokaci α[77].

Ve zbytku textu je hodnota proměnné v v rámci *Store* s značena jako $s(v)$. Výraz $v.a$ značí hodnotu atributu a pro buňku na kterou ukazuje proměnná v. Vyraz $v.a := e$ pak značí zápis hodnoty e do atributu a buňky, na kterou ukazuje v (viz axiom mutace 4.24).

Pravidla vycházející ze separační logiky

Syntax imperativního a ověřovacího jazyka popsanou v tabulce 4.2 můžeme rozšířit o nové instrukce a tvrzení podle tabulky 4.3. Nové formy tvrzení vychází z operátorů tvrzení popsaných v předchozí části. Nové instrukce (alokace, čtení, zápis/mutace a dealokace) jsou blíže popsané v následujícím textu.

Základem pravidel pro instrukce separační logiky je tzv. *rámcové pravidlo* (*frame rule*), které vychází ze separační konjunkce[77]:

$$\frac{\{P * R\}\ C\ \{Q * R\}}{\{P\}C\{Q\}} \tag{4.22}$$

kde žádná z volných proměnných vyskytujících se ve tvrzení R nejsou modifikované programem C. Rámcové pravidlo je vyjádřením lokálního uvažování separační logiky a vychází z pravidla neměnnosti tradiční Hoareovy logiky (viz rovnice 4.8), které při přestupu k separační logice pozbývá platnosti.

Například následující úvaha je platná z hlediska Hoareovy logiky:

$$\frac{\{x \mapsto \_\}x.a := 4\{x \mapsto (a : 4)\}}{\{x \mapsto \_ \wedge y \mapsto (a : 3)\}\ x.a := 4\ \{x \mapsto (a : 4) \wedge y \mapsto (a : 3)\}} \tag{4.23}$$

avšak v separační logice neplatí, protože vstupní podmínka nevylučuje $x = y$ (aliasing ukazatelů).

1. Axiom mutace, zápisu (lokální a globální varianta):

$$\frac{}{\{v \mapsto (b : \_)\}\ v.b = e\ \{v \mapsto (b : e)\}} \tag{4.24}$$

$$\frac{}{\{v \mapsto (b : \_) * R\}\ v.b = e\ \{v \mapsto (b : e) * R\}} \tag{4.25}$$

   kde v není volná proměnná ve výrazu R.

2. Axiom dealokace (lokální a globální varianta):

$$\frac{}{\{v \mapsto (\_)\}\ \texttt{dispose}\ v\ \{\mathbf{emp}\}} \tag{4.26}$$

$$\frac{}{\{v \mapsto (\_) * R\}\ \texttt{dispose}\ v\ \{R\}} \tag{4.27}$$

3. Axiom alokace (lokální a globální varianta):

$$\frac{}{\{v = \_ \wedge \mathbf{emp}\}\ v := \texttt{cons}(t, \bar{a} : \bar{e})\ \{v \mapsto (\bar{a} : \bar{e})\}} \tag{4.28}$$

   kde v není volná proměnná ve výrazech $\bar{e}$.

$$\frac{}{\{R\}\ v := \texttt{cons}(t, \bar{a} : \bar{e})\ \{\exists v\ v \mapsto (\bar{a} : \bar{e}) * R\}} \tag{4.29}$$

   kde v není volná proměnná ve výrazech $\bar{e}$ ani R.

4. Axiom načtení hodnoty (lokální a globální varianta):

$$\frac{}{\{v =\!\!\mapsto (a : e) \wedge v' = e'\}\ v' := v.a\ \{v =\!\!\mapsto (a : e) \wedge v' = e\}} \tag{4.30}$$

   kde v a v' jsou rozdílné.

$$\frac{}{\{v =\!\!\mapsto (a : e) \wedge v' = e' * R\}\ v' := v.a\ \{v =\!\!\mapsto (a : e) \wedge v' = e * R\}} \tag{4.31}$$

   kde v a v' jsou rozdílné a nejsou volné proměnné v R.

4.2 Ověřování modelu

Ověřování modelu (*model checking*) nachází důkaz či kontradikci korektnosti pomocí ověření platnosti formulí nad abstrakcí systémů – jeho dynamickým modelem. Existuje celá řada modelů, které poskytují abstrakci konkrétních typů systémů[25]. *Model checker*[4] je potom nástroj, který ověřování modelu provádí.

[4]Pro absenci rozumného českého použit originální termín *checker*

4.2.1 Model a logika nad jeho vlastnostmi

Dynamickým modelem je míněna trojice $\langle S, T, L \rangle$, kde S je množina všech stavů, $R \subseteq S \times S$ popisuje přechody mezi stavy a $L : S \to \{\mathbf{true}, \mathbf{false}\}^A$, kde A je množina všech atomických logických tvrzení o systému a $L(s)$ popisuje která tvrzení ve stavu s platí a která neplatí. Trojice $\langle S, T, L \rangle$ se nazývá *Kripkeho struktura*, ke které můžeme definovat orientovaný graf (S, T) který popisuje možné přechody mezi stavy modelu. Dynamické chování Kripkeho struktury je reprezentováno cestou π grafem (S, T), která je tvořena potenciálně nekonečnou posloupností stavů $s_0, s_1, ...$, takových, že $(s_i, s_{i+1}) \in T$[25].

Pro vyjádření specifikace systému se používají formule temporální logiky nad cestami v grafu modelu, viz část 4.1.2.

4.2.2 Model explicitního stavu

Pro verifikaci „standardního“ softwaru (programy napsané v imperativním sekvenčním jazyce, které je možné spouštěn ve více procesech, které se navzájem ovlivňují) se efektivně využívají modely *explicitního stavu* (*explicit-state*). Explicitní stav je vhodný pro verifikaci softwaru kvůli jeho schopnostem vyjádřit asynchronní a paralelní procesy a jeho kompatibilitě s *redukcí parciálního pořadí* (*partial-order reduction*, POR)[49].

Ověřování modelu explicitního stavu vyžaduje naplnění 2 předpokladů[49]:

1. Systém musí být reprezentovatelný konečným stavem, tzn. lze jej vyjádřit jako konečnou n-tici hodnot z konečné domény, a systém může měnit svůj stav pouze pomocí přechodů mezi stavy. To znamená, že model systému může být konečnou množinou *konečných stavových automatů* (FSA).
2. Exekuce systému musí být modelovatelná pomocí posloupnosti oddělených přechodů mezi stavy. Tento předpoklad umožní paralelní exekuci systému modelovat jako jeden stavový automat s libovolným prokládáním akcí jednotlivých procesů.

Nechť je konečný stavový automat $A = \{S, s_0, L, T, F\}$, kde S je konečná množina stavů, s_0 je počáteční stav, L je abeceda, $T \subseteq S \times L \times S$ je množina

přechodů mezi stavy a $F \subseteq S$ je množina konečných stavů. Ověření modelu explicitního stavu je založené na vybudování FSA $A' = \{S', s'_0, L', T', F'\}$, což je prokládaný součin automatů všech procesů systému $A_0, ..., A_N$. Jeho vybudování je definované jako[49]:

$$\begin{aligned} S' &= A_0.S \times A_1.S \times ... \times A_N.S \\ s'_0 &= (A_0.s_0, ..., A_N.s_0) \\ L' &= A_0.L \cup A_1.L \cup ... \cup A_N.L \\ T' \subseteq\ & S' \times L' \times S'\ \forall((A_0.s_0, ..., A_N.s_N), e, (A_0.t_0, ..., A_N.t_N)) \in T' \\ & \exists i\ (A_i.s_i, e, A_i.t_i) \in A_i.T \wedge e \in A_i.L \wedge \forall j \neq i\ A_j.s_j = A_j.t_j \\ F' &= A_0.F \times A_1.F \times ... \times A_N.F \end{aligned} \tag{4.32}$$

Z tvorby je složitější pouze množina přechodů. Každý přechod v prokládaném součinu automatů odpovídá právě jednomu přechodu v rámci jednoho z původních automatů. Znamená to také, že jeden přechod zaznamená změnu pouze v rámci jednoho stavu. Těmito vlastnostmi je zajištěná schopnost modelovat vícevláknové exekuce na jednom či více výpočetních jednotkách a ověřit libovolné pořadí jednotlivých instrukcí[49].

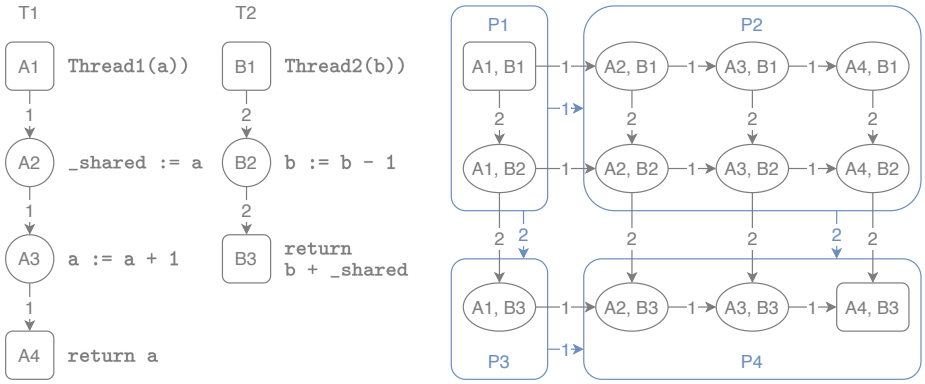
Samotné ověření modelu znamená prohledání stavového prostoru a na každém stavu prokázat vlastnosti softwaru. Obecně se vlastnosti dělí do dvou skupin: vlastnosti *bezpečnosti* (*safety*) a *živosti*(*liveness*). První skupina popisuje vlastnosti systému označením zakázaného chování – nesplnění ověření (*assert*) či zablokování (*deadlock*) nebo souběh (*race condition*), zatímco druhá definuje požadované chování. Obecně jsou podmínky formulované pomocí LTL (viz 4.1.2)[49].

Redukce parciálního pořadí (*partial order reduction*, POR)

Model explicitního stavu je ale náchylný ke kombinatorické explozi, jak je vidět z tvorby prokládaného FSA. Silným nástrojem pro zefektivnění prohledávání prostoru modelu explicitního stavu reprezentující asynchronní systémy je *redukce parciálního pořadí* (POR)[72].

V prokládaném FSA lze přejít mezi dvěma stavy pomocí více cest (což reprezentuje část exekuce), které se ale liší pouze pořadím nezávislých přechodů (pořadí instrukcí). Redukce parciálního pořadí (POR) pak tyto posloupnosti redukuje na jeden stav, čímž vytváří nový stavový prostor, ve kterém se nachází *třídy ekvivalence exekucí*[72].

Uvažujme program 4.1. Vlákno t_1 respektive t_2, spouštějící metodu `Thread1`



Obrázek 4.2: Prokládaný součin a POR pro program 4.1. Abeceda $\Sigma_{T1} = \{1\}$ a $\Sigma_{T2} = \{2\}$. Pro FSA prokládaného součinu je pak abeceda $\Sigma = \{1, 2\}$, kde 1 a 2 odpovídá spuštění následující instrukce prvního, respektive druhého procesu.

respektive `Thread1`, lze modelovat pomocí FSA o 4 stavech a 3 přechodech, respektive 3 stavech a 2 přechodech. Na obrázku 4.2 je ukázaný prokládaný součin těchto FSA, který obsahuje $4 \times 3 = 12$ stavů a $(4-1) \times 3 + (3-1) \times 4 = 17$ přechodů. Možných cest (exekucí) je pak[5] $\frac{((4-1)+(3-1))!}{(4-1)! \times (3-1)!} = 10$.

Podle obrázku 4.2 lze tento velký FSA redukovat na FSA se 4 stavy, 4 přechody a 2 průchody. Protože z pořadí instrukcí je důležité pouze zda instrukce, která upravuje sdílenou proměnou `_shared` (řádek 5 a stav $(A2)$) proběhne před nebo po instrukci, která z ní čte (řádek 12 a stav $(B3)$). V redukovaném FSA stav $P1$ reprezentuje část exekuce před zápisem do sdílené proměnné, stav $P2$ po zápise a před čtením, $P3$ po čtení a před zápisem a $P4$ po čtení a po zápise.

4.3 Symbolická exekuce

Dalším nástrojem pro verifikaci softwaru je symbolická exekuce. Jejím základním principem je nahrazení všech možných konkrétních vstupů symboly. Program pak manipuluje tyto symboly místo konkrétních hodnot. Symbolická exekuce je studována již od 70. let minulého století [55]. Její rozvoj jde ruku v ruce s vývojem SMT solverů a technik, které využívají (viz část 4.1.1).

[5] Pro tento případ odpovídá FSA mřížce s šířkou 4 a výškou 3, ve které se lze pohybovat doprava a dolů. Počet všech možných průchodů touto mřížkou z buňky $(1, 1)$ do (w, h) je $\frac{((w-1)+(h-1))!}{(w-1)!(h-1)!}$, tedy vybíráme všechny možné permutace pro $(w-1)+(h-1)$ kroků, ale rozlišujeme pouze krok doprava proti dolů a ne mezi jednotlivými kroky doprava respektive dolů, proto vydělíme všemi permutacemi kroků doprava $(w-1)!$ v kombinaci se všemi permutacemi kroků dolů $(h-1)!$.

```
 1 class DataRace {
 2     int _shared;
 3
 4     int Thread1(int a) {
 5         _shared = a;
 6         a++;
 7         return a;
 8     }
 9
10     int Thread2(int b) {
11         b--;
12         return b + _shared;
13     }
14 }
```

Program 4.1: Příklad pro redukci parciálního pořadí.

Dnes je základem desítek nástrojů pro analýzu kódu. Neúplný přehled těchto nástrojů je zpracovaný v [95].

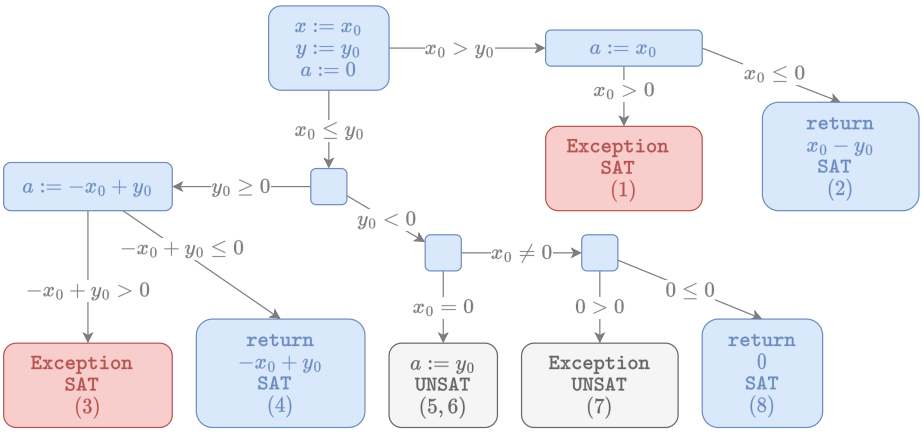
Tato část se věnuje symbolické exekuci. Nejprve je představen postup symbolické exekuce, způsoby její implementace, a následně jsou popsány její nedostatky: kombinatorická exploze symbolických stavů, závislost na SMT solverech a jejich (ne)schopnosti určit splnitelnost výrazů a interakce s prostředím.

4.3.1 Základní postup symbolické exekuce

Symbolická exekuce pracuje podobně jako normální (konkrétní) exekuce. Zatímco konkrétní exekuce ukládá vypočítané hodnoty do proměnných a tak vytváří konkrétní stav programu, symbolická exekuce místo vypočítaných hodnot ukládá symbolický výraz a vytváří tak *symbolický stav* programu[93].

Při návštěvě rozhodovacího bodu, se exekuce rozdělí (*fork*) na dvě větve, kde v první je podmínka v rozhodovacím bodě splněná a v druhé není. Symbolický stav se pak rozšíří o tzv. *podmínku cesty* (anglicky *path condition*), omezení nad vstupními symboly, které musí být naplněno, aby exekuce navštívila právě tuto lokaci v programu. Tímto způsobem se identifikují všechny možné (dosažitelné) průchody zkoumaným programem[93].

Výstupem symbolické exekuce je *strom symbolické exekuce*. Každý vrchol představuje symbolický stav, listy představují koncové stavy (s chybou nebo



Obrázek 4.3: Strom symbolické exekuce programu 4.2. Modré uzly - dostupné stavy, červené uzly - chybové stavy, šedé uzly - nedostupné stavy. Pro přehlednost se vypisuje pouze změna symbolického stavu. Omezení jsou popsaná v závislosti na vstupních hodnotách x_0 a y_0.

s návratovou hodnotou) nebo nedosažitelné stavy (situace, kdy je podmínka cesty nesplnitelná). Hrany symbolizují přechody mezi symbolickými stavy [93].

Uvažujme program 4.2. Podle analýzy struktury metody existuje celkem 8 průchodů tímto programem (4 průchody zřetězeného `if - then - else` (řádky 3 až 5) a nezávisle na tom 2 průchody okolo `if` na řádku 7, tedy $4 \times 2 = 8$). Nicméně, jak ukazuje obrázek 4.3, splnitelných průchodů je pouze 5 a zbylé 3 jsou nesplnitelné.

```
1 int Foo(int x, int y) {
2   int a = 0;
3   if (x > y) a = x;
4   else if (y >= 0) a = -x + y;
5   else if (x == 0) a = y;
6
7   if (a > 0) throw new Exception();
8
9   return a;
10 }
```

Program 4.2: Jednoduché argumenty s lineárními omezeními

Pro průchody označené (5) a (6) je to způsobené podmínkou cesty nutné pro naplnění podmínky na řádku 5. Samotné $x = 0$ je snadno splnitelné, avšak při konjunkci s nesplněním předchozích podmínek vznikne omezení

$$x \leqslant y \wedge y < 0 \wedge x = 0 \tag{4.33}$$

Případ	Podmínka	X	Y	Výstup
(1)	$x > y \wedge x > 0$	1	0	`Exception`
(2)	$x > y \wedge x \leqslant 0$	0	-1	1, $(x - y)$
(3)	$x \leqslant y \wedge y \geqslant 0 \wedge -x + y > 0$	-1	0	`Exception`
(4)	$x \leqslant y \wedge y \geqslant 0 \wedge -x + y \leqslant 0$	0	0	0, $(-x + y)$
(8)	$x \leqslant y \wedge y < 0 \wedge x \neq 0 \wedge 0 \leqslant 0$	-1	-1	0, (0)

Tabulka 4.4: Testovací případy pro program 4.2

které je zjevně nesplnitelné (první dva členy konjunkce omezí x na $x < 0$, což vylučuje $x = 0$). Tím se ztratí 2 z původních 8 průchodů a zároveň symbolická exekuce odhalila nedosažitelný kód, což může indikovat chybu v programu (například zbytečné větvení nebo špatně zformulovaná podmínka).

Nesplnitelný průchod (7) odpovídá situaci, při které není naplněna žádná podmínka v první části funkce a proměnná a proto zůstane rovna 0. Poté bude podmínka pro vyhození výjimky $(0 < 0)$ nesplnitelná. V tomto případě nebyl odhalen nedosažitelný kód, ale existuje podmínka, pro kterou je část metody nedosažitelná. Jedná se o standardní situace a může být například výsledkem ochrany před `NullReferenceException`.

Pro úplné pokrytí metody je potřeba 5 testovacích případů, které jsou vypsány v tabulce 4.4.

Implementace

Symbolická exekuce se v praxi zajišťuje dvěma způsoby. První spočívá v instrumentaci kódu. Proměnné konkrétních typů (například `int`) jsou nahrazené proměnnými typu vyjadřující symbolický výraz (například `Expression`) a operace nad konkrétními typy nahrazeny metodami, které vytváří odpovídající výrok (například operaci `x + y` na `Expression.MakeAdd(x, y)`). Tento nebo podobný přístup je použit v [54], [4] nebo [81].

Druhý přístup vychází z virtualizace prostředí exekuce, které interpretuje instrukce programu přímo nad symboly a výrazy. Hlavní výhoda je totožnost testovaného kódu s reálným kódem (instrumentace jej mění) avšak je nutné vytvořit prostředí, které dokáže testovaný kód interpretovat. Tento přístup je použitý například v [74], [62] nebo [71].

4.3.2 Nedostatky symbolické exekuce

Symbolická exekuce má několik nedostatků. Ty vycházejí z komplexnosti programů a nedostatků schopností vyjádřit a rozhodnout omezení při použití SMT solverů.

Kombinatorická exploze

Symbolická exekuce je podobně jako jiná prohledávání stavového prostoru náchylná na kombinatorickou explozi. Ta nastává ze tří důvodů:

Větvení. Primitivní programy (jako je například program 4.2) neobsahují příliš mnoho větvení. V reálných programech jich je ale podstatně více, což zvětšuje prohledávací prostor. V porovnání vůči jiným příčinám se ale jedná o menší problém[93].

Cykly a rekurze. Cykly a rekurze vedou na značně rychlejší růst stavového prostoru. Každá iterace způsobuje nové větvení a tedy i nové symbolické stavy. Tělo cyklu navíc může být komplikovanější a obsahovat více průchodů. Celkově roste počet symbolických stavů exponenciálně[93] a prohledávání může být i nekonečné, jako například program 4.3[93, 85].

Metody pro eliminaci tohoto problému jsou založené na omezení opakování cyklu, pomocí horní hranice počtu iterací anebo na analýze cyklu - průchodech jeho tělem, podmínkách a proměnných, kterém v něm figurují a případně interagují s programem mimo cyklus. Pomocí těchto analýz se naleznou průchody tělem cyklu tak, aby proběhlo minimální množství iterací. Nalezené průchody se poté rozšíří na celý program. Nástroje pracující tímto způsobem jsou představeny například v [92].

Dynamická data. Poslední příčinou kombinatorické exploze symbolické exekuce jsou dynamická data. Pro primitivní nebo skalární hodnoty je symbolická exekuce přímočará. Pokud symbolická exekuce objeví novou proměnnou, určí její hodnotu jako nový symbol. V případě struktur a objektů (kde hraje roli rovnost lokálních členů) nebo dynamických dat (kde hraje roli i rovnost ukazatelů anebo referencí, tedy identity objektu či instance) nastávají komplikovanější případy[54].

```
1 int Pow(int a, int n) {
2   int z = 1;
3   for (int i = 0; i < n; ++i) {
4     z *= a;
5   }
6   return z;
7 }
```

Program 4.3: Výpočet n-té mocniny. Předpokládá se, že `n` je kladné číslo.

V prvních pracích zabývajících se symbolickou exekucí programů s mutujícími dynamickými daty byla navržena tzv *odložená inicializace* (*lazy initialization*, LI)[54]. Při objevení nové proměnné se pro její inicializaci použije buď jeden z již existujících symbolů (daného typu), hodnota **nil** nebo nový symbol. Tímto přístupem se při objevení nové proměnné exekuce rozvětví na $n + 2$ větví, kde n je počet již známých hodnot daného typu.

Způsobů jak minimalizovat množství nových stavů je několik. Základní technikou je *vázaná odložená inicializace* (*bounded lazy initialization*, BLI), která omezuje hloubku inicializace[31]. Dalším rozšířením je značení existujících symbolů, které se vylučují nebo neovlivňují, za použití SMT solveru, čímž zmenší počet nových stavů[78]. Jiným způsobem je *odloženější inicializace* (*lazier initialization*), při které se struktury inicializují až v momentě, kdy jsou zapotřebí jejich členové[17].

Rozhodovatelnost

Programy, které zatím byly představeny, mají pouze lineární omezení nad celými čísly a jsou proto snadno rozhodnutelné pomocí SMT solverů. Ty ale mají své limity a nejsou potom schopné rozhodnout splnitelnost omezení[41].

Nelineární omezení. Jedním z případů je použití nelineárních omezení. Problém s rozhodnutím by mohl nastat při použití operací jako exponenciála, logaritmus nebo sinus[92]. Moderní SMT solvery, jako třeba *Z3* nebo *CVC5* dokáží pracovat s některými nelineárními omezeními, jako je třeba mocnina[30, 7]. Obecně ale nelineární omezení, která vznikají například při hashování, výpočtu kontrolních součtů (CRC), kryptografii apod, neumí rozhodnout. Tento problém může řešit tzv. *konkolická exekuce* (*concolic execution*)[41], více v kapitole 4.4.

Omezení nad dynamickými daty. Součástí SMT solverů nejsou teorie pro vyjádření obecných omezení nad dynamickými daty - jejich velikostí, tvarem a obsahem. Existují formalismy těchto omezení, například v [16] je popsán jazyk HEX(*heap exploration logic*) pro popis parciálních hald. Rozšířenějším popisem je separační logika představená v části 4.1.4. Ani ta ale není standardní součástí SMT solverů.

Prostředí programu

Účel většiny programů je mimo jiné i interakce se svým okolím (například souborový systém nebo síťová komunikace atp). Programy pro podobné interakce s operačním systémem využívají metod, které nemůže symbolická exekuce vhodně prozkoumat (například systémová volání) a není schopná určit jejich efekt na symbolický stav[85].

4.4 Konkolická exekuce

Výše popsaná symbolická exekuce je přímočarý, ale silný nástroj pro verifikaci programu. Má ale nedostatky, které její využití značně omezují. Řadu z těchto komplikací – rozhodování nelineárních podmínek, zjednodušení podmínky cesty, interakce s prostředím aj. – řeší *konkolická exekuce* (z anglického *concolic*, spojení slov *concrete* a *symbolic*). Jiný název pro tuto techniku je dynamicky symbolická exekuce (*dynamic symbolic exeuction*)[91].

Je to technika, která kombinuje konkrétní a symbolickou exekuci. Při exekuci programu se použijí konkrétní vstupy. Zároveň se ale exekuce programu sleduje symbolicky. Symbolická část exekuce, zejména pak podmínka cesty, se poté využije k určení dalších konkrétních vstupů[82].

4.4.1 Základní algoritmus konkolické exekuce

Program se postupně spouští s konkrétními vstupy. Po každém běhu se ze symbolické exekuce určí podmínka cesty a na základě předešlých běhů a poslední podmínky se určí vstupní podmínky pro další běh. Z těchto vstupních podmínek se pak vytvoří nové konkrétní vstupy (pomocí SMT solveru) a

program se spustí znovu. Tento proces se opakuje, dokud se nedosáhne požadovaného pokrytí[82].

Proces je popsaný v algoritmu 4.1. V této formulaci je možné způsob prohledávání upravit variací metody `GetNextPreconditions`. Základní přístup vychází z postupného otáčení (negování) podmínek na všech rozhodovacích bodech, dokud se nepokryjí všechny varianty. Pokud ale na nějakém rozhodovacím bodě existuje komplexnější podmínka, například $\alpha \wedge \beta$ může její (ne)splnění být způsobeno vícero ohodnoceními (viz část 2.2.3), čímž se z negace jedné podmínky vytvoří více než jedna vstupní podmínka. Pokud vrátí prázdnou množinu, ukončuje prohledávání dané větve.

```
1:  procedure ConcolicExecution(P)
2:      inputs ← {GetDefaultInput(P)}
3:      while ‖inputs‖ > 0 do
4:          input = inputs.Pop()
5:          PC = ConcolicExecute(P, input)
6:          for c in GetNextPreconditions(PC) do
7:              (isSat, model) = CheckSat(c)
8:              if isSat then
9:                  inputs.Push(model)
10:             end if
11:         end for
12:     end while
13: end procedure
```

Algoritmus 4.1: Základní algoritmus konkolické exekuce.

4.4.2 Vlastnosti konkolické exekuce

Oproti symbolické exekuci má konkolická exekuce nástroje pro vyřešení podmínek, které by symbolická exekuce nerozhodla[41]. Uvažujme program 4.4. Program v sobě obsahuje jeden rozhodovací bod (řádek 3). Podmínka je $x^y < z$. Jedná se o nelineární omezení. Nicméně konkolická exekuce může zjistit průběh pro nějaké konkrétní hodnoty, dejme tomu $x := 2$ a $y := 3$. Získá se tak konkrétní hodnota výrazu $x^y = 8$. Poté je možné nahradit omezení $x^y < z$ podmínkou $8 < z \wedge x = 2 \wedge y = 3$, která je lineární a snadno řešitelná[41].

Další výhoda spočívá v práci s prostředím programu nebo kódem, který nelze prozkoumat. Protože v každém stavu jsou k dispozici konkrétní hodnoty, může neprozkoumatelnou část programu spustit s nimi a poté použít výsledek pro následující průzkum. Toto chování je možné rozšířit pomocí implementace symbolických modelů prostředí jako například nástroj *KLEE*[23], který umožňuje práci se symbolickými soubory.

```
1 int Foo(float x, float y, float z) {
2   float a = Pow(x, y);
3
4   if (a < z) {
5     return 0;
6   }
7     return 1;
8 }
```

Program 4.4: Program z primitivními vstupy a nelineárním omezením

Konkolická exekuce nicméně neřeší problémy kombinatorické exploze způsobené komplexností programu (větvení, cykly nebo rekurze) či problémy s dynamickými daty popsané v části 4.3.2.

Prohledávání programu a heuristiky

Konkolickou (a také symbolickou) exekuci je možné popsat jako prohledávání stavového prostoru omezení tvořeného rozhodovacími body v programu. A jako při jiných průzkumech stavového prostoru je možné aplikovat heuristiky. Je zřejmé, že efektivita heuristik výrazně závisí na struktuře programu, jak ukazuje [51].

Nejpoužívanější přístup vychází z prohledávání do hloubky (DFS). Z podmínky cesty, konjunkci omezení vycházející z jednotlivých rozhodovacích bodů, se vybere poslední zatím neprozkoumaná alternativa (omezení se zneguje) a spolu s předchozími omezeními tvoří novou podmínku cesty (například v [40] a [80]).

Tato strategie umožňuje dobře využít optimalizací řešení podmínek cesty (pokud známe řešení formule ϕ – model dokazující její splnitelnost – tak lze snáze rozhodnout formuli $\phi \wedge \phi'$, zejména pokud je ϕ' atomická formule)[22]. Nevýhoda ale spočívá v pomalém objevování nových částí programu. Následující iterace prozkoumá totožný průchod a tedy prozkoumá totožnou část programu, s výjimkou konce. Jednoduchá úprava základní strategie spočívá v omezení hloubky (počet větvení)[22]. Existuje řada prací, které zkoumají heuristiku pro konkolickou exekuci, viz část 5.2.

Kromě strategií na zefektivnění průzkumu programu jsou i metody na zmenšení samotného prostoru. V práci [53] je využit Craigův interpolant, který dokáže zjednodušit nesplnitelné formule a pomocí této informace následně

usuzuje nesplnitelnost dalších omezení a tím efektivně zmenšuje prostor k průzkumu.

V nástroji *SMART* (rozšíření nástroje *DART*), představeném v [41], se komplexní program dekomponuje na jednotlivé metody, které se poté prohledávají separátně. Z průzkumu se vytvoří jejich sumarizace (vztah mezi vstupy a výstupy). Pokud se má při průzkumu zavolat již prozkoumaná metoda, tak se místo jejího volání využije její sumarizace pro zjištění návratové hodnoty.

Kapitola 5

Související práce

Existuje řada nástrojů pro symbolickou a konkolickou exekuci, mířených na řadu programovacích jazyků (C/C++, Java, JavaScript, C# ...) a ekosystémů (x86, *java virtual machine* (JVM), .NET, WASM, ...), neúplný přehled je zpracovaný v [95]. Třebaže je symbolická exekuce studována již od 70. let minulého století ([55]), zaznamenáváme její prudký rozvoj až v posledních 2 dekádách, společně s rozvojem SMT solverů.

5.1 Nástroje symbolické a konkolické exekuce

Mezi první nástroje konkolické a symbolické exekuce patří *DART* z roku 2005 [40], který provádí první varianty konkolického prohledávání. Jeho rozšíření *SMART* z roku 2007 [41] pak umožňuje systematicky prohledávat komplexní programy pomocí dekompozice na jednotlivé metody, pro které vypracuje sumarizaci. Tu poté použije místo zavolání právě zkoumanou metodou.

Zároveň s *DARTem* vzniká i *CUTE*[80]. Oproti *DARTu* umožňuje vytvářet omezení i nad aritmetikou ukazatelů, nicméně nepoužívá systematickou strategii pro prozkoumávání jednotlivých metod v izolaci nástroje *SMART*. Existuje i jeho varianta pro Javu *jCUTE*[81].

Podobným nástrojem je *EXE* [24], který dokáže pracovat s čistě symbolickou aritmetikou ukazatelů. Rozšiřuje jej práce [14], která eliminuje symbolické stavy na základě přístupů do symbolické paměti.

Jeho nástupcem je pak *KLEE*[23], který obsahuje řadu optimalizací v oblastech řešení omezení a reprezentace stavů, využívá heuristiky pro maximalizaci pokrytí kódu a vhodným způsobem simuluje chování prostředí testovaného programu. Tento nástroj je pak rozšiřován třeba v práci [51] nebo [47], které pro volbu heuristiky používají techniky strojového učení.

Jedním z nejpoužívanějších nástrojů pro nízkoúrovňové jazyky, respektive instrukční sadu x86, je *SAGE* z roku 2008, představený v [42] a [43]. Provádí fuzzing testování podpořené konkolickou exekucí a míří zejména na parsovací programy. Oproti předchozím nástrojům konkolické exekuce (*DART* a *CUTE*) nepracuje se zdrojovým kódem, ale strojovým kódem, což značně rozšiřuje jeho pole působnosti. Zásadní problém – kombinatorická exploze způsobená mnoha větveními a cykly – řeší pomocí tzv. *generačního generování vstupů*, kde z jednoho běhu programu vygeneruje více testových vstupů najednou.

Podobným nástrojem je *DRILLER*[87] nebo *Legion*[61], který využívá kombinace fuzzing testování a konkolické exekuce k prozkoumání co nejvíce částí programu. To zároveň kombinuje s Monte Carlo algoritmem prohledávání stromu.

Důležitým krokem ve vývoji symbolické exekuce byla v roce 2003 práce [54], která zavedla princip *odložené inicializace*, čímž zobecnila symbolickou exekuci ze skalárních hodnot na komplexní struktury, což je nutným předpokladem pro symbolickou exekuci programů napsaných v jazycích vyšší úrovně, jako třeba Java nebo C#. Tento algoritmus byl rozšířen v dalších pracích. V roce 2006 v práci [31] (*KIASAN*) byla přidána alternativa s omezením na velikost či hloubku struktur na haldě a princip *odloženější inicializace* nebo v roce 2016 v práci [78] (*BLISS*), která kombinuje několik strategií pro snížení kombinatorické exploze vycházející z odložené inicializace.

V oblasti objektově orientovaných programovacích jazyků je jedním z prvních nástrojů *Symstra*[100]. Dalším je *PET*[1], který provádí symbolickou exekuci Java bajtkódu který převedou na ekvivalentní CLP (*constrained logic program*). *jPET*[2] poté nalezené testovací případy převádí do Javy.

Řada nástrojů pro symbolickou či konkolickou exekuci Java programů vzniklo jako nadstavba model checkeru *Java PathFinder* (*JPF*)[97]. Mezi jeho symbolické nadstavby patří již zmíněná práce [54] nebo [98, 4], které ale využívají instrumentaci kódu, místo virtuálního prostředí poskytnutého *JPF*. V roce 2012 vznikl nástroj *Symbolic PathFinder*[71], který již nevyužívá instrumentaci kódu a upravuje přímo prostředí exekuce. Kromě toho používá i metodu konkrétně-symbolického řešení omezení, ve kterém využívá cíleně konkrétní exekuce pro zjednodušení omezujících výrazů. Práci s reálnými čísly v něm rozšiřuje [15] pomocí kombinace řešení intervalů a meta-heuristiky.

Konkolickou exekuci pro Javu pomocí JPF pak mimo jiné implementuje nástroj *JDart*[62], vyvinutý pro NASA a verifikaci *mission ciritical* systémů. Poskytuje modulární architekturu, která umožňuje jeho snadné rozšiřování, například práce [50].

Pro programy, které jsou založené na manipulaci struktur na haldě, je nicméně potřeba poskytnou formální systém pro uvažování nad velikostí, tvarem a obsahem haldy. V [18] je představen nástroj JBSE, který využívá HEX (*heap exploration logic*) pro formulaci omezení nad haldou. V nástroji *Java StarFinder*[74] a jeho konkolické variantě *Java CoStarFinder* [73] se pro omezení využívá separační logika, zejména pak její indukční predikáty, pomocí které také provádí testování na základě specifikace.

Symbolickou exekuci s ověřování modelu kombinují v [3], kde řeší kombinatorickou explozi z neomezeně velkých struktur na haldě pomocí eliminace stavů, které jsou zahrnuty v již prozkoumaných stavech.

Nevznikají ale pouze nástroje mířené na Java ekosystém. Nástroj *Pex* [91] provádí konkolickou (respektive dynamicky symbolickou) exekuci .NET programů, a přímo generuje testovací kód. Dnes se *Pex* vyvinul do nástroje *IntelliTest*, který je součástí IDE Visual Studio. *Pex* byl rozšiřován v rámci několika dalších prací, například [5] nebo [90].

Pro ověřování modelu .NET programů existuje nástroj *MoonWalker*, viz 6.1). Obdobnou funkcionalitu nabízí také nástroj *XRT*[45].

Pro JavaScript vznikl v roce 2013 nástroj *Jalangi*[83], který kromě konkolické a symbolické exekuce obsahuje další nástroje, například allocator profiler nebo detektor neslučitelnosti typů. Práce [32] řeší škálovatelnost konkolického testování na omezení vycházející z typů proměnných, což může být v případě JavaScriptu, který umožňuje dynamické typování, problém kvůli vytváření řady redundantních omezení[32]. Problém řeší pomocí systému pro udržení povědomí o typech.

Nástroj *StackFull*[94] kombinuje konkolické testování klientské a serverové části JavaScriptové aplikace. Díky tomu je možné propagovat omezení z klientské části do části serveru a obráceně.

Pro webové aplikace, respektive technologii WASM, vznikl například nástroj *WASP* (*web assembly symbolic processor*)[65].

Ve specializovaných oblastech s konkolickou exekucí pracuje, například práce [6], která zkoumá její využití pro testování aplikací na mobilní zařízení, [36] který popisuje konkolickou exekuci logických programů, [88] jí aplikuje na testování neuronových sítí nebo [60] a [59] využívají konkolickou exekuci pro ověření *MPI* programů.

5.2 Heuristiky

Důležitou komponentou konkolické exekuce je strategie prohledávání omezení, pro kterou je možné využít heuristik. Základní přístup založený na DFS má několik výhod (viz část 4.4.2), nicméně pro efektivní průzkum, za cílem pokrytí, je vhodnější použít jiné přístupy[22].

V [84] je použitá strategie založená na prohledávání do šířky (BFS) (oproti základní strategii se volí první neprozkoumaná alternativa), ta je poté doplněná o další rozhodování na základě předchozích průběhů. Zde se nové části programu objeví rychleji, nicméně lze obtížněji využít optimalizace při řešení podmínek cest.

Dalším způsobem, jak snížit pravděpodobnost „uvíznutí“ v prozkoumané oblasti, je tvořit novou vstupní podmínku pomocí náhodně vybrané neprozkoumané podmínce[22]. Tento přístup je použit v rozšíření CUTE[63].

Komplikovanější strategie (například *context-guided search*[22]) používají pro maximalizaci prozkoumaných částí programu analýzu CFG a statického grafu volání. Průzkum poté sleduje pokrytí CFG a volí taková omezení, aby pokryl co možná nejvíce hran v CFG[22].

Nástroje *LEARCH*[47] a *ParaDySe*[26, 51], využívají automatické generování heuristik pro konkolické prohledávání, které jsou pak přizpůsobené konkrétním programům.

5.3 Formulace testového kódu

Problematika pro generování testovacích vstupů pro daný program použitím symbolické či konkolické exekuce je široce studované téma. Problém generování souboru testů, respektive spustitelného testovacího kódu, není řešený příliš do hloubky. Mezi nástroje, které za tím účelem byly vytvořené patří *Seeker* ([90]), nadstavba *Pexu*, která využívá statickou analýzu pro generování posloupnosti konstruktoru a volání validních metod pro inicializaci, přičemž připouští, že pro některé testovací vstupy není schopný najít řešení. Takové testovací vstupy považuje za nevalidní, protože neodpovídají vlastnostem dat vycházející z definic tříd.

Dalším nástrojem je *SUSHI* ([18]), nadstavba *JBSE*, který problém převádí na optimalizační problém lineárního programování. Snaží se pak najít sekvenci konstruktoru a volání validních metod, které inicializují vstupní data tak, aby jejich vlastnosti co nejvíce odpovídaly vlastnostem vstupů nalezených předchozí konkolickou exekucí.

Čitelnost jednotkových testů analyzuje práce [28], která formuluje model čitelnosti pomocí které upravuje výstup nástroje *EvoSuite* [38], evolučního nástroje na generování testovacích případů pro Javu.

Pro Javu je také vyvíjen komerční nástroj *Diffblue*[102] v rámci IDE IntelliJ IDEA, který pro generování jednotkových testů využívá umělou inteligenci.

Kapitola 6

Implementace

Výstupem této práce je nástroj pro generování jednotkových testů pro .NET programy za využití model checkeru. V této kapitole jsou popsány implementační detaily výstupu práce a to nástroje *dnWalker*.

dnWalker se pak skládá ze 2 programů:

1. *dnWalker* - nástroj pro konkolickou exekuci .NET programů, skládající se ze dvou komponent:
   (a) *MoonWalker*[79, 69] - model checker pro .NET, rozšířený v rámci této práce,
   (b) Konkolický průzkumník - sada rozšíření a nadstavba nad *MoonWalkerem*, které zajišťují konkolickou exekuci
2. *dnWalker.TestWriter* - nástroj pro generování souboru testů (*test suite*) v jazyce C#,

Implementace navazovala na předchozí vývoj, ve kterém proběhlo rozšíření *MoonWalkeru* o lokaci služeb (viz níže) a částečná implementace konkolické exekuce, podporující primitivní hodnoty, přičemž pro symbolické výrazy byla využita standardní knihovna *LINQ*. Tato implementace byla předělána pomocí vlastní knihovny pro reprezentaci symbolických výrazů. Předchozí vývoj zavedl použití SMT solveru *Z3*[30] a knihovny *dnlib*[103].

6.1 MoonWalker - model checker pro CLI

MoonWalker (původně *MonoModelChecker*) je model checker vyvíjený na univerzitě Twente [79, 69]. Aplikuje metody ověřování modelu explicitního stavu

(viz část 4.2.2) na programy vytvořené pro *Common Language Infrastructure* (CLI), viz následující sekce. Jeho struktura je podobná struktuře *JPF* (*java path finderu*, model checker pro JVM[98]).

Umožňuje odhalit porušení vlastností bezpečnostni: *Assert*, nechycené výjimky a problémy s paralelismem – zablokování (*deadlock*) a souběh (*race condition*)[79]. Kromě POR implementuje i redukci založenou na symetrii haldy[69].

6.1.1 Common language infrastructure (CLI)

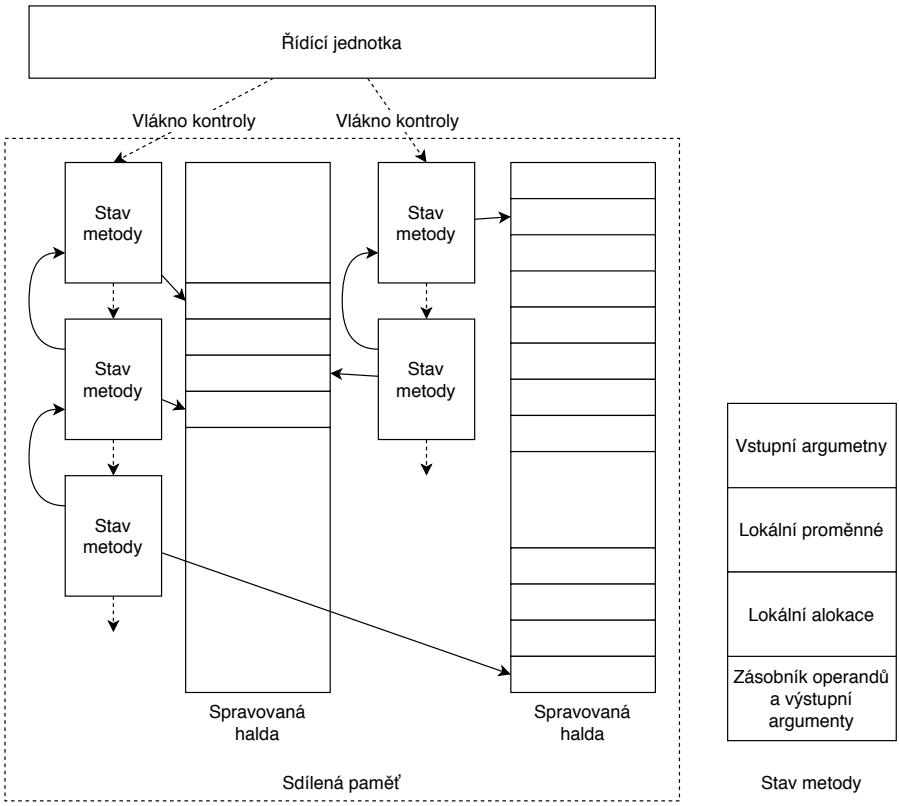
Common language infrastrucutre (CLI) je specifikace pro programovací jazyky a exekuční prostředí, za účelem vývoje platformě nezávislých programů[101]. Nejrozšířenější implementací CLI je .NET[105] a Mono[109].

CLI formalizuje systém typů (CTS, *common type system*), který specifikuje dělení typů a možnosti definování nových. Dále poskytuje pravidla pro programovací jazyky (CLS, *common language specification*), pomocí kterých se zajistí jejich interoperabilita. Spustitelný kód (definice typů, metod atp) je strukturován do sady tabulek (metadat). Ty jsou poté zdrojem informací pro *virtuální exekuční systém* (VES, *virtual execution system*) – prostředí, které spouští kód a poskytuje podporu pro vestavěné typy a další služby[101].

CLI popisuje abstraktní model exekuce (viz obrázek 6.1), který pracuje s pamětí uspořádanou do jedné či více spravovaných hald. Exekuce je rozdělená do jednoho či více *vláken řízení* (*thread of control*). Vlákno řízení je sekvence *stavů metody* (*method state*). Stav metody obsahuje pole *vstupních argumentů*, *lokálních proměnných*, *lokálních alokací* a *zásobník operandů*, který zároveň slouží pro předání *výstupních argumentů* volajícím metodám. Konkrétní reprezentace stavu exekuce je pak záležitostí jednotlivých implementací.

Tělo metody, sekvence instrukcí, je popsáno pomocí CIL (*common intermediate language*), což je zásobníkový, objektově orientovaný bajtkód programovací jazyk. Neúplný seznam instrukcí, které jsou použité v tomto textu, je v příloze A.

Pro vytvoření *modulu* CLI (ucelené jednotky interpretovatelné pomocí VES je nutné zajistit kompilaci zdrojového kódu právě do CLI a zajistit dodržení CLS. Existuje řada jazyků (nebo jejich variant), které toto umožňují. Nejpoužívanějším jazykem pro CLI je C# vyvíjený Microsoftem. Jiné jazyky



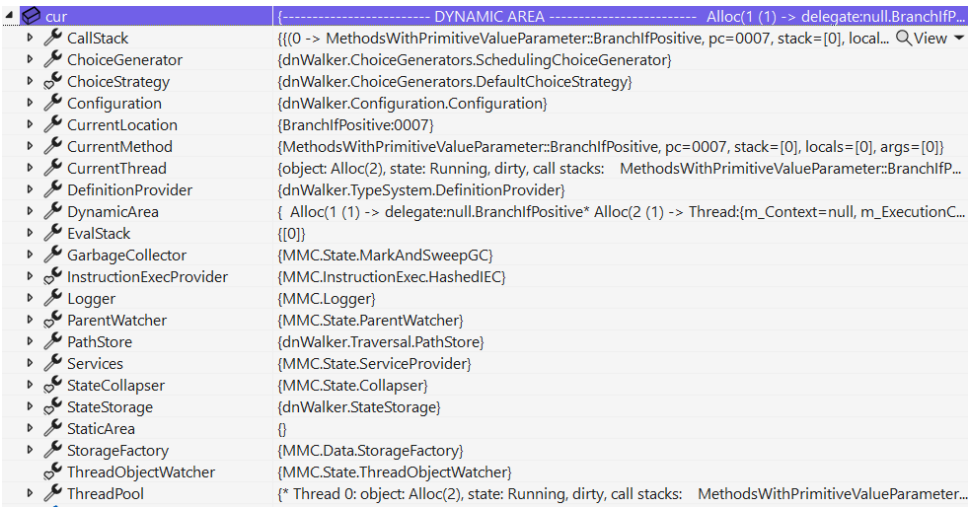
Obrázek 6.1: Stav exekuce v CLI.

jsou například F#, varianta ML[1] programovacího jazyka, nebo C++/CLI, verze C++[2].

Provedení CLI je pak záležitostí konkrétní implementace VES. Například .NET provádí JIT kompilaci do strojového kódu, který pak spouští ve standardním procesu operačního systému pomocí CLR (*common language runtime*), respektive v novějších verzích CoreCLR. V případě *MoonWalkeru* (a tedy i nástroje *dnWalker*) je každá instrukce interpretována na úrovni CLI a její efekt aplikován na model stavu exekuce (viz následující části).

[1]ML, *meta language* – funkcionální programovací jazyk, `https://cs.lmu.edu/~ray/notes/introml/`

[2]Širší seznam CLI jazyků je k dispozici na `https://en.wikipedia.org/wiki/List_of_CLI_languages`



Obrázek 6.2: Náhled vlastností třídy `ExplicitActiveState`

6.1.2 Obecný popis MoonWalkeru

MoonWalker je implementace VES. Všechna data jsou reprezentována pomocí datových elementů (implementací rozhraní `IDataElement`), přičemž pro každý typ popsaný ve specifikaci ([101]) je poskytnuta implementace.

Základní komponentou *MoonWalkeru* je třída `ExplicitActiveState`, která obsahuje všechny informace o exekuci (mimo jiné stav programu) a prohledávání (viz obrázek 6.2). Stav programu je pak reprezentovaný spravovanou pamětí, rozdělenou na dynamickou a statickou oblast, skupinou vláken, které každé přesně implementuje model ukázaný na obrázku 6.1. Dynamická oblast obsahuje všechny dynamické alokace (objekty a pole). Statická oblast obsahuje statické alokace – načtené třídy a jejich statické *fieldy*[3].

Exekuce je zajištěna pomocí exekutorů instrukcí. Pro každý typ instrukce je implementovaná třída, která zajistí její exekuci (modifikace stavu programu), ale poskytuje další informace, například označí které oblasti paměti využívá (čte anebo zapisuje), což je nezbytná informace pro POR a detekci chyb spojených s paralelismem.

Podrobnější informace o vlastnostech, architektuře a implementaci *MoonWalkeru* jsou v pracích [79, 69].

[3]Třídní proměnnou CLI nazývá *field*. Při překladu tohoto termínu by docházelo k nejasnosti se zavedeným českým termínem pro *array*, proto je použit originální termín.

V této práci byl *MoonWalker* modifikován, aby jej bylo možné na několika místech rozšířit a tím zajistit konkolickou exekuci. Jedná se o úpravu komponent `ExplicitActiveState`, exekutoru instrukcí a refaktorizace mechanismu obsluhy volání vestavěných metod.

6.1.3 Sdílení služeb

Jednotlivé komponenty konkolického průzkumníku vyžadují přístup ke sdíleným službám. Ten jim je poskytnut právě přes `ExplicitActiveState`, který se předává napříč exekucí v *MoonWalkeru*. Tyto služby existují ve dvou kontextech - jedna exekuce (iterace) a celkové prohledávání.

Pro rozlišení jednotlivých exekucí vznikla komponenta `PathStore`, která spravuje informace o jednotlivých cestách v prohledávaném programu, a tedy odpovídá jedné exekuci. Přes jednotlivé `Path` objekty jsou poskytnuté rozšiřující informace.

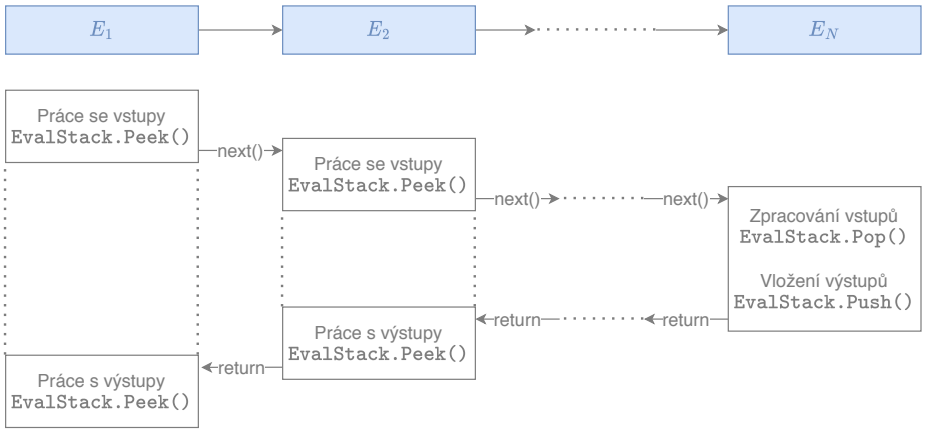
Samotný `ExplicitActiveState` byl rozšířený o jednoduchý lokátor služeb, který umožňuje registraci a rezoluci služeb pomocí jejich typů.

6.1.4 Exekutor instrukcí

V původní verzi *MoonWalkeru* je možné děděním třídy `InstructionExecBase` a implementací rozhraní `IInstructionExecutorProvider` (které pro danou instrukci poskytne instanci exekutoru) nahradit chování jednotlivých instrukcí vlastní implementací.

Nicméně pro zajištění konkolické exekuce bylo aplikováno flexibilnější řešení, které umožňuje rozšířit chování jednotlivých instrukcí pomocí modulárních rozšíření. Tím je možné komplexní chování implementovat pomocí sady jednodušších jednotek, které poskládají *pipeline*.

Rozšíření jsou implementována pomocí rozhraní `IInstructionExecutor` (viz program 6.1). To obsahuje způsob pro identifikaci podporovaných operačních kódů (vlastnost `SupportedOpCodes`) a poté metodu `Execute`, pomocí které se provede samotná exekuce. Kromě kontextu instrukce (parametry `baseExecutor` – informace o instrukci a `cur` – informace o stavu programu)



Obrázek 6.3: Vnořená exekuce rozšíření. Kontrola dalším exekutorům se předává zavoláním delegátu `next` a vrací pomocí instrukce `return`.

obsahuje i metodu `next`, pomocí kterého převede kontrolu na zbytek pipeliny, čímž dochází ke vnořené exekuci dalších rozšíření (podle obrázku 6.3). Dá se tak snadno zpracovat vstupní data (obsah zásobníku před zavoláním metody `next`) a výstupní data (obsah zásobníku po zavolání metody `next`). Posledním členem pipeliny je pak základní implementace instrukce, která zpracovává pouze konkrétní vstupy.

```
1 public interface IInstructionExecutor {
2     public IEnumerable<OpCode> SupportedOpCodes { get; }
3
4     public IIEReturnValue Execute (
5         InstructionExecBase baseExecutor,
6         ExplicitActiveState cur,
7         InstructionExecution next);
8 }
```

Program 6.1: Rozhraní `IInstructionExecutor`, které implementují všechna rozšíření instrukcí.

Program 6.2 představuje takové rozšíření, které zpracovává symbolickou exekuci instrukce `brfalse`. Pokud je hodnota na zásobníku symbolická hodnota, provede aktualizaci podmínky cesty a přidá nová omezení do prostoru omezení pomocí metody `DecisionHelper.JumpOrNext` (viz části 6.2.2 a 6.2.4).

```
 1 public IIEReturnValue Execute(
 2   InstructionExecBase baseExecutor,
 3   ExplicitActiveState cur,
 4   InstructionExecution next) {
 5
 6   // read the value without modifying the stack
 7   IDataElement operandDE = cur.EvalStack.Peek();
 8
 9   // perform the concrete execution
10   IIEReturnValue retValue = next(baseExecutor, cur);
11
12   if (!operandDE.TryGetExpression(
13       cur,
14       out Expression expression)) {
15
16     // the value is not symbolic => do nothing
17     return retValue;
18   }
19
20   // value is symbolic, use the expression to describe decisions
21   DecisionHelper.JumpOrNext(
22     cur,
23     retValue,
24     Expression.MakeNot(expression.AsBoolean()),
25     expression.AsBoolean());
26
27   return retValue;
28 }
```

Program 6.2: Implementace metody `Execute` v symbolickém rozšíření `brfalse`.

6.1.5 Náhrada vestavěných a externích metod

Podle specifikace CLI musí VES poskytovat podporu pro řadu vestavěných typů, respektive metod. Jedná se například o `System.String` nebo `System.Math`. *dnWalker* proto musí napodobit chování těchto typů.

Třebaže má *dnWalker* k dispozici implementaci těchto typů zprostředkovanou základními (*core*) moduly, je mnoho z nich závislých na nativních metodách, definovaných v knihovnách operačního systému. Proto je nutné řadu z těchto metod plně nahradit manuální změnou stavu programu (operace nad `ExplicitActiveState`).

Za tím účelem je implementovaný systém tzv. *native peer*, podobně jako v *JPF*[4]. V *dnWalker* jsou pomocí rozšíření instrukcí `call`, `callvirt` a `calli`

[4] `https://github.com/javapathfinder/jpf-core/wiki/Model-Java-Interface`

odchycena volání vestavěných metod a místo nich spuštěna obslužná rutina, která vhodně upraví stav programu, například pro případ metody `String::IsNullOrEmpty` vloží na zásobník **true**, pokud je argument typu `string` roven `null` nebo má délku 0, v opačném případě **false**.

U řady z těchto obsluh se s argumenty přímo použijí vestavěné typy a metody (například pro obsluhu třídy `Math`), ale pro jiné typy (například `Thread`) je nutné interagovat se stavem programu.

6.2 Konkolická exekuce

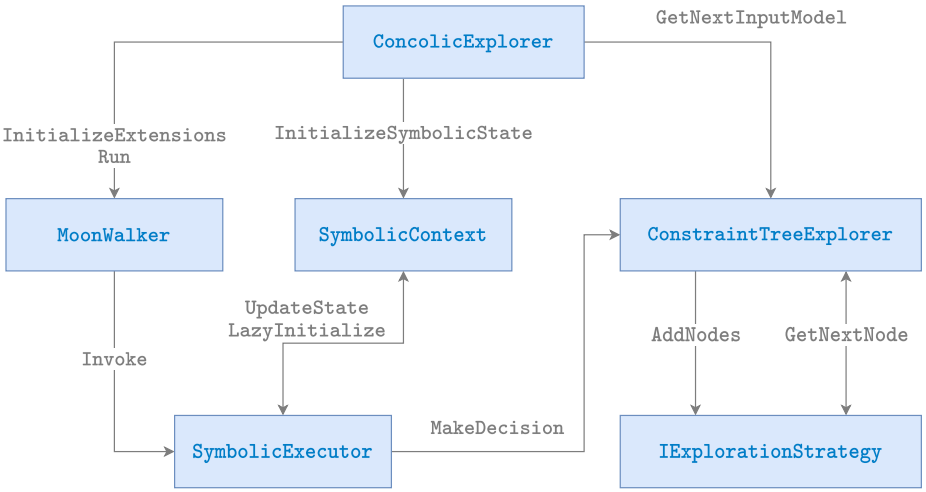
Systém, který řídí o konkolickou exekuci je znázorněn na obrázku 6.4. Vstupním bodem je třída `ConcolicExplorer`, která inicializuje zbylé komponenty a iterativně spouští konkolickou exekuci programu. Konkrétní exekuce je zajištěna model checkerem *MoonWalker*. Symbolická exekuce je zprostředkována přes sérii rozšíření instrukcí (viz sekce 6.2.4, které komunikují se třídou `SymbolicContext`, která obsahuje informace o symbolickém stavu aktuálního běhu programu. Před spuštěním exekuce se symbolický stav inicializuje podle vstupních podmínek dané iterace.

Další důležitou součástí je `ConstraintTreeExplorer`. Tato třída má na starosti správu prostoru omezení v programu (viz sekce 6.2.2). Prostor omezení roste při průzkumu metody pomocí symbolické exekuce, která nová omezení přes `ConstrintTreeExplorer` předávají běžící `IExplorationStrategy`. Ta se stará o jejich zpracování a rozhoduje kdy a zda se použijí při formulaci nových konkrétních vstupů (viz 6.2.3).

6.2.1 Struktura metody – CFG

Pro orientaci v testované metodě využívá *dnWalker* graf toku řízení (CFG). Jak bylo zmíněno v části 2.2.3, kromě standardních větvení způsobených řídícími strukturami (`if - then - else`, `for`, atp.) pracuje *dnWalker* také s větvením způsobeným vyhozením výjimek při neplatném stavu programu pro danou instrukci.

Existuje proto mapování operačních kódů (typů instrukcí) na množinu výjimek, které tyto instrukce mohou vyhodit. Pokud se při tvorbě CFG



Obrázek 6.4: Struktura konkolického průzkumníka a vztahy mezi jednotlivými komponentami.

narazí na instrukci, která může potenciálně vyhodit výjimku, tak se pro danou výjimku vytvoří nový uzel, který reprezentuje její obsluhu, a vhodně se připojí ke zbytku CFG.

dnWalker vytváří CFG z CLI. Uvažujme například program 6.3, který obsahuje CLI vzniklý kompilací programu 3.3 a na obrázku 6.5 je ukázaný výsledný CFG, kde je vidět, jak instrukce `callvirt` a `ldfld` mohou vyhodit výjimku `NullReferenceException` pokud je instance na zásobníku `null`.

Podobně systém pracuje i s native peers. Pokud instrukce `callvirt` nebo `call` má zavolat metodu obslouženou přes native peer, tak se CFG rozšíří o výjimky, které daná metoda může vyhodit.

CFG se skládá ze dvou typů uzlů: `InstructionBlock` a `VirtualExceptionHandler`. První reprezentuje blok jedné či více po sobě jdoucích instrukcí a je zakončen instrukcí, která má více následných instrukcí (`brfalse`), může vyhodit jednu či více výjimek (`ldelem`, `add.ovf.un`) nebo její (jediný) následník nemusí být bezprostředně následující instrukce (`throw`, `br`). `VirtualExceptionHandler` je uzel, který se použije, pokud neexistuje konkrétní obsluha pro danou výjimku. Pro každý typ výjimky existuje právě jeden.

CFG také rozlišuje hrany. Základní jsou `Next` a `Jump` hrany, které reprezentují přechod k bezprostředně následující instrukci, respektive skok na jiný adresu. Ty jsou doplněné o `Exception` hranu, která představuje reakci na vyhození dané výjimky. Posledním typem hrany je `AssertionViolation`,

```
.method public hidebysig instance int32 Foo (
    class IBar bar) cil managed {
  .maxstack 8

  IL_0000: ldc.i4.2
  IL_0001: ldarg.1
  IL_0002: callvirt instance int32 IBar::GetValue()
  IL_0007: mul
  IL_0008: ldc.i4.3
  IL_0009: sub
  IL_000a: ldarg.0
  IL_000b: ldfld int32 SimpleMethod::Value
  IL_0010: bne.un.s IL_0014

  IL_0012: ldc.i4.4
  IL_0013: ret

  IL_0014: ldc.i4.5
  IL_0015: ret
}
```

Program 6.3: CLI verze programu 3.3, vstup pro vytvoření CFG.

která se využije pouze v případě volání metody `Debug::Assert`.

6.2.2 Prostor omezeni

Základním vstupem do jedné iterace konkolického prohledávání je omezení nad vstupními daty – vstupní podmínky (*precondition*). Tato omezení jsou vyjádřena pomocí formulí separační logiky a jsou vybudována pomocí předchozích iterací průzkumu v implementaci rozhraní `IExplorationStrategy`. Všechna omezení, která jsou v průběhu konkolického prohledávání objevena, se ukládají do tzv. *stromu omezení* (*constraint tree*).

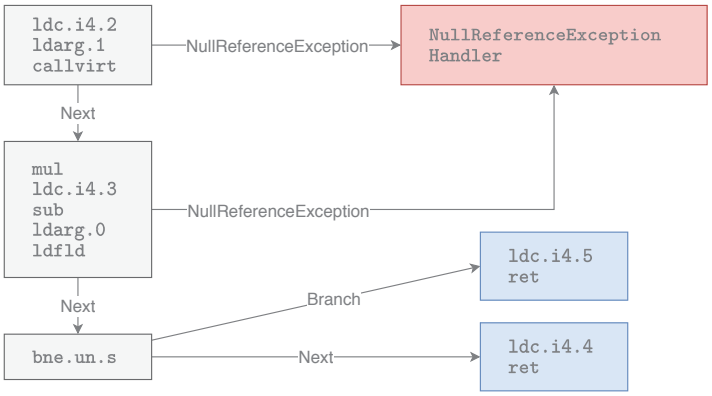
Uvažujme program 6.4, který hledá minium ze tří čísel. Tímto programem lze projít 4 různými způsoby (viz CFG na obrázku 6.6). Vzniklé podmínky jsou:

$$a_0 > b_0 \quad \wedge \quad b_0 > c_0 \quad \wedge \quad \texttt{return}\ c_0 \tag{6.1}$$

$$a_0 > b_0 \quad \wedge \quad b_0 \leqslant c_0 \quad \wedge \quad \texttt{return}\ b_0 \tag{6.2}$$

$$a_0 \leqslant b_0 \quad \wedge \quad a_0 > c_0 \quad \wedge \quad \texttt{return}\ c_0 \tag{6.3}$$

$$a_0 \leqslant b_0 \quad \wedge \quad a_0 \leqslant c_0 \quad \wedge \quad \texttt{return}\ a_0 \tag{6.4}$$



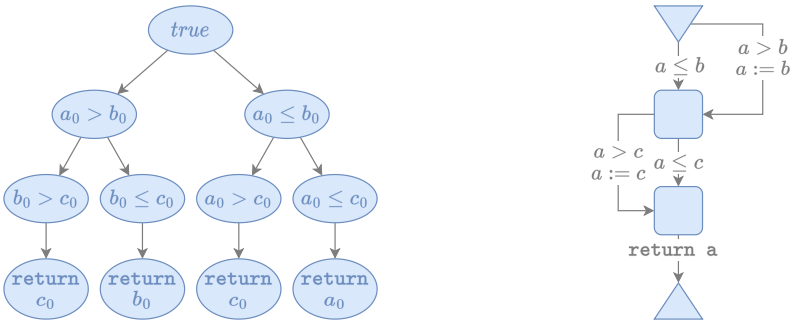
Obrázek 6.5: CFG pro program 6.3 rozšířený o výjimky.

```
1 int Min3(int a, int b, int c) {
2   if (a > b) {
3     a = b;
4   }
5   if (a > c) {
6     a = c;
7   }
8   return a;
9 }
```

Program 6.4: Minimum ze tří hodnot

Podmínky lze přestavět do stromu omezení zobrazeném na obrázku 6.6. Strom omezení má 4 listy, které odpovídají 4 cestám skrz program. Formule 6.1 až 6.4 vzniknou jako konjunkce omezení ve vrcholech v cestě z kořene do listů ve stromu omezení. Každý uzel stromu omezení odpovídá hraně v CFG, s výjimkou kořene, která obsahuje vstupní podmínku. Na příkladě na obrázku 6.6 není žádné omezení nad vstupními argumenty, tudíž vstupní podmínka je prázdná, nebo **true**. Mohla by ale být nastavena na libovolné omezení, například $a \leqslant 0 \wedge b \neq 3$.

Uzel stromu omezení (uzel omezení) má odkaz na svého předka (nebo `null` pokud to je kořen), hranu v grafu toku řízení (nebo `null` pokud není spojen s žádnou hranou CFG, například pokud to je kořen), formuli, která vyjadřuje jeho omezení a pole potomků. Uzel má několik dalších vlastností. Předně obsahuje indikátor, jestli již byl prozkoumán. Další indikátor určuje, jestli



Obrázek 6.6: Strom omezení a CFG 6.4.

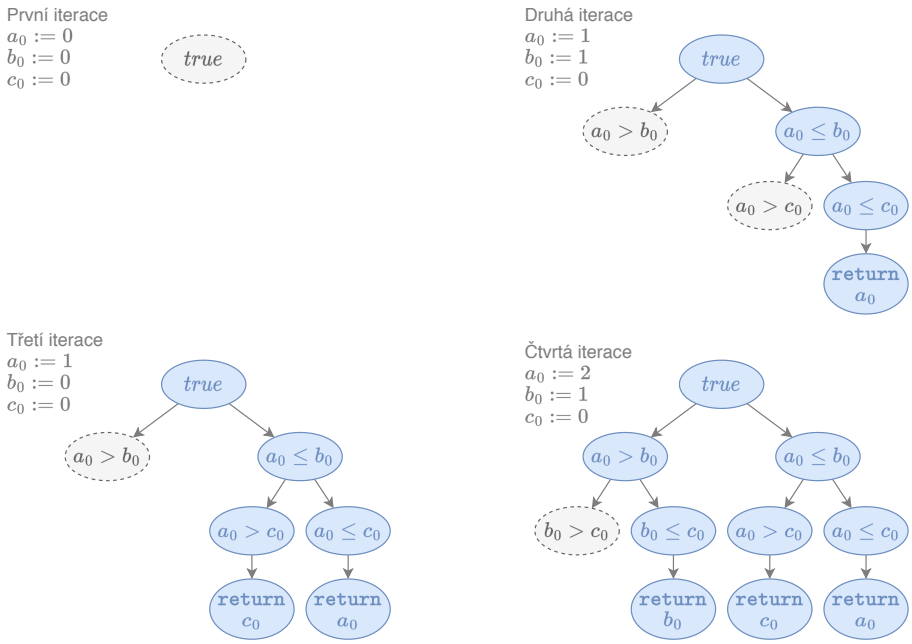
popisuje nesplnitelnou nebo nerozhodnutelnou podmínku.

Pokud je nad vstupy do metody definováno nějaké omezení, vyjádří se v kořenovém uzlu. Omezení vstupních podmínek bývá často vyjádřeno jako disjunkce více podmínek. V takovém případě se pro každý disjunkt vytvoří vlastní strom omezení a celý prostor omezení pak tvoří les těchto stromů. Tento les spravuje třída `ConstraintTreeExplorer`. Ta má přístup ke všem `ConstraintTree` a udržuje informaci o aktuální pozici v aktuálně prohledávaném stromu omezení (uzel omezení).

Strom omezení se buduje v průběhu exekuce. Pokud exekuce narazí na instrukci, jejíž chování je podmíněné daty na zásobníku, tak se pro každou variantu vytvoří nový uzel omezení a vloží se do stromu omezení. Třída `ConstraintTreeExplorer` toto zprostředkovává pomocí metody `MakeDecision`, popsané v algoritmu 6.1. Tato metoda vytvoří nové uzly omezení, pokud je to nutné, a aktualizuje pozici ve stromu omezení.

```
1:  procedure MakeDecision(d, c̄, ē, cn)
2:      if cn.IsExpanded then
3:          for (c_i, e_i) in (c̄, ē) do
4:              cn.Children.Add((c_i, e_i))
5:          end for
6:          cn.IsExpanded = true
7:          UpdateStrategy(cn.Children)
8:      end if
9:      return cn.Children[d]
10: end procedure
```

Algoritmus 6.1: Metoda `MakeDecision`. d je index, který určuje aktuálně provedené rozhodnutí. $\bar{c}$ je vektor disjunktních podmínek, pro které platí, že splnění podmínky c_i způsobí na daném bodu rozhodnutí i. $\bar{e}$ je vektor hran v grafu toku řízení, kde hrana e_i odpovídá rozhodnutí i. cn je aktuální uzel ve stromě omezení.



Obrázek 6.7: Růst stromu omezení pro program 6.4. Stav stromu před i-tou iterací.

Příklad tvorby stromu omezení

Růst stromu omezení ilustrujme na případě prohledávání metody 6.4. Jeho změny v průběhu 4 iterací průzkumu znázorňuje obrázek 6.7.

Před první iterací obsahuje strom pouze jeden uzel s podmínkou **true**. Na řádku 2 dojde k rozšíření a vloží se nové dva uzle: $a_0 > b_0$ a $a_0 \leqslant b_0$, přičemž byl zvolen druhý z nich (první není splněný). Na řádku 5 dojde k podobné situaci. Proto před druhou iterací má strom celkem 6 uzlů a dva z nich nejsou prozkoumané.

Pro druhou iteraci byl zvolen uzel s podmínkou $a_0 > c_0$ a do stromu byl přidán pouze jeden nový uzel (`return` c_0). Pro třetí iteraci zbývá pouze uzel $a_0 > b_0$ a tím se prozkoumá druhá část stromu a přidají 3 nové uzly.

Ve čtvrté iteraci se prozkoumá poslední neprozkoumaný uzel a výsledný strom odpovídá tomu na obrázku 6.6.

6.2.3 Prohledávací heuristiky

dnWalker poskytuje několik způsobů, jak ovlivnit průzkum kódu a jedním z nich je prohledávací strategie, která implementuje uživatelem vybranou heuristiku. V aktuální verzi je implementováno 5 jednoduchých strategií, které využívají heuristiky pro řízení průzkumu. Heuristika je poskytnutá přes implementaci rozhraní `IExplorationStrategy` (viz program 6.5).

```
1 interface IExplorationStrategy {
2   // Invoked before the exploration
3   void Initialize(ExplicitActiveState activeState,
4     MethodDef entryPoint, IConfiguration configuration);
5
6   // Produces next input model.
7   bool TryGetNextSatisfiableNode(ISolver solver,
8     out ConstraintNode node,  out IModel inputModel);
9
10  // Adds new discovered node for the exploration.
11  void AddDiscoveredNode(ConstraintNode node);
12
13  // Adds explored node.
14  void AddExploredNode(ConstraintNode node);
15 }
```

Program 6.5: Rozhraní `IExplorationStrategy`, pomocí kterého je možné přidat do průzkumu heuristiku.

Pomocí metody `Initialize` je možné nastavit chování podle aktuálního průzkumu. Metody `AddDiscoveredNode` a `AddExploredNode` jsou využité k ohlášení objevení nových uzlů respektive prozkoumání uzlů. Metoda `TryGetNextSatisfiableNode` poskytne omezovací uzel a důkaz jeho splnitelnosti (model), který má být využit pro následující iteraci. Pokut žádný takový neexistuje vrací `false` a tím ukončí průzkum.

AllNodesCoverage. Jednoduchá heuristika, která měří pokrytí uzlů CFG a volí pouze ty uzly omezení, které vedou na prohledání ještě nepokrytých uzlů.

AllEdgesCoverage. Jednoduchá heuristika, která měří pokrytí hran CFG a volí pouze ty uzle omezení, které vedou na prohledání ještě nepokrytých hran. Od předchozí strategie se liší v průzkumu všech paralelních hran, které *AllNodesCoverage* bude ignorovat.

AllPathsCoverage. Naivní strategie, která provádí DFS prostoru omezení. Prozkoumá všechny nalezené omezovací uzly. Pokud metoda obsahuje cyklus či rekurzi, jejichž opakování je omezené vstupními argumenty, tak korektně neskončí a v cyklu „uvízne“.

FirstNPathsCoverage. Úprava předchozí strategie, která ale skončí po n iteracích.

SmartPathsCoverage. Komplexnější strategie (viz algoritmus 6.2), která primárně provádí DFS prostoru omezení, ale zároveň kontroluje, kolikrát byla hrana, na kterou vede aktuální uzel stromu omezení, již prozkoumána. Pokud je hodnota vyšší než limit, odloží uzel na později a prozkoumá zbylé uzly. Pokud jí zbývají pouze odložené uzly, tak prochází odložené uzly a zvolí první splnitelný z nich, přičemž navýší limit opakování. Tímto mechanismem se zajistí, že průzkum neuvízne na začátku cyklu programu, jako *AllPathsCoverage*, ale nejdříve zkusí prohledat tělo cyklu, než navýší počet opakování.

Druhý mechanismus spočívá ve filtraci uzlů stromu omezení. Strategie počítá kolik proběhlo iterací od poslední nově objevené hrany a pokud je překročen konfigurovatelný limit (defaultně 10), tak odmítne přijímat uzly, které nové hrany neobjevují. Tímto způsobem korektně ukončí průzkum, pokud „usoudí“ že další průzkum nemá smysl, protože se pouze opakuje jeden či více cyklů.

6.2.4 Symbolická exekuce

Součástí konkolické exekuce je i symbolická exekuce, která souběžně s konkrétní exekucí pracuje se symbolickým stavem, rozšiřuje prostor omezeni a inicializuje konkrétní stav podle vstupní podmínky.

Tvorba symbolických výrazů

Konkrétní exekuce se v *MoonWalkeru* provádí nad základními datovými elementy, které mají unikátní označení a reprezentují jakákoliv data. Pomocí úprav v model checkeru je možné k těmto datovým elementům připojit symbolický výraz či jakákoliv jiná metadata. S těmito výrazy pak pracují rozšíření

```
1:  procedure GetNextNode(s)
2:      while ¬s.Nodes.IsEmpty() do
3:          n ← s.Nodes.Pop()
4:          c ← s.GetCounter(n.Edge)
5:          if c ⩾ s.CMax then
6:              s.Delayed.Push(n)
7:          else
8:              if s.IsSAT(n) then
9:                  s.SetCounter(n.Edge, c + 1)
10:                 return(n, s.GetModel(n))
11:             end if
12:         end if
13:     end while
14:     while ¬s.Delayed.IsEmpty() do
15:         if s.IsSAT(n) then
16:             s.SetCounter(n.Edge, c + 1)
17:             s.CMax ← c + 1
18:             return(n, s.GetModel(n))
19:         end if
20:     end while
21:     return nil
22: end procedure
```

Algoritmus 6.2: Postup strategie *SmartPathsCoverage*, vstup je stav strategie, výstup je omezovací uzel a model dokazující jeho splnitelnost nebo **nil**.

jednotlivých instrukcí. Pro každý datový element (`DE`) a jeho symbolický výraz ($Symbolic(x)$) musí platit následující podmínka:

$$\texttt{DE} = Symbolic(\texttt{DE}), \tag{6.5}$$

Podmínka 6.5 vyjadřuje, že symbolický výraz platí pouze v kontextu daného datového elementu. Uvažujme například CLI kód 6.6, který je spuštěn s konkrétní hodnotou argumentu $x = 1$. Po instrukci `IL_0002` se na zásobníku octne hodnota 0, která je anotována symbolickým výrazem $(x < 0)$, který je sám o sobě nepravdivý $((x = 1) > 0)$. Výraz, který je sám o sobě pravdivý se objeví až po exekuci instrukce `IL_0004`, na kterém instrukce `brfalse` interpretuje připojený výraz $(x < 0)$ vůči jeho datovému elementu (0, respektive **false**) a vytvoří korektní omezení cesty $((x < 0) = \textbf{false}) = (x \geqslant 0)$.

Každá instrukce má pomocí rozšíření základního exekutoru instrukce definovanou změnu symbolického stavu. Instrukce lze rozdělit do 4 skupin podle způsobů úpravy symbolického stavu. V první řadě instrukce nemusí dělat nic. Jedná se o instrukce, jejichž provedení nemá vliv na symbolický stav programu (například `noop`, `br`).

Druhá kategorie instrukcí pracuje s načtením dat z paměti na zásobník, či uložení dat ze zásobníku do paměti. Načítání datových elementů pracuje ve dvou režimech: daný datový element ještě nikdy nebyl načten, nebo daný

```
.method public hidebysig static void SingleBranching (
    int32 x) cil managed {
  .maxstack 2

  IL_0000: ldarg.0       // loads arg x (concrete 1) on the stack
  IL_0001: ldc.i4.0      // loads constant 0 on the stack
  IL_0002: clt           // loads 0, (1 > 0), symbolic expr x < 0
  IL_0004: brfalse.s ... // adds path constraint !(x < 0)

  ...
}
```

Program 6.6: Větvení v CLI se symbolickými hodnotami. Průběh pro vstup $x = 1$.

datový element již byl načten. V prvním případě se data odloženě inicializují (*lazy initialization*) podle aktuálního symbolického modelu (viz 6.2.4). V druhém případě není třeba dělat nic. Při ukládání dat do paměti se musí aktualizovat symbolický model. Jedná se například o `ldarg` (načte argument metody), `ldfld` (načte hodnotu fieldu objektu) nebo `stelem` (uloží hodnotu jako element pole).

Instrukce z třetí kategorie provádí nějakou operaci nad daty na evaluačním zásobníku. Symbolická exekuce vytvoří nový výraz, nad výrazy operandů, který reprezentuje danou operaci a napojí jej na výsledek. Nedělá nic, pokud žádný z operandů nemá symbolickou reprezentaci (není závislý na vstupu). Jedná se například o `ceq` (vloží na zásobník 1, pokud jsou si operandy rovny, jinak 0) nebo `add` (vloží na zásobník součet operandů).

Poslední kategorie instrukcí upravuje tok řízení. Na základě symbolických výrazů operandů vytvoří možnosti, které mohou nastat a společně s volbou, provedenou aktuální konkrétní exekucí, předají informaci `ConstraintTreeExplorer`, pomocí jeho metody `MakeDecision`. Jedná se například o instrukci `brfalse`. V programu 6.6 se na řádku 4 vytvoří 2 možnosti: $(x < 0)$ a $(x \geqslant 0)$. V průběhu pro $x = 1$ zvolí konkrétní exekuce druhou variantu, protože $1 \geqslant 0$ a v některé z příštích iterací by se měla provést podmínka $x < 0$.

Instrukce ale nemusí spadat do pouze jedné kategorie. Jedná se například o instrukci `ldelem` (na zásobník vloží z pole položku na daném indexu). V tomto případě totiž mohou nastat 3 situace: reference na pole je `null`, index je mimo rozsah (menší než 0 nebo větší nebo roven délce pole) a operandy jsou bez problémů. V prvním, respektive druhém případě se vyhodí výjimka `NullReferenceException` respektive `IndexOutOfRangeException` a ve třetím případě proběhne odložená inicializace elementu pole. Proto instrukce `ldelem` patří do druhé i čtvrté skupiny instrukcí.

Název	Značení	Význam
Členské proměnné		
`ArrayElement`	`[index]`	Element v poli pod daným indexem
`ArrayLength`	*Length*	Délka pole
`InstanceField`	`field`	Hodnota fieldu objektu
`MethodResult`	`method`$(i : n)$	Návratová hodnota dané metody při n-tém zavolání
`ConstrainedMethodResult`	`method`(p)	Návratová hodnota dané metody, argumenty splňují predikát p
Kořenové proměnné		
`MethodArgument`	`argname`	Argument zkoumané metody
`ReturnValue`		Návratová hodnota zkoumané metody
`StaticFieldValue`	`Class.field`	Hodnota statického fieldu třídy

Tabulka 6.1: Typy proměnných v symbolickém modelu.

Proměnné

Symbolické výrazy jsou operacemi nad proměnnými a konstantami. Konstanty jsou čísla, logické hodnoty, konkrétní řetězce a **nil**. Proměnné reprezentují část paměti modelované na základě separační logiky. Existují 2 základní typy proměnných - *kořenové*(*root*) a *členské*(*member*).

Kořenové proměnné reprezentují oblasti v paměti, které jsou přímo dosažitelné při exekuci metody. Jedná se mimo jiné o argumenty metody a statické fieldy tříd. Členské proměnné reprezentují oblasti v paměti relativně vůči jiné proměnné - kořenové nebo členské. Jedná se mimo jiné o field objektu, element pole nebo návratovou hodnotu metody. Seznam všech aktuálně podporovaných proměnných je v tabulce 6.1.

V ukázce kódu v programu 6.7 jsou v metodě `Foo` k dispozici 3 kořenové proměnné: statický field `Bar.MagicNumber`, implicitní argument metody `this` a normální argument metody `baz`. Členské proměnné jsou použité 2: field objektu `baz.X` a návratová hodnota metody `this.GetMagicNumber2()`.

Obsah haldy lze pak reprezentovat jako les, kde kořenové proměnné jsou kořeny a členské zbylými uzly jednotlivých stromů. Reálně ale dochází k aliasingu , takže některé vrcholy splývají a mohou vznikat i cykly. Při řešení splnitelnosti formulí obsahující členské proměnné je proto nutné aplikovat omezení vycházející ze separační logiky, protože členské proměnné implikují tvrzení *ukazuje na*. Například proměnná zmíněná výše `baz.X` implikuje

```
 1 public abstract class Bar {
 2   private static int MagicNumber;
 3
 4   protected abstract int GetMagicNumber2();
 5
 6   public void Foo(Baz baz) {
 7     if (baz.X == Bar.MagicNumber ||
 8         baz.X == this.GetMagicNumber2())
 9         ...
10   }
11 }
```

Program 6.7: Příklad různých proměnných

omezení $baz \mapsto (X : \dots)$ (viz část 6.2.5).

Odložená inicializace (LI) a symbolický model

Při konkolickém průzkumu se používá varianta odležené inicializace (*lazy initialization* LI). Exekuce model checkeru se spustí s defaultními, neinicializovanými vstupy - prázdná dynamická oblast, defaultní hodnoty datových elementů. Pokud exekuce potřebuje načíst hodnotu na zásobník, provede symbolická exekuce její inicializaci.

Výchozími daty pro inicializaci je vstupní symbolický model, tedy výstup ze solveru. Každou oblast paměti, ze které se mají data načíst, lze reprezentovat jako proměnnou (viz sekce 6.2.4). Pokud má vstupní model pro danou proměnnou definovanou hodnotu, tak se podle ní inicializují konkrétní hodnoty v rámci model checkeru. Pokud žádná hodnota definována není (proměnná se neúčastní vstupních podmínek), tak se žádná inicializace nekoná a ponechá se defaultní hodnota, tedy 0 či **false** v případě numerických respektive logických typů a `null` v případě referenčních typů.

Při inicializaci se provede inicializace pouze první vrstvy dat, například pokud se má inicializovat objekt, tak se alokuje v dynamické oblasti, ale neinicializují se mu hodnoty fieldů, třebaže by v symbolickém modelu měly definované hodnoty. Ty se nastaví až v případě, pokud by to bylo potřeba, tedy až se nad právě inicializovaným objektem provede instrukce `ldfld` (načíst hodnotu fieldu).

Symbolická exekuce udržuje dva symbolické modely. První popisuje vstupní podmínky a je neměnný. Na druhém, který je na začátku exekuce kopií

vstupního modelu, se ale sledují změny a na konci exekuce tak popisuje výstupní stav. Například při exekuci instrukce `stfld` se nastaví hodnota dané proměnné ve výstupním modelu. Ten se také používá pro odloženou inicializaci.

Chování metod

Pokud *dnWalker* narazí při exekuci na neimplementovanou metodu (metoda rozhraní nebo abstraktní metoda) tak její zavolání nahradí vložením výsledku přímo na zásobník a s výsledkem pracuje jako se symbolickou proměnnou. Při určení výsledku pracuje ve dvou módech: počítání volání a podmíněné volání.

Počítání volání. Defaultní chování počítá pokolikáté se metoda na dané instanci volá a každé zavolání vyhodnotí jako nezávislou hodnotu. Při tomto chování se ignorují hodnoty argumentů a instance se pak chová jako *Test Stub*, který postupně vrací sekvenci hodnot.

Toto chování lze nastavit pomocí vstupního modelu (viz část 6.4) a zároveň se generuje při exekuci – při opakovaném volání se vytváří se tak nové nezávislé proměnné, které jsou pak součástí vstupního modelu následujících iterací.

Podmíněné volání. Umožňuje uživateli nastavit jednoduché chování metody pomocí predikátu nad vstupními argumenty. Uvažujme například metodu `int Foo(int x)`. Pomocí predikátů $x < 0 \implies 5$, $0 \leqslant x < 5 \implies 3$ a $x \geqslant 5 \implies 1$ se definuje chování, která pro dané hodnoty x vrací čísla 5, 3 nebo 1. Každý predikát tak představuje jednu třídu ekvivalence. Celá skupina predikátů P proto musí splňovat následující vlastnosti:

1. $\bigvee_{p \in P}$ je platná formule – pro jakékoliv ohodnocení je alespoň jeden predikát p splněný,
2. $\forall p_1, p_2 \in P \; p_1 \wedge p_2$ je nesplnitelná formule – žádné dva predikáty p_1 a p_2 nemohou být splněné současně.

Pokud tyto vlastnosti nejsou zajištěné, tak v prvním respektive druhém případě *dnWalker* skončí s chybou, respektive mockované chování není deterministické a při konkolickém průzkumu může probíhat jinak, než při spuštění vygenerovaných testů.

Podmíněné výsledky je v aktuální verzi možné nastavit pouze pomocí vstupního modelu a predikáty lze definovat pouze nad proměnnými primitivních typů, kvůli omezenému parseru symbolických výrazů. Pro tuto funkcionalitu se také nabízí možnost využít neinterpretované funkce v SMT teoriích, což je předmětem dalšího vývoje.

dnWalker pracuje s podmíněným voláním jako s rozhodovacím bodem a aktivně nastavuje vstupní podmínku tak, aby prozkoumal všechny varianty daného chování.

Příklad

Průběh konkolické exekuce je možné ukázat na následujícím příkladu metody `ProcessNext` ve třídě `Processor`. Kód v C# je zobrazen v programu 6.8 a jeho CLI v programu 6.9. Metoda `ProcessNext` zavolá metodu `IRunner::Run`, přičemž předá jako parametr aktuální hodnotu fieldu `_chunkId`. Vrací výsledek zavolané metody `Run` a zvýší hodnotu `_chunkId`.

Přestože metoda neobsahuje struktury kontroly řízení obsahuje větvící bod a to na instrukci `IL_0012: callvirt`. Pokud je argument `null`, tak se vyhodí výjimka `NullReferenceException`, v opačném případě se vyřeší volání metody (v tomto případě pomocí počítání volání) a pokračuje se další instrukcí. Prohledávání této metody se proto provede ve dvou iteracích.

```
 1 public class Processor {
 2   int _chunkId = 0;
 3
 4   public int ProcessNext(IRunner r) {
 5     return r.Run(_chunkId++);
 6   }
 7 }
 8
 9 public interface IRunner {
10     int Run(int chunkId);
11 }
```

Program 6.8: Metoda `Processor::ProcessNext`, C#. Pro ukázku průběhu konkolické exekuce.

Chování exekuce při prvním průchodu (vstupní podmínka je $this \neq$ **nil**):

`IL_0000` LI na `null`, žádná omezení nad argumentem `r`.

`IL_0001` LI na prázdnou instanci, $this \neq$ **nil**.

`IL_0002` Žádná LI, datový element je již inicializovaný. Zásobník obsahuje dvě reference na `this` a poté `null`
`IL_0003` LI na `0`, žádná omezení nad $this.\_chunkId$.
`IL_0008` Do lokální proměnné `[0]` se vloží 0
`IL_0009` Z lokální proměnné `[0]` se na zásobník načte 0
`IL_000a` Na zásobník se načte konstanta 1
`IL_000b` Na zásobník se vloží $0 + 1 = 1$ a k výslednému datovému elementu se napojí výraz $this.\_chunkId + 1$
`IL_000c` Aktualizace výstupního modelu: $this.\_chunkId := 1$ a aktualizace konkrétního stavu.
`IL_0011` Z lokální proměnné `[0]` se na zásobník načte 0
`IL_0012` Vyhodí se výjimka `NullReferenceException` a pomocí metody `MakeDecision` se předají omezení $r =$ **nil** a $r \neq$ **nil**, přičemž bylo zvoleno první z nich.
`IL_0017` Neuskuteční se, protože předchozí instrukce vyhodí výjimku.

Chování exekuce při druhém průchodu (vstupní podmínka je $this \neq$ **nil** $\wedge\ r \neq$ **nil**):
`IL_0000` LI na prázdnou instanci, $r \neq$ **nil**.
`IL_0001` LI na prázdnou instanci, $this \neq$ **nil**.
`IL_0002` Žádná LI, datový element je již inicializovaný. Zásobník obsahuje dvě reference na `this` a referenci na `r`
`IL_0003` LI na `0`, žádná omezení nad $this.\_chunkId$
`IL_0008` Do lokální proměnné `[0]` se vloží 0
`IL_0009` Z lokální proměnné `[0]` se na zásobník načte 0
`IL_000a` Na zásobník se načte konstanta 1
`IL_000b` Na výsledek $(0 + 1 = 1)$ se napojí výraz $this.\_chunkId + 1$
`IL_000c` Aktualizace výstupního modelu, $this.\_chunkId := 1$
`IL_0011` Z lokální proměnné `[0]` se na zásobník načte 0
`IL_0012` Na zásobník vloží 0 s výrazem $r.Run(i : 1)$ (počítání volání neimplementované metody) a pomocí metody `MakeDecision` se předají omezení $r =$ **nil** a $r \neq$ **nil**, přičemž bylo zvoleno druhé z nich.
`IL_0017` Aktualizace výstupního modelu, návratová hodnota je 0 s výrazem $r.Run(i : 1)$ a stejná hodnota se vloží na zásobník volající metody.

6.2.5 Solver

Jedním ze základů pro konkolickou exekuci je nástroj pro rozhodnutí splnitelnosti logických formulí – SMT solver. Pro *dnWalker* byl zvolen solver *Z3*[30]. Uvažovalo se i o použití solveru *CVC5*[7], jehož hlavní výhodou oproti *Z3* je

```
.method public hidebysig  instance int32 RunNext (
    class IRunner r) cil managed {
  .maxstack 4
  .locals init ([0] int32)

  IL_0000: ldarg.1
  IL_0001: ldarg.0
  IL_0002: ldarg.0
  IL_0003: ldfld int32 Processor::_chunkId
  IL_0008: stloc.0
  IL_0009: ldloc.0
  IL_000a: ldc.i4.1
  IL_000b: add
  IL_000c: stfld int32 Processor::_chunkId
  IL_0011: ldloc.0
  IL_0012: callvirt instance int32 IRunner::Run(int32)
  IL_0017: ret
}
```

Program 6.9: Metoda `Processor::ProcessNext`, CLI. Pro ukázku průběhu konkolické exekuce.

integrace separační logiky. Oproti *Z3* ale neposkytuje rozhraní pro .NET[5]. Protože přinejmenším pro účely aktuální verze stačí níže popsaná popsaná procedura překladu formule separační logiky na SMT formuli, jejíž implementace je podstatně jednodušší než integrovat *CVC5* do .NET prostředí, nebyl vybrán *CVC5*.

Logické výrazy

Pro vyjádření symbolických výrazů je implementována samostatná knihovna *dnWalker.Symbolic*. Kromě tvorby výrazů, které jsou reprezentovány jako abstraktní syntaktické stromy, poskytuje knihovna třídy pro reprezentaci modelu separační logiky, podle formulace 4.17.

Vlastní implementace symbolických výrazů, namísto využití existujících řešení, jako například tříd z `System.Linq.Expressions`, byla zvolena za účelem vyjádření výrazů separační logiky a možnost snazší integrace s použitými knihovnami, zejména pak *dnlib*.

Třídy v knihovně *dnWalker.Symbolic* nejsou závislé na konkrétním SMT

[5] `https://github.com/cvc5/cvc5/issues/6997`

Formule	Φ	::=	$\Delta \mid \Phi_1 \vee \Phi_2$
Symbolická halda	Δ	::=	$\exists \bar{v}\ (\kappa \wedge \alpha \wedge \phi)$
Prostorová formule	κ	::=	$\mathbf{emp} \mid x \mapsto c(f_i : v_i) \mid \kappa_1 \star \kappa_2$
(Ne)rovnost ukazatelů	α	::=	$\mathbf{true} \mid \mathbf{false} \mid v_1 = v_2 \mid v = \mathbf{nil} \mid$ $v_1 \neq v_2 \mid v \neq \mathbf{nil} \mid \alpha_1 \wedge \alpha_2$
Pressburgerova aritmetika	ϕ	::=	$\mathbf{true} \mid i \mid \exists v\ \phi \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$
Lineární aritmetika	i	::=	$a_1 = a_2 \mid a_1 \leqslant a_2$
Lineární výraz	a	::=	$k \times a \mid a_1 + a_2 \mid -a \mid k \mid v \mid$ $\max(a_1, a_2) \mid \min(a_1, a_2)$

Tabulka 6.2: Syntax použitého fragmentu separační logiky, zjednodušení fragmentu z [58].

solveru a představují tak abstraktní rozhraní mezi *dnWalkerem* a SMT solverem.

Rozhodovatelnost separační logiky

Jak bylo řečeno dříve, použitý solver *Z3* nepodporuje výrazy separační logiky. Pro její rozhodnutí je proto implementovaná jednoduchá procedura, která přeloží formuli v separační logice na ekvivalentně splnitelnou SMT formuli. Procedura je založená na práci [58] a přeneseně i [20].

V [58] je definován rozhodovatelný fragment separační logiky, který navíc podporuje indukční predikáty a Pressburgerovu aritmetiku pro neprostorová omezení proměnných. Třebaže indukční predikáty mají velký potenciál pro nástroje jako *dnWalker*, tak jsou předmětem dalšího vývoje (viz 8.2) a v aktuální verzi *dnWalkeru* je použit fragment popsaný v tabulce 6.2. Ten odpovídá fragmentu z [58] s výjimkou využití indukčních predikátů.

V rámci použité formulace separační logiky se pracuje s abstraktními atributy datových buněk. V aktuální verzi se pracuje s 6 typy atributů, které odpovídají typům členských proměnných představených v tabulce 6.1 rozšířené o 6. atribut – *typ*. V aktuální verzi se nepracuje s omezeními nad typy, nicméně jedná se o důležité rozšíření *dnWalkeru*, viz část 8.2.

Překlad na SMT formuli

Pro určení splnitelnosti formule separační logiky odpovídající fragmentu 6.2 a získání modelu (důkazu splnitelnosti) je nutné ji přeložit na ekvivalentně splnitelnou SMT formuli a v případě splnitelné formule i přeložit SMT model (mapování neinterpretovaných funkcí na konstanty) na model separační logiky (objekty $s \in Stores, h \in Heaps$).

```
 1 class Segment {
 2     int[] data;
 3     Segment next;
 4     ...
 5
 6     public void Foo(Segment other, int x) {
 7         if (x == 5 && data != null) {
 8             ...
 9         }
10         ...
11     }
12 }
```

Program 6.10: Definice třídy `Segment` použité v příkladě překladu formule separační logiky.

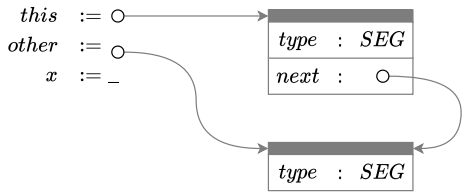
V následujícím textu je tento proces popsán a představen na příkladu podmínky z programu 6.10, která musí být splněna pro dosažení řádku 8 a je rozšířena o vstupní podmínku (druhý řádek formule, zobrazená na obrázku 6.8):

$$x = 5 \wedge this.data \neq \mathbf{nil} \wedge \\ this \mapsto (type : SEG, next : other) * other \mapsto (type : SEG) \quad (6.6)$$

kde x je proměnná typu `int` a typ proměnných *this*, *this.next* a *other* je SEG (třída `Segment` z programu 6.10). Proměnná *obj.data* je pole `int`. Proměnné *this*, *other* a x odpovídají argumentům metody `Foo` a jsou proto kořenové proměnné, *this.data* je členská proměnná.

Poznámka k terminologii. Separační logika pracuje s pojmy *ukazatel* a *ukazuje* (*points to*). V doméně CLI je pro obdobné chování používaný pojem *(spravovaná) reference* (*managed reference*) respektive *odkazuje*[6].V CLI se

[6]CLI pracuje s pojmem *ukazatel*, který může být spravovaný (*managed*) a poté ukazuje na spravovanou část paměti (lokální proměnná, field objektu atp) nebo nespravovaný, který může existovat pouze v tzv. *unsafe* kontextu a v takovém případě odpovídá „standardnímu“ ukazateli, na který lze aplikovat stejné operace s jakými se pracuje v nízkoúrovňových jazycích. Při použití nespravovaných ukazatelů je také zakázán *garbase collector* (GC), aby se zajistila jeho bezpečnost. V rámci této práce ale není tato vlastnost podporována.



Obrázek 6.8: Schéma vstupní podmínky pro příklad transformace formule separační logiky na výraz rozhodovatelný SMT solverem.

také rozlišují pojmy *rovnost* a *identita*. Zatímco (ne)rovnost odpovídá doméně datového typu a funguje stejně pro hodnotové a referenční typy (jedná se o výsledek operátoru `==` respektive `!=` či metody `Equals`, které lze přetížit), identita má smysl pouze pro referenční typy a je založená na totožnosti odkazovaných objektů (jejich reálnému umístění v paměti). Odpovídá tak rovnosti lokací v rámci separační logiky.

V následujícím textu je použitá terminologie separační logiky. Pro referenční typy odpovídá termínu *rovnost identita* a původní význam rovnosti z CLI se neuvažuje[7].

Mapování proměnných a konstant. Ke každé proměnné a konstantě, která se vyskytují v rozhodované formuli, je třeba vytvořit jejich ekvivalenci ve výsledné SMT formuli, což je neinterpretovaná funkce nulové arity.

Pro konstanty a proměnné primitivních typů je překlad přímočarý, protože pro ně existují zavedené teorie a druhy. Problém ale nastává pro proměnné referenčních typů. Podle použité formulace separační logiky je množina všech lokací (potenciálních hodnot proměnných a konstant referenčních typů) disjunktní s množinou čísel a jiných skalárních hodnot (potenciální hodnoty proměnných a konstant primitivních typů). Proto je pro jejich reprezentaci vytvořený nový neinterpretovaný druh jménem *loc*. Pro reprezentaci konstanty **nil** (`null`) je také vytvořena speciální konstanta druhu *loc*, pojmenovaná *nil*.

Proměnné a konstanty z formule 6.6 jsou na neinterpretované funkce ma-

[7]Tuto úvahu je možné provést, protože v praxi při práci s rovností dvou objektů se konkolicky prohledá přetížení jedné z výše zmíněných metod a návratová hodnota pak ponese vlastní výraz vybudovaný v rámci exekuce dané metody, který se bude skládat z rovnosti primitivních hodnot a identity referencí.

povány podle rovnice 6.7

$$\begin{array}{rclrcl} x & \equiv & (int_x) & this.data & \equiv & (loc_{this.data}) \\ \textbf{nil} & \equiv & (loc_{nil}) & this.next & \equiv & (loc_{this.next}) \\ this & \equiv & (loc_{this}) & other & \equiv & (loc_{other}) \end{array} \tag{6.7}$$

kde název dané funkce odpovídá jejímu *druhu* a názvu původní proměnné. Toto pojmenování je zde zvolené pro čitelnost, nicméně pro ušetření paměti jsou v implementaci jména generována z arbitrárně určeného pořadí proměnných, tedy například $!k0!$ pro první přeloženou proměnnou.

Mapování primitivních konstant (v tomto případě pouze čísla 5) není zobrazené, protože se jedná o triviální operaci.

Odebrání separační konjunkce a tvrzení ukazuje. Formuli lze podle fragmentu 6.2 rozdělit na konjunkci tří formulí: prostorové (κ), (ne)rovnosti ukazatelů (α) a Pressburgerovy aritmetiky (ϕ). Toho rozdělení ilustrují rovnice 6.8. Formule α a ϕ také spojíme do tzv. *čisté*(*pure*) formule π, která neobsahuje separační konjunkci a tvrzení ukazuje na.

$$\begin{array}{rcl} \kappa & = & this \mapsto (type : SEG, next : other) * other \mapsto (type : SEG) \\ \alpha & = & this.data \neq \textbf{nil} \\ \phi & = & x = 5 \\ \pi & = & (\alpha \wedge \phi) = (this.data \neq \textbf{nil} \wedge x = 5) \end{array} \tag{6.8}$$

Formule π se přeloží snadno, protože syntax Pressburgerovy aritmetiky je podporován SMT solverem a výrazy (ne)rovnosti ukazatelů lze interpretovat pomocí teorie rovnosti.

Problém nastává až při zpracování formule κ. Zde je použitá rozšířená procedura **eXPure** (rovnice 6.9), popsána v [58], která koresponduje proceduře představené v [20]. Rozšíření oproti [58] spočívá ve třetím členu, který rozkládá tvrzení *ukazuje na* a nahrazuje je konjunkcí rovnosti členských proměnných, které vytvoří podle použitých atributů. První člen zajistí, že proměnná, která ukazuje na nějakou buňku není **nil** a druhý člen vylučuje *pointer aliasing* struktur, které leží na disjunktních částech haldy.

$$\begin{aligned}\textbf{eXPure2}(\exists \bar{w}\ x_1 \mapsto (\bar{a}_1 : \bar{e}_1) \ast \cdots \ast x_n \mapsto (\bar{a}_n : \bar{e}_n) \wedge \pi) \equiv \\ \exists \bar{w} \quad \left(\bigwedge_{i \in \{1 \ldots n\}} x_i \neq \textbf{nil} \right) \\ \wedge \quad \left(\bigwedge_{i,j \in \{1 \ldots n\}, i \neq j} x_i \neq x_j \right) \\ \wedge \quad \left(\bigwedge_{i \in \{1 \ldots n\}} \bigwedge_{j \in \{1 \ldots |\bar{a}_i|\}} x_i.a_j = e_j \right) \wedge \pi\end{aligned} \tag{6.9}$$

Po aplikaci procedury **eXPure2** získáme formuli v logice podporované SMT solverem

$$\begin{aligned}int_x = 5 \wedge loc_{this.data} \neq loc_{nil} \\ \wedge\ loc_{this} \neq loc_{nil} \wedge loc_{other} \neq loc_{nil} \\ \wedge\ loc_{this} \neq loc_{other} \wedge loc_{this.next} = loc_{other}\end{aligned} \tag{6.10}$$

Pointer aliasing. Zavedení členských proměnných (viz 6.2.4) sice vede ke zjednodušení symbolické exekuce, ale může poté vzniknout následující formule, která musí být nesplnitelná:

$$x.a = 5 \wedge y.a = 6 \wedge x = y \tag{6.11}$$

Pro vyřešení potenciálního aliasingu ukazatelů je přidána následující podmínka:

$$x_1 = x_2 \implies \left(\bigwedge_{x_1.a,\ x_2.a \in \mathbf{FV}(\Phi)} x_1.a = x_2.a \right) \tag{6.12}$$

kde $\mathbf{FV}(\Phi)$ je množina všech volných proměnných formule Φ. Pro každou členskou proměnnou proměnných x_1 a x_2, které reprezentují stejný atribut, je zaručena rovnost. Překládaná formule je prozkoumána a pokud se v ní vyskytuje rovnost či nerovnost referenčních proměnných, tak se aplikuje výše popsané omezení. Je nutné zahrnout i nerovnost, protože výraz může být výše znegovaný ($\neg(x \neq y)$). Zároveň je možné, že k rovnosti nikdy dojít nemůže (kvůli dalším částem formule), proto je použitá implikace.

Kvůli následujícímu kroku (zajištění typové bezpečnosti) je nutné tento krok aplikovat pouze na proměnné stejného typu. Pro proměnné jiného typu je bezpředmětný, protože předpoklad nebude nikdy splněný. Zároveň díky němu můžeme zanedbat rovnost typů, jediný atribut, který v aktuální verzi nelze vyjádřit pomocí členské proměnné.

V našem příkladě formuli 6.10 nemusíme rozšiřovat, protože jediný výraz rovnosti proměnných je $loc_{this.next} = loc_{other}$ a pro tyto proměnné se ve formuli nevyskytují žádné členské proměnné, které by proto bylo nutné omezit.

Typová bezpečnost. Posledním krokem je zajištění typové bezpečnosti. Všechny proměnné referenčních typů jsou reprezentovány neinterpretovaných funkcemi nulové arity jednoho druhu, tudíž je možné, aby se 2 referenční proměnné různých typů rovnaly. Proto se pro každé 2 referenční proměnné různých typů zajistí následující tvrzení:

$$x_1 \neq x_2 \vee (x_1 = \mathbf{nil} \wedge x_2 = \mathbf{nil}) \tag{6.13}$$

respektive, dvě reference různých typů se mohou rovnat, pouze tehdy, pokud jsou obě dvě `null`.

Tyto výrazy se přidají pro 3 dvojice a to (loc_{this}, $loc_{this.data}$), ($loc_{this.next}$, $loc_{this.data}$) a (loc_{other}, $loc_{this.data}$). Všechny ostatní dvojice jsou stejného typu (`Segment`) a proto se z hlediska typové bezpečnosti mohou rovnat.

Výsledná formule, která je vstupem do SMT solveru je tedy:

$$\begin{aligned} int_x = 5 \wedge loc_{this.data} \neq loc_{nil} \wedge loc_{this} \neq loc_{nil} \wedge loc_{other} \neq loc_{nil} \wedge \\ loc_{this} \neq loc_{other} \wedge loc_{this.next} = loc_{other} \wedge \\ (loc_{this} \neq loc_{this.data} \vee (loc_{this} = loc_{nil} \wedge loc_{this.data} = loc_{nil})) \wedge \\ (loc_{this.next} \neq loc_{this.data} \vee (loc_{this.next} = loc_{nil} \wedge loc_{this.data} = loc_{nil})) \wedge \\ (loc_{other} \neq loc_{this.data} \vee (loc_{other} = loc_{nil} \wedge loc_{this.data} = loc_{nil})) \end{aligned} \tag{6.14}$$

Vytvoření modelu separační logiky

Výstup z SMT solveru rozhoduje, zda se jedná o splnitelnou či nesplnitelnou formuli (případně že nelze rozhodnout) a případně poskytuje důkaz splnitelnosti, respektive ohodnocení proměnných konkrétními hodnotami (model). Budování modelu separační logiky z tohoto modelu je přímočaré.

Pracuje se s proměnnými původní formule, pro které známe mapování na proměnné SMT formule. Tyto proměnné se seřadí do pořadí dle hloubky, nejprve tedy kořenové proměnné, poté první členové, následně druhé atd. Shodou okolností se jedná o stejné pořadí jako v rovnici 6.7. V tomto pořadí se poté proměnné postupně inicializují podle hodnot z SMT modelu. Díky

seřazení dle hloubky je zajištěno, že proměnná výše ve stromu proměnných bude inicializována předtím, než se začne inicializovat její potomek. V našem případě to znamená, že když je potřeba inicializovat proměnnou *this.data*, tak v objektu *Heap* již existuje datová buňka, na kterou ukazuje proměnná *this*.

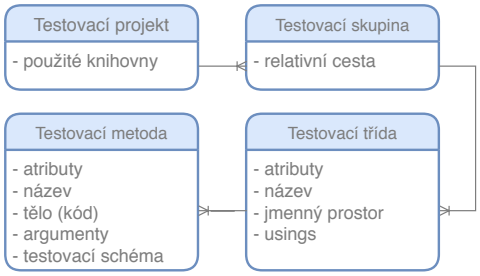
Inicializace proměnných primitivního typu je opět přímočará, protože je možné snadno převést SMT primitivní konstanty na vyjádření konstant použité symbolickou exekucí. Při převádění proměnných typu reference dochází k mapování hodnot neinterpretovaného druhu (tedy definic funkcí s nulovou aritou) na hodnoty typu `Location`. Přičemž při prvním nalezení se v modelu separační logiky alokuje datová buňka s vhodným typem.

Výše popsaná procedura pro řešení omezení vyjádřených pomocí separační logiky má mnohé nedostatky. V první řadě se jedná o naivní implementaci, kterou by bylo určitě možné zefektivnit, aby nevytvářela příliš velké množství redundantních omezení mezi referencemi. Zároveň nepodporuje práci s indukčními predikáty, což je velmi silný nástroj separační logiky. Nejzásadnějším problémem je ale chybějící podpora polymorfismu, dědění a podpory omezení nad typy obecně, což patří mezi nutné další kroky (viz část 8.2).

V popise byla vynechána práci s řetězci. Jedná se o jakýsi hybrid mezi referenčními a primitivními hodnotami. Jedná se o reference, protože mohou být rovny `null`, ale zároveň je vhodné s nimi pracovat jako s primitivními hodnotami, protože jsou neměnné a jejich rovnost se odvíjí od hodnoty a ne identity. Navíc mnoho SMT solverů implementuje teorii řetězců a proto je možné omezení nad nimi vyjadřovat pomocí výrazů z této teorie. Ta nicméně nepočítá s hodnotou **nil**, proto je použit podobný trik jako při práci s **nil** u referenčních hodnot a to vytvoření konstanty druhu *string* s názvem *string-null*.

6.3 Generátor testů

Generátor testů má za úkol zpracovat výstup z konkolické exekuce, tedy data obsahující informace o jednotlivých iteracích exekuce. Výstupem je zdrojový kód jednotkových testů, uspořádaný v testovacím projektu. Při návrhu generátoru testů byl kladen velký důraz na rozšiřitelnost systému a nezávislost na konkrétní knihovně či testovacím frameworku. To umožňuje uživateli použít libovolnou kombinaci nástrojů, pakliže pro ně poskytne vhodný modul *dnWalkeru*, ale zároveň to vyžaduje od uživatele korektní nastavení



Obrázek 6.9: Struktura souboru testů.

všech použitých komponent.

V první fázi generování dochází k analýze dat z průzkumu a pro každou iteraci průzkumu jsou vybrána *testová schémata*. Testové schéma poté pomocí *testové šablony* vygeneruje tělo jednotlivých testových metod. Ty jsou poté uspořádány do testových tříd, testových skupin a testového projektu, který tak tvoří výsledný *soubor testů* (*test suite*), viz obrázek 6.9.

6.3.1 Testové schéma

Při generování testů je kladen důraz na čitelnost. Každý test proto testuje pouze jednu událost, která pro dané vstupní parametry nastane. Tím se předchází tzv. *ruletě ověření*(*assertion roulette*), která nastává, pokud je obtížné určit, proč test nebyl úspěšný[66]. Pro jednu kombinaci vstupů (iterace konkolické exekuce) proto může vzniknout i více než jeden test. Testové schéma obsahuje informace o vstupních a výstupních datech a umí pomocí testové šablony (viz sekce 6.3.2), vygenerovat zdrojový kód testu.

Ve výchozím stavu jsou poskytnuta 4 testová schémata, viz tabulka 6.3. Dodání nových schémat je umožněno přes rozhraní `ITestSchemaProvider` a `ITestSchema`. Implementace `ITestSchemaProvider` musí z dodané iterace konkolického průzkumu vytvořit množinu objektů implementující `ITestSchema`. Toto rozhraní pak obsahuje metodu `Write`, která pomocí dodané testové šablony vygeneruje zdrojový kód metody.

Název	Popis
`Exceptions`	Testuje (ne)vyhození výjimky.
`ChangedArray`	Testuje mutaci pole, které bylo součástí vstupu metody.
`ChangedObject`	Testuje mutaci objektu, který byl součástí vstupu metody.
`ReturnValue`	Testuje hodnotu či identitu návratové hodnoty metody.

Tabulka 6.3: Základní schémata testu.

Název	Popis
`CreateInstance`	Vytvořit novou instanci.
`InitializeField`	Nastavit field dané instanci.
`InitializeStaticField`	Nastavit statický field dané třídy.
`InitializeMethod`	Nastavit výstupy N volání metody dané instanci.
`InitializeStaticMethod`	Nastavit výstupy N volání statické metody dané třídy.
`InitializeArrayElement`	Nastavit element daného pole.

Tabulka 6.4: Testová primitiva - uspořádání.

6.3.2 Testová šablona

Generování jednotkových testů je řešené přes testová primitiva (viz tabulky 6.4, 6.5 a 6.6). Každé primitivum představuje jeden základní blok testovacího kódu. Testová šablona je agregát služeb, které testovací primitivy implementují.

Integrace primitivů je možná pomocí implementace rozhraní `IArrange`/`IAct`/`IAssertPrimitives`, které poskytují primitiva z dané skupiny. Ne všechna primitiva ale musí být poskytnutá v rámci jedné implementace. Například implementace `Arrange` primitive pomocí již zmíněné knihovny *moq*, která nabízí izolační framework, nebude poskytovat primitiva pro uspořádání soukromých fieldů objektu.

Testová šablona udržuje zásobník všech implementací a při žádosti o dané primitivum je postupně zkouší. Pokud žádná z implementací dané primitivum neposkytuje, tak proces končí s chybou.

Ve výchozím stavu jsou poskytnuté implementace všech primitiv s výjimkou `InitializeMethod`, `InitializeStaticMethod` a (`Not`)`Equivalent`, za pomoci reflexe a třídy `Debug`. V rámci této práce jsou také implementovaná primitiva pro uspořádání pomocí izolačního frameworku *moq*[110] a v rámci implementace testovacího frameworku *xUnit.Net*[116] i ověřovací primitivy pomocí třídy `Xunit.Assert`. Očekává se, že uživatel poskytne implementaci všech primitivů, které jsou potřeba pro generování testů z jím nastavené

Název	Popis
`Act`	Spustí testovanou metodu.
`ActDelegate`	Vytvoří delegáta, který spustí testovanou metodu.

Tabulka 6.5: Testová primitiva - vykonání.

konfigurace.

6.3.3 Uspořádání testovacích dat

Jednou z výzev napsání testu je správná inicializace testových dat (fáze uspořádání, *arrange*). Pro primitivní hodnoty se nejedná o náročnou operaci, protože stačí vypsat jejich hodnot pomocí výrazu. Problém přestavují dynamická data na haldě a testoví dvojníci. Pro tato data je nutné dodržet závislosti mezi nimi.

```
1:  procedure ArrangeHeap(M)
2:      H ← M.heap
3:      G ← BuildHeapGraph(H)
4:      G ← CondensateSSC(G)
5:      V ← TopologicalSort(G)
6:      for v in V do
7:          for e in v do
8:              WriteConstruction(e)
9:          end for
10:         for e in v do
11:             WriteInitialization(e)
12:         end for
13:     end for
14: end procedure
```

Algoritmus 6.3: Uspořádání testovacích dat.

Za tím účelem byl navržen algoritmus 6.3. Vstupem je symbolický model. Z jeho haldy se vybuduje graf závislostí, ve kterém se zkondenzují silně souvislé komponenty a získá se tak graf, kde vrcholy představují skupiny cyklicky závislých elementů haldy. Tyto skupiny mohou mít i jeden element, pakliže daný element není součástí žádné cyklické závislosti.

Pomocí topologického uspořádání kondenzovaného grafu se určí pořadí nutné pro inicializaci jednotlivých cyklických skupiny. Pro každou skupinu se poté nejprve vytvoří instance každého elementu a následně se každému elementu inicializují jeho fieldy a mockované metody.

Název	Popis
`Null`	Ověří daná proměnná je `null`.
`NotNull`	Ověří daná proměnná není `null`.
`Equal`	Ověří dané proměnné si jsou rovné (rovnost hodnot, ne identita).
`NotEqual`	Ověří dané proměnná si nejsou rovné (rovnost hodnot, ne identita).
`Same`	Ověří dané proměnné si jsou totožné (identita, ne rovnost hodnot).
`NotSame`	Ověří dané proměnná si nejsou totožné (identita, ne rovnost hodnot).
`Length`	Ověří dané pole má danou délku.
`Count`	Ověří dané `IEnumerable` má daný počet elementů.
`ExceptionThrown`	Ověří výjimka daného typu byla vyhozena (vyžaduje vytvoření delegáta).
`ExceptionNotThrown`	Ověří výjimka daného typu nebyla vyhozena (vyžaduje vytvoření delegáta).
`NoExceptionThrown`	Ověří žádná výjimka nebyla vyhozena (vyžaduje vytvoření delegáta).
`OfType`	Ověří daná proměnná je instance daného typu.
`NotOfType`	Ověří daná proměnná je instance není daného typu.
`Equivalent`	Ověří hodnota dané proměnné je ekvivalentní s danou hodnotou - postupný průzkum do hloubky porovnávaných objektů.
`NotEquivalent`	Ověří hodnota dané proměnné není ekvivalentní s danou hodnotou - postupný průzkum do hloubky porovnávaných objektů..

Tabulka 6.6: Testová primitiva - ověření.

Tento algoritmus nebere v potaz řadu problematik spojených s inicializací dat, například závislosti předané pomocí konstruktoru nebo využití čistě viditelných (veřejné či zpřístupněné) metod objektů (viz část 8.2).

6.3.4 Soubor testů

Jeden běh generátoru testů vygeneruje jeden soubor testů (viz obrázek 6.9), respektive testový projektu. Ten obsahuje informace pro vygenerování C# projektu (soubor .csproj[8]) a případně souboru *AssemblyInfo.cs*[9].

[8] XML soubor obsahující data o projektu, jeho závislosti, konfigurace aj.

[9] Soubor, který obsahuje atributy, které cílí na celé assembly a další informace, o assembly, které jsou zkompilované do modulu CLI.

Testový projekt také obsahuje kolekci testových skupin. Testová skupina sdružuje testové třídy, které testují metody ze tříd ve stejném jmenném prostoru (*namespace*). Při zápise na disk se pro každou skupinu vytvoří vlastní složka odpovídající danému jmennému prostoru.

Testová třída obsahuje informace potřebné pro vygenerování zdrojového kódu třídy. Jedná se mimo jiné o atributy, importované jmenné prostory atp. Pro každou testovanou metodu se vytvoří jedna testovací třída, takže pokud se testuje například pět metod v rámci jedné třídy `Bar`, tak se vygeneruje jedna testová skupina, odpovídající jmennému prostoru třídy `Bar`, která obsahuje pět testových tříd, která každá odpovídá jedné testované metodě. Pokud ve stejném jmenném prostoru jako je třída `Bar` existuje i testovaná třída `Baz`, tak se stejné testové skupiny přidají testové třídy pro testované metody třídy `Baz`.

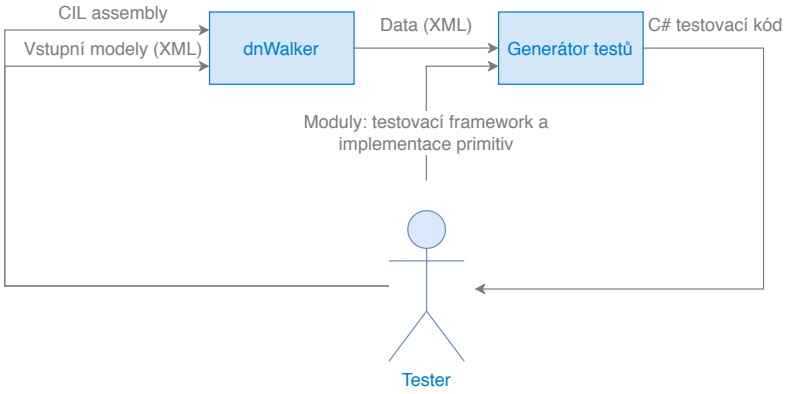
Pro každou iteraci konkolické exekuce je vytvořena jedna či více testových metod (viz sekce 6.3.1), což jsou objekty, které kromě informací o atributech obsahují i výsledný zdrojový kód. Testová třída pak výslednou obsahuje kolekci testových metod.

Jedním z problémů tohoto systému je vhodný způsob předání informací o importovaných jmenných prostorech a způsobu odkazování na knihovny. Nakonec byl zvolen způsob přes atributy. Každý objekt, který potřebuje zajistit import jmenných prostorů nebo závislost na externích knihovnách může být anotovaný atributy `AddNamespaceAttribute` respektive `AddReferenceAttribute`. V aktuální verzi je podporované pouze odkazování na *nuget* balíky[10], nicméně neměl by být problém systém rozšířit i o podporu jiných nástrojů nebo odkazování na lokální knihovny. Tyto atributy jsou pak zpracovány pomocí reflexe (způsob čtení metadat načtených CLI modulů za běhu aplikace a další možnosti jak interagovat přímo s CLR a použity při generování testů.

6.4 Vstupní a výstupní data

Při návrhu celého řešení bylo rozhodnuto o rozdělení na 2 nezávislé moduly, kde se první stará o konkolickou exekuci a druhý o formulaci a vygenerování jednotkových testů. Zároveň jedním z požadavků je možnost testera upravit běh konkolického průzkumu pomocí konfigurace vstupních parametrů a ovlivnit generování testů, respektive využití preferovaných testových

[10] Nástroj pro správu závislostí projektu `https://www.nuget.org/`.



Obrázek 6.10: Tok dat mezi uživatelem, *dnWalkerem* a generátorem testů.

knihoven. Proto je nutné zajistit tok dat mezi testerem a *dnWalkerem* a mezi *dnWalkerem* a jeho konzumentem - generátorem testů (viz obrázek 6.10).

6.4.1 Vstup

Existuje několik možností, jak může tester upravit konkolickou exekuci. První spočívá ve vlastní implementaci rozhraní `IExplorationStrategy`, která řídí celý průzkum. Další mechanismus umožňuje jemné úpravy týkající se přímo zkoumaného programu, a to poskytnutím vstupních podmínek pro zkoumanou metodu. Vstupní podmínkou je míněný predikát nad vstupními argumenty, který se použije jako omezení v kořeni stromu omezení a tím ovlivní prohledávání v celém stromu. Vstupních podmínek je možné definovat více a každá z nich poté vytvoří vlastní nezávislý strom omezení.

V aktuální verzi neexistuje parser pro všechny výrazy podporované *dnWalkerem* (pouze pro primitivní hodnoty a aritmetické a logické operace nad nimi) a proto nelze nastavit vstupní podmínky přímo pomocí logické formule. Nicméně je implementovaný systém tzv. *uživatelských modelů*(*user model*), ze kterého *dnWalker* vygeneruje formuli vstupních podmínek.

Popis XML formátu

Třebaže je možné uživatelské modely popsat pomocí řady markup jazyků (JSON, YAML atp), byl pro aktuální verzi vybrán XML, kvůli flexibilitě,

kterou umožňují jeho atributy. Díky nim je možné přímočaře doplnit metadata jednotlivých struktur (typ, délka pole, identifikátor atp) a odlišit tyto hodnoty od prvků popisující jejich obsah (fieldy, návratové hodnoty mockovaných metod, elementy pole atp).

Kód 6.11 ukazuje příklad takového XML, který popisuje dvě různé vstupní podmínky (modely). Pro ilustraci je v programu 6.12 odpovídající C# kód a v rovnici 6.15 jsou stejné podmínky zformulované pomocí separační logiky, jejichž model je obsažen na obrázku 6.11 (pro zpřehlednění jsou použité zkratky $DO \sim$ `DataObject`, $IDB \sim$ `IDatabase`, $DR \sim$ `DataRecords`, $FDO \sim$ `FileDataObject` a $GetRecs \sim$ `GetRecords`). Následující popis se odkazuje na XML v kódu 6.11.

$$
\begin{array}{lcl}
\Delta_1 & \equiv & this \mapsto (type : DO) \\
 & * & database \mapsto (type : IDB, GetRecs(1) : \mathbf{nil}) \\
\Delta_2 & \equiv & this \mapsto (type : FDO, Author : "John\ Doe") \\
 & * & database \mapsto (type : IDB, GetRecs(id \leqslant 0) : \mathbf{nil}, \\
 & * & GetRecs(id = 1) : dr5, GetRecs(id > 1) : dr2) \\
 & * & dr5 \mapsto (type : DR[], length : 5) \\
 & * & dr2 \mapsto (type : DR[], length : 5, [0] : dc, [1] : dla) \\
 & * & dc \mapsto (type : DR, Name : "Created", IntValue : 15) \\
 & * & dla \mapsto (type : DR, Name : "LastAccess", IntValue : 5)
\end{array}
\tag{6.15}
$$

Každý soubor modelů obsahuje sekci `SharedData` (řádky 2 až 13), která obsahuje objekty a struktury, které mohou být sdílené mezi více modely, a jeden či více modelů (`UserModel`), v tomto případě jsou modely dva (řádky 15 až 24, respektive 26 až 43).

Jakákoliv data mohou být označena atributem *Id* a poté na ně lze vytvořit odkaz pomocí elementu `Reference` (řádek 39). Pro referenční typy je pak zaručena identita.

Při definování datových struktur je možné upravit typ objektu (řádek 28), případně typ elementů pokud se jedná o pole. U pole lze explicitně nastavit jeho délku (řádky 3 a 36) a poté index u jednotlivých elementů. Pokud se index nenastaví, tak se zvolí další možný v pořadí.

Podobný mechanismus je použit i pro nastavení výsledků mockovaných metod založených na počítání volání (takto je nastavené chování metody na řádku 21). Specifikace podmíněného volání metody je na řádcích 34 až 40.

```
<UserModels>
  <SharedData>
    <Array Id="inv" ElementType="DataRecord" Length="2">
      <Object>
          <Name>Created</Name>
          <IntValue>15</IntValue>
      </Object>
      <Object>
          <Name>LastAccess</Name>
          <IntValue>5</IntValue>
      </Object>
    </Array>
  <SharedData>
  <!-- model 1 -->
  <UserModel EntryPoint="DataObject::ReadData">
    <m-this>
      <Object/>
    </m-this>
    <database>
      <Object>
        <GetRecords>null</GetRecords>
      </Object>
    <database>
  </UserModel>
  <!-- model 2 -->
  <UserModel EntryPoint="DataObject::ReadData">
    <m-this>
      <Object Type="FileDataObject">
        <Author>John Doe</Author>
      </Object>
    </m-this>
    <database>
      <Object>
          <GetRecords Condition="id <= 0">null</GetRecords>
          <GetRecords Condition="id == 1">
              <Array Length="5">
          </GetRecords>
          <GetRecords Condition="id > 1">
              <Reference>inv</Reference>
          </GetRecords>
      <Object>
    <database>
  </UserModel>
</UserModels>
```

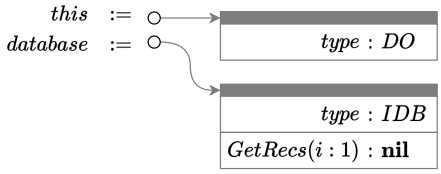
Program 6.11: Příklad uživatelského modelu pro metodu 3.1. Třída `FileDataObject` dědí ze třídy `DataObject`.

Nenastavené hodnoty jsou předmětem symbolické exekuce a odložené inicializace (viz část 6.2.4). V druhém modelu je pevně daná hodnota fieldu *this.Author* := "*JohnDoe*" (řádek 29), zatímco v prvním modelu s jeho hodnotou může konkolická exekuce manipulovat.

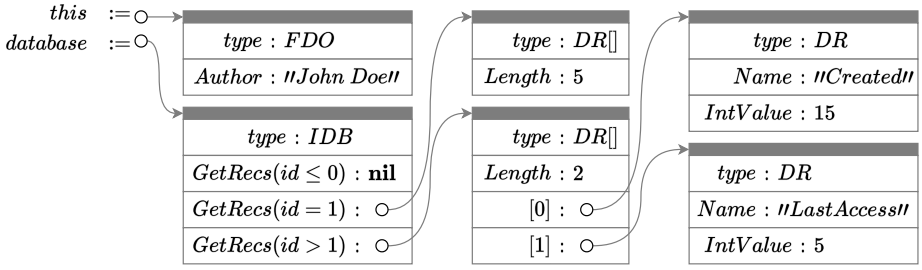
Hlavní využití uživatelských modelů je pomoci *dnWalkeru* správně zpracovat abstrakci komplexních systémů, jako je třeba databáze. Tester tak může pro *dnWalker* omezit možnosti, takže se vygeneruje méně neplatných testovacích dat, případě zavést nějaké podmíněné chování pro konkrétní hodnoty.

```
 1 // model 1
 2 DataObject do = new Mock<DataObject>().Object;
 3
 4 Mock<IDatabase dbMock = new Mock<IDatabase>();
 5 dbMock.Setup(o => o.GetRecords(It.IsAny<int>))
 6   .Returns((DataObject[])null);
 7
 8 IDatabase db = dbMock.Object;
 9
10 // model 2
11 FileDataObject do = new FileDataObject();
12 do.Author = "John Doe";
13
14 Mock<IDatabase dbMock = new Mock<IDatabase>();
15 dbMock.Setup(o => o.GetRecords(It.IsAny<int>))
16   .Returns(id =>
17   {
18     if (id <= 0) return null;
19     else if (id == 0) return new DataRecord[5];
20     else return new DataRecord[] {
21       new DataRecord {
22         Name = "Created",
23         IntValue = 15,
24       },new DataRecord {
25         Name = "LastAccess",
26         IntValue = 5,
27       }
28     };
29   })
30
31 IDatabase db = dbMock.Object;
```

Program 6.12: C# odpovídající uživatelskému modelu 6.11. Abstraktní typy vytvořeny pomocí knihovny *moq*. Třída `FileDataObject` dědí ze třídy `DataObject` a není abstraktní.



(a) : Vstupní model 1, formule Δ_1.



(b) : Vstupní model 2, formule Δ_2.

Obrázek 6.11: Modely odpovídající podmínkám Δ_1 a Δ_2 v rovnici 6.15.

6.4.2 Výstup

Data o konkolickém průzkumu obsahují informace o použité konfiguraci a poté informace o každé iteraci. Nejdůležitější částí je informace o vstupním stavu (vstupní model) a vstupní podmínka, informace o výstupním stavu (výstupní model) a vyhozená, nezpracovaná výjimka. Tato data jsou serializovaná do XML souboru (kód 6.13 ukazuje příklad takového souboru) a jsou poté vstupem pro generátor testů.

```
<Exploration
  AssemblyName="Examples.dll"
  AssemblyFileName="Full\Path\To\Examples.dll"
  MethodSignature="DataObject::Read"
  Solver="Z3Solver"
  Strategy="SmartAllPathsCoverage"
  Start="2022/12/15 12:10:21.495"
  End="2022/12/15 12:10:22.768" Failed="false">
  <Iteration
    Iteration="1"
    Start="2022/12/15 12:10:21.496"
    End="2022/12/15 12:10:21.509"
    PathConstraint="(database == null) &
                    ($this$ != null) & ($this$ != null)"
    Exception="NullReferenceException"
    StandardOutput="" ErrorOutput="">
    <InputModel>
      <Variables>
        <MethodArgument Name="$this$" Value="@00000001" />
        <MethodArgument Name="database" Value="null" />
      </Variables>
      <Heap>
        <ObjectNode Location="@00000001" Type="DataObject">
          <InstanceField DeclaringType="DataObject"
            FieldName="Id" Value="0" />
        </ObjectNode>
      </Heap>
    </InputModel>
    <OutputModel>
      <Variables>
        <MethodArgument Name="$this$" Value="@00000001" />
        <MethodArgument Name="database" Value="null" />
      </Variables>
      <Heap>
        <ObjectNode IsDirty="true" Location="@00000001"
          Type="DataObject">
          <InstanceField DeclaringType="DataObject"
            FieldName="Id" Value="0" />
        </ObjectNode>
      </Heap>
    </OutputModel>
  </Iteration>
  ...
</Exploration>
```

Program 6.13: Příklad XML dat konkolického průzkumu pro metodu `DataObject::Read`, první iterace. Jmenné prostory tříd jsou vynechané.

Kapitola 7

Výsledky

V této kapitole je provedené srovnání *dnWalkeru* s podobnými nástroji. Zvoleny byly nástroje *IntelliTest* a *JDart*, které jsou krátce představeny v kapitole 5.

Srovnání probíhalo na souboru ukázkových metod. Část z nich již byla před představena v předchozích částech textu, zbylé jsou v příloze B. Metody jsou rozdělené do 3 skupin podle vlastností které mají nástroje ověřit:

1. Primitivní data: metody ověřují práci s primitivními typy a omezení nad nimi.
2. Konkrétní data: metody ověřují práci s dynamickými daty (objekty a poli), které mají konkrétní a implementované členy.
3. Abstraktní data: metody ověřují práci s dynamickými daty (objekty, rozhraní a pole), které mají mimo jiné abstraktní či neimplementované metody.

Úplný seznam testovaných metod a srovnání jejich zpracování nástroji jsou v tabulce 7.1. Testy proběhly na počítači s 3.80 GHz procesorem AMD Ryzen 7 5800X a 32.0 GB pamětí, operační systém Windows 10 verze 22H2. *JDart* byl spuštěn pomocí *Dockeru* a zdrojový kód byl manuálně přeložen ze C# do Javy.

Dle očekávání, *JDart* pracuje dobře s primitivními hodnotami, nicméně neposkytuje řešení nelineárních výrazů a není schopen prozkoumávat omezení založená tvaru, velikosti a obsahu dynamických dat. Čas potřebný pro prozkoumání metod je přibližně 5× vyšší než-li u nástroje *dnWalker*. V případě metody `MixedNumbers` dokonce více než 10×. Nižší rozdíl je možné přisoudit mechanismu spuštění, protože *JDart* byl pokaždé spouštěn v novém procesu, tudíž se nevyužily optimalizace v rámci víceúrovňové (*tiered*) JIT kompilace a naplnění cache. Nicméně 10× pomalejší průběh indikuje nižší výkon způsobený programem samotným.

Do tohoto srovnání nejsou započítané časy běhu metody `IndexOf`, protože obsahuje cyklus závislý na délce vstupního pole. Ta musela být kvůli neschopnosti *JDartu* práce s dynamickými daty, nastavena na konstantní hodnotu 5. *dnWalker* ale prohledával pole s délkami 0 – 8, což je exponenciálně více cest.

JDart zároveň nenašel chybovou cestu v metodě `ModuloByZero`, ve které je možné vyhodit výjimku `DivideByNullException`.

Nástroj *IntelliTest* je schopný prozkoumat i omezení pomocí nelineárních operací, avšak v takovém případě uživatele upozorní na možnost o neschopnosti rozhodnutí podmínek. Napříč všemi metodami zajišťuje vysoké pokrytí kódu. Jediná situace, ve které je horší než *dnWalker* je metoda `DataObject::Read` (viz program 3.3), ve které neprozkoumal řádek 35.

Pro konfiguraci vstupních dat nabízí *IntelliTest* uživatelsky příjemný mechanismus. V prvním kroku vygeneruje parametrizované testy, které může uživatel upravit. Pomocí statické třídy `PexAssume` a atributů nad vstupními argumenty může nastavit vstupní podmínky a následně pomocí ověření ověřit výstupní podmínku.

IntelliTest pracuje s omezením pouze na viditelné metody a vlastnosti. Díky tomu dokáže vygenerovat testovací data, která poté může korektně uspořádat. Nicméně z toho plyne neschopnost automatické generace testového dvojníka. Pokud se má v testované metodě prozkoumat chování na základě rozhraní či abstraktní třídy, musí uživatel manuálně poskytnout konkrétní implementaci. Tu nicméně může vytvořit použitím pomocné třídy `PexChoose`, která vytváří nové symbolické proměnné. Lze tak docílit stejného chování, které *dnWalker* poskytuje ve standardním režimu.

Oproti *dnWalkeru* je ale tímto mechanismem snazší formulovat podmíněné chování, které lze napojit i na stav objektu, což *dnWalker* nepodporuje. Nevýhodou tohoto systému je ale silná provázanost s knihovnami *IntelliTest*, respektive *Pex*.

Zároveň řeší problém, který *dnWalker* v aktuální verzi ignoruje, a to vytvoření instancí tříd s konstruktorem s argumenty. Uživatel může poskytnou vytvářecí metodu, ve které inicializuje argumenty a vrátí potřebnou instanci.

Program 7.1 ukazuje příklad testu vygenerovaným nástrojem *IntelliSense*.

dnWalker poskytuje dobré výsledky v průzkumu metody s lineárními

```
 1 // Test stub
 2 [PexMethod]
 3 public int MethodWithArgsTest(
 4   [PexAssumeUnderTest] SimpleMethod target,
 5   [PexAssumeNotNull] BarMock bar,
 6   double x,
 7   double y
 8 )
 9 {
10  // TODO: add assumptions
11  int result = target.MethodWithArgs(bar, x, y);
12  return result;
13  // TODO: add assertions
14 }
15
16 // Manually specified "parametrized" mock
17 public class BarMock : IBar
18 {
19   public int GetValue()
20   {
21     return PexChoose.Value<int>("GetValue");
22   }
23
24   public int GetValueWithArgs(double x, double y)
25   {
26     if (x < y) return 5;
27     else return 10;
28   }
29 }
30
31 // Actual test
32 [TestMethod]
33 [PexGeneratedBy(typeof(SimpleMethodTest))]
34 public void MethodWithArgsTest815()
35 {
36     int i;
37     SimpleMethod s0 = new SimpleMethod();
38     s0.Value = 0;
39     SimpleMethodTest.BarMock s1 = new SimpleMethodTest.BarMock();
40     i = this.MethodWithArgsTest(s0, s1, 0, 0);
41     PexAssert.AreEqual<int>(1, i);
42     PexAssert.IsNotNull((object)s0);
43     PexAssert.AreEqual<int>(0, s0.Value);
44 }
```

Program 7.1: Test vygenerovaný pomocí nástroje *IntelliTest* pro metodu v programu B.12.

omezeními, omezeními nad dynamickými daty i v přítomnosti potenciálně nekonečných cyklů. Řešení nelineárních omezení ale není implementované a proto v nich dosahuje horších výsledků. Obdobně nemá pokryté všechny případy plynoucí z omezení datových typů (například aktivní hledání chyb z důvodu `OverflowException`).

Z hlediska generování testů má oproti *IntelliSense* hrubší a méně flexibilní systém konfigurace vstupních argumentů. Na druhou stranu vygenerované testy nejsou závislé na třídách nástroje *dnWalker*, ale pouze na třídách použitých testovacích a izolačních frameworků. Příklady testů vygenerovaných *dnWalkerem* byly představeny v předchozím textu, zejména v kapitole 3.

dnWalker nicméně nepodporuje celou šíři .NET možností. Nedokáže zpracovat například konstruktory s parametry, generické typy a neomezuje se čistě na viditelné členy testovaných tříd, což vede k manipulaci s neveřejnými fieldy, které je nutné inicializovat pomocí reflexe a lze docílit stavů, které při korektním použití třídy nemůžou nastat a nejsou proto validním testovacím případem.

Zásadní výhoda *dnWalkeru* oproti *IntelliSense* je nezávislost na verzi .NET (a obecně podpora jakéhokoliv CIL modulu) a na prostředí. *IntelliSense* lze použít pouze přes grafické rozhraní IDE Visual Studio (respektive jeho komerčních edicí), a pouze pro .NET Framework[1].

dnWalker tato omezení nesdílí, respektive v podpoře všech verzí se vyskytuje problém v použité knihovně *dnlib* při importu standardních typů, který je řešen pomocí vytvoření vlastní vyhledávací tabulky těchto typů[2].

[1] Vývoj .NET probíhá ve dvou větvích: .NET Framework a .NET Core, přičemž druhá je aktivně rozvíjená, zatímco první pouze udržovaná kvůli kompatibilitě, viz `https://versionsof.net/`

[2] `https://github.com/0xd4d/dnlib/issues/453`

Metoda	dnWalker				JDart		IntelliTest[f]			
	Cesty[a]	Čas [ms]	Testy	Hrany[%]	Cesty[a]	Čas [ms]	Bloky	Běhy	Testy	Hrany[%]
MixedNumbersB.1	4/0/0	$42,54 \pm 8,71$	4	100	0/4/0	$7355,00 \pm 58,38$	7/7	3	4	100
UsePowB.2	3/0/0	$29,48 \pm 5,04$	3	100	0/1/1	$122,50 \pm 5,28$	6/6	3 [b]	4	100
UseSinB.3	1/0/2	$22,32 \pm 1,04$	1	25	0/2/1	$125,00 \pm 3,44$	6/6	3 [b]	3	100
UseCosB.4	1/0/2	$21,65 \pm 0.35$	1	25	0/2/1	$121,75 \pm 2,39$	6/6	3 [b]	3	100
DivideByZeroB.5	2/1/0[c]	$29,63 \pm 1,97$	3	100	2/1/0	$135,25 \pm 3,98$	3/3	3 [c]	4	100
ModuloByZeroB.6	2/1/0[c]	$34,96 \pm 8,23$	3	100	2/0/0	$148,25 \pm 12,42$	3/3	3 [c]	4	100
Min36.4	4/0/0	$47,32 \pm 8,53$	4	100	0/4/0	$127,00 \pm 5,36$	5/5	3	3	100
Segment::AppendB.7	13/0/0	$686,42 \pm 58,14$	13	100			8/8	103 [d]	3	100
Segment::InsertB.8	1/0/0	$39,85 \pm 3,50$	1	100			1/1	1	1	100
Segment::DeleteB.9	29/0/0	$1128,86 \pm 67,56$	37	100			17/17	131 [d]	13	100
Segment::CountB.10	21/0/0	$566,83 \pm 50,94$	21	100			6/6	106 [d]	5	100
IndexOf2.3	15/1/0	$370,87 \pm 36,19$	16	100	0/11/0	$135,75 \pm 6,80$	8/8	106 [d]	3	100
DataObject::Read3.1	32/9/0	$1298,55 \pm 42,84$	60	100			40/41	161[de]	23	95
NoArgsB.11	2/1/0	$36,89 \pm 1,87$	3	100			4/4	3	3	100
WithArgsB.12	2/1/0	$41,57 \pm 3,03$	3	100			4/4	2	2	100

Tabulka 7.1: Srovnání průzkumu metod a generování testů pomocí nástrojů *dnWalker*, *JDart* a *IntelliTest*.

[a] Cesty jsou ve formátu *ok/chyba/nerozhodnutelné*.
[b] Hlásí problém s rozhodnutím splnitelnosti některých operací (`sin`, `cos`, `pow` a násobení floatů).
[c] Možná chyba `OverflowException` nebyla odhalena.
[d] Průzkum skončil po dosažení limitu běhů bez nalezení nového testu (*100*).
[e] Některé cesty nebyly prohledány kvůli přesažení časového limitu pro solver (*1s*).
[f] Nelze změřit čas protože nástroj je součástí IDE Visual Studio a nenabízí API, které by podobná měření umožňovalo (`https://stackoverflow.com/questions/35305709/measuring-test-generation-time-in-pex`).

Kapitola 8

Závěr

8.1 Naplnění cílů

V úvodu je vytyčeno 6 cílů, které má výsledný nástroj *dnWalker* naplnit.

1. Vstupem nástroje je CLI modul obsahující testovanou metodu a žádný či více XML souborů, pomocí kterých může uživatel nakonfigurovat vstupní argumenty pro jednotlivé testované metody.

   Třebaže možnosti tohoto formátu jsou velmi flexibilní, je *dnWalker* schopen v aktuální verzi zpracovat flexibilnější omezení daná separační logikou. Není ale implementovaný způsob, jak tyto výrazy předat do nástroje.

2. *dnWalker* implementuje základní verzi konkolické exekuce s několika jednoduchými heuristikami. Dle experimentálních výsledků je hlavní heuristika v některých případech efektivnější než komerční nástroj *IntelliTest*. Nicméně mnoho studovaných možností konkolické exekuce (například kompoziční řešení nelineárních omezení) nemá implementované a chybí obsluha řady konkrétních situací (například hledání podmínky pro vyvolání `OverflowException`). Uživatel má možnost dodat vlastní prohledávací strategii.

3. *dnWalker* generuje funkční jednotkové testy pro kód odpovídající ukázkovým příkladům použitých v kapitole 7, které zajistí pokrytí dle zvolené strategie. Pro metodu s cykly či rekurzí ale vygeneruje mnoho redundantních testů, které nerozpozná, jako to dělá třeba nástroj *IntelliTest*.

4. Pomocí konfigurace vstupních argumentů může *dnWalker* mockovat chování abstraktních metod či metod rozhraní. Uživatel má k dispozici 2 základní mechaniky mockování metod a to pomocí explicitního výčtu návratových hodnot, nebo pomocí podmíněného volání, kde *dnWalker* vybere návratovou hodnotu podle splnění přiložených predikátů nad argumenty mockované metody.

5. *dnWalker* implementuje modulární mechanismus pro generování jednotlivých částí testů. Uživatel může zvolit kombinaci testovacích a izolačních frameworků, dle svých preferencí.

Výsledná verze *dnWalkeru* je nástroj, který nabízí neúplnou podporu CLI. Chybí například paralelní exekuce, podpora generických typů, podpora *struct* (komplexní hodnotové typy) nebo uživatelsky přívětivý systém pro konfiguraci jak nástroje samotného tak testovaných metod.

Podle experimentálních výsledků je ale již nyní schopen v některých případech překonat komerční nástroj (*IntelliTest*). Má vysoce modulární strukturu, pomocí které je vývoj chybějících vlastností značně usnadněn.

8.2 Další kroky

Řada z oblastí, na kterých je nutné pracovat již byla popsána v předchozích částech textu. Níže je rozepsáno několik dalších.

Omezení nad typy. CLI nabízí bohatý systém typů a možnosti, jak pomocí nich vytvářet podmínky. Například výraz `if (obj is Bar bar)`, který vytváří omezení nad typem proměnné *obj*. Aktuálně *dnWalker* podobné výrazy nepodporuje. Podpora typové bezpečnosti obecně vyžaduje komplexní teorii, která bude pracovat s polymorfismem, dědičností a možností implementovat rozhraní. *dnWalker* by v některých případech i mohl generovat nové typy, tak, aby odpovídaly podmínkám (například série zmíněných `is` výrazů, které dané proměnné přinutí implementovat řadu rozhraní).

Indukční predikáty separační logiky. V aktuální verzi *dnWalkeru* je implementován nejmenší možný fragment separační logiky. Tato implementace by se měla rozšířit alespoň o podporu indukčních predikátů. Práci s nimi popisuje [58]. Formule 8.1 ukazuje příklad predikátu, který popisuje všechny seřazené spojové seznamy o délce len, jejichž první prvek je zespoda omezen min.

$$\begin{aligned} &\mathbf{sortll}(root, min, len) \equiv \\ &\quad (root = \mathbf{nil} \wedge len = 0) \\ &\vee \quad (\exists v, n, l \cdot root \mapsto (val : v, nxt : n) * \mathbf{sortll}(n, v, l) \wedge \\ &\quad\ \ l = len - 1 \wedge min \leqslant v) \end{aligned} \tag{8.1}$$

Podpora indukčních predikátů povede na řadu možností, jak vylepšit a ovlivnit fungování nástroje. V první řadě bude uživateli poskytnut silný nástroj pro popis vstupních podmínek. Za druhé to umožní validaci testovacích případů a zabrání to tak *dnWalkeru* generovat nesmyslné testovací případy, po vzoru prací [73, 74].

Kompoziční průzkum. Jednou z hlavních výhod konkolické exekuce je možnost nahrazení výrazů konstantou, která odpovídá jejich hodnotě pro danou vstupní podmínku. Tento mechanismus je možné využít ke kompozičními řešení omezení. Průzkum je pak řízen, aby našel hodnoty pro nelineární omezení pomocí konkrétní exekuce. Tato strategie pro řešení nelineárních omezení je popsána třeba v [41] nebo v [71].

Širší využiží SMT teorií. Ve standardních SMT solverech je implementována řada teorií pro práci s častými datovými strukturami. Například teorie polí, množin atp. Pomocí výrazů těchto teoriích je možné zjednodušeně vyjádřit omezení nad komplexními strukturami, například pro třídu `Dictionary<TKey,TValue>`, kde vyjádření i pouze jednoho mapování z konstanty na konstantu vyžaduje velmi komplikovanou formuli separační logiky. Ta navíc závisí na vnitřní implementaci třídy ze základních modulů, která ale testera vůbec nezajímá. Abstrakce takových známých typů a rozhraní, které lze popsat pomocí zavedených SMT teorií, tak může vést k zásadnímu zjednodušení prohledávání.

Využití konstruktoru a veřejných metod pro uspořádání. Jedním z nejzávažnějších nedostatků *dnWalkeru* je jeho naivní přístup k uspořádání konkrétních dat. Nástroj neřeší, zda jsou data zapouzdřená a mají dedikované přístupové metody nebo jestli je daná hodnota nastavitelná pouze pomocí konstruktoru. Počítá s tím, že všechny třídní proměnné jsou nezávislé hodnoty, které je možné nastavit jak je potřeba pro splnění dané podmínky, což při generování testů vede k použití reflexe (způsob interakce s runtime, který umožňuje mimo jiné nastavit hodnotu prakticky libovolné třídní proměnné) a jiným antipatternům.

dnWalker by měl zanalyzovat testované typy a pro uspořádání jejich instancí použít výhradně jejich veřejné, respektive viditelné, členy. A to jak při generování testů tak při konkolické exekuci. S touto podmínkou pracuje *IntelliTest* a zajímavý přístup také nabízí nástroj *SUSHI*[18]. Použití tohoto omezení také zajistí validitu testovacích dat.

Integrace do vývojového cyklu. V aktuální verzi je *dnWalker* schopný generovat použitelné jednotkové testy. Nicméně jeho zapojení do vývojového cyklu softwaru je velmi náročné. Dalším logickým krokem je proto zjednodušení použití *dnWalkeru*. Největší výzvou této integrace je umožnění úprav nástroje uživatelem – volbu vlastního testovacího a izolačního frameworku, případně využití dalších testovacích knihoven.

Příloha A

Common Intermediate Language (CIL)

CIL je objektově orientovaný zásobníkový jazyk. Jeho instrukce se skládají z operačního kódu a popřípadě parametru. Parametr může být konkrétní hodnota (číslo) nebo odkaz v rámci metadat (typ, metody, řetězec..). Tabulka A.1 Popisuje varianty instrukcí, které jsou specifikovány přímo v operačním kódu.

`ovf`	součástí provedení je kontrola přetečení a může proto vyhodit výjimku `OverflowException`
`un`	indikuje *unsigned* (pro celočíselné hodnoty) nebo *unordered* (pro floating point) porovnání
n	specializace instrukce pro konstantu danou číslem *n*
`s`	indikuje zkrácenou verzi, parametr bude $1B$ číslo namísto $2B$

Tabulka A.1: Varianty instrukcí

V následující tabulce jsou vypsány některé instrukce *CIL* (mimo jiné ty, které jsou zmíněné v textu). Úplný seznam je v [101].

Název	**Varianty**	**Popis**
Základní instrukce		
`conv`	`ovf`, `un`	Provede konverzi hodnoty na zásobníku. Existují varianty pro všechny numerické typy.
`dup`		Zduplikuje hodnotu na zásobníku.
`ldarg`	`0-3`, `s`	Vloží na zásobník hodnotu *i*-tého argumentu metody.
`ldc`	`-1 - 8`, `s`	Vloží na zásobník konstantu. Existují varianty pro všechny numerické typy.
`ldind`		Načte hodnotu ze spravovaného ukazatele (určeno hodnotou na zásobníku).
`ldloc`	`0-3`	Vloží na zásobník hodnotu *i*-té lokální proměnné.
`ldloca`	`s`	Vloží na zásobník adresu *i*-té lokální proměnné.
`ldnull`		Vloží na zásobník hodnotu `null`.
`ldstr`		Vloží na zásobník konstantní řetězec.
`nop`		Žádná operace.
`pop`		Zahodí hodnotu na zásobníku.
`ret`		Návrat z metody.
`starg`	`s`	Uložení hodnoty (určeno hodnotou na zásobníku) do *i*-tého argumentu.

Název	Varianty	Popis
`stloc`	`0-3`	Uloží hodnotu (určeno hodnotou na zásobníku) do i-té lokální proměnné.
`stind`		Uloží hodnotu (určeno hodnotou na zásobníku) na adresu (určeno hodnotou na zásobníku). Existují varianty pro všechny datové typy.
Aritmeticko logické instrukce		
`add`	`ovf`, `un`	Sečte dvě hodnoty a vrátí výsledek. Varianty pouze pro celočíselné argumenty.
`and`		Provede bitové $\wedge$ nad celočíselnými hodnotami a vrátí výsledek.
`ceq`		Vrátí 1 (**true**), pokud jsou si hodnoty rovné, jinak 0 (**false**).
`cgt`	`un`	Vrátí 1 (**true**), pokud je $lhs > rhs$, jinak 0 (**false**).
`ckfinite`		Vyhodí `ArithmeticException` pokud hodnota není konečná.
`clt`	`un`	Vrátí 1 (**true**), pokud je $lhs < rhs$, jinak 0 (**false**).
`div`	`un`	Provede (celočíselné) dělení a vrátí výsledek. V případě celočíselných operandů může vyhodit výjimku `DivideByZeroException`.
`mul`	`ovf`, `un`	Provede (celočíselné) násobení a vrátí výsledek. Varianty pouze pro celočíselné argumenty.
`neg`		Neguje hodnotu na zásobníku.
`not`		Bitové doplněk hodnoty na zásobníku.
`or`		Bitové logické $\vee$, pouze pro celočíselné hodnoty.
`rem`	`un`	Zbytek po dělení celočíselných hodnot.
`shl`		Bitový posun vlevo, pro celočíselné hodnoty.
`shr`	`un`	Bitový posun vpravo, pro celočíselné hodnoty.
`sub`	`ovf`, `un`	Provede rozdíl hodnot na zásobníku. Varianty pouze pro celočíselné argumenty.
`xor`		Bitové logické $\oplus$, pouze pro celočíselné hodnoty.
Větvící instrukce		
`beq`	`s`	Větví k cíli, pokud jsou si hodnoty rovné.
`bge`	`s`, `un`	Větví k cíli, pokud je $lhs \geqslant rhs$.
`bgt`	`s`, `un`	Větví k cíli, pokud je $lhs > rhs$.
`ble`	`s`, `un`	Větví k cíli, pokud je $lhs \leqslant rhs$.
`blt`	`s`, `un`	Větví k cíli, pokud je $lhs < rhs$.
`bne`	`s`, `un`	Větví k cíli, pokud je $lhs \neq rhs$.
`br`	`s`	Větví k cíli.
`brfalse`	`s`	Větví k cíli pokud je hodnota 0 (e.i. **false**, `null`).
`brtrue`	`s`	Větví k cíli pokud je hodnota různá od 0 (e.i. **true**, různá od `null`).
`switch`		Větví k i-tému (určeno hodnotou na zásobníku) cíli (určeno parametrem – pole instrukcí).
Instrukce modelu objektu		
`box`		Zabalí hodnotu do objektu.
`castclass`		Převede instanci na jiný typ.
`isinst`		Otestuje zda je hodnota instancí třídy dané parametrem a vrátí `null` nebo danou instanci.
`ldelem`		Načte položku pole na indexu (určeno hodnotami na zásobníku). Existují varianty pro všechny datové typy.
`ldelema`		Načte spravovaný ukazatel na položku pole na indexu (určeno hodnotami na zásobníku).
`ldfld`		Načte hodnotu field (určeno parametrem) objektu (určeno hodnotou na zásobníku).

Název	Varianty	Popis
`ldflda`		Načte spravovaný ukazatel na field (určeno parametrem) objektu (určeno hodnotou na zásobníku).
`ldlen`		Načte délku pole (určeno hodnotou na zásobníku).
`ldsfld`		Načte hodnotu statického fieldu (určeno parametrem).
`ldsflda`		Načte adresu statického fieldu (určeno parametrem).
`newarr`		Alokuje nové pole (typ elementů určen parametrem) a dané délce (určeno hodnotou na zásobníku) a vloží referenci na zásobník.
`newobj`		Alokuje nový objekt (typ určen parametrem) a zavolá konstruktor (určen parametrem), který vloží referenci na zásobník.
`stelem`		Uložení hodnoty (určeno hodnotou na zásobníku) do i-tého elementu (určeno hodnotou na zásobníku) pole (určeno hodnotou na zásobníku). Existují varianty pro všechny datové typy.
`stfld`		Uloží hodnotu (určeno hodnotou na zásobníku) do fieldu (určeno parametrem) objektu (určeno hodnotou na zásobníku).
`stsfld`		Uloží hodnotu (určeno hodnotou na zásobníku) do statického fieldu (určeno parametrem).
Instrukce zavolání metod		
`call`		Zavolá metodu popsanou parametrem instrukce.
`calli`		Zavolá metodu určenou hodnotou na zásobníku.
`callvirt`		Zavolá metodu určenou parametrem instrukce a konkrétní instancí na zásobníku (provede vyhledávání overriden metod).
Instrukce modelu výjimek		
`endfault`		Zakončí *fault* blok.
`endfilter`		Zakončí *filter* blok.
`endfinally`		Zakončí *finally* blok.
`leave`	`s`	Opustí chráněnou oblast kódu.
`rethrow`		Opakované vyhození aktuální výjimky.
`throw`		Vyhodí výjimku (určenou hodnotou na zásobníku).

Příloha B

Testovací metody

```
1 double MixedNumbers(double x, int n) {
2   if (x < n) {
3     return x * n;
4   }
5   else if (x == n) {
6     if (x >= 0) {
7       return x / n;
8     }
9   }
10  return x + n;
11 }
```

Program B.1: Kód metody `MixedNumbers`, primitivní data.

```
1 int UsePow(double x, double y, double z) {
2   if (Math.Pow(x, y) < z && z > 36) {
3     return 0;
4   }
5   return 1;
6 }
```

Program B.2: Kód metody `UsePow`, primitivní data.

```
1 int UseSin(double x, double y) {
2   if (Math.Sin(x) >= y && y < -0.5) {
3     return 0;
4   }
5   return 1;
6 }
```

Program B.3: Kód metody `UseSin`, primitivní data.

```
1 int UseCos(double x, double y) {
2   if (Math.Cos(x) >= y && y > 0.5) {
3     return 0;
4   }
5   return 1;
6 }
```

Program B.4: Kód metody `UseCos`, primitivní data.

```
1 int DivideByZero(int x, int y, int z) {
2     if (5 * (x / y) + 3 >= z) {
3         return 1;
4     }
5     return 0;
6 }
```

Program B.5: Kód metody `DivideByZero`, primitivní data.

```
1 int ModuloByZero(int x, int y, int z) {
2     if (5 * (x % y) + 3 >= z) {
3         return 1;
4     }
5     return 0;
6 }
```

Program B.6: Kód metody `ModuloByZero`, primitivní data.

```
1 void Append(int[] data) {
2   Segment s = this;
3   while (s.Next != null) {
4     s = s.Next;
5   }
6
7   s.Insert(data);
8 }
```

Program B.7: Kód metody `Segment::Append`, součást třídy definované v 6.10, konkrétní data.

```
1 void Insert(int[] data) {
2   Next = new Segment() {
3     Data = data,
4     Next = Next
5   };
6 }
```

Program B.8: Kód metody `Segment::Insert`, součást třídy definované v 6.10, konkrétní data.

```
1 Segment Delete(int[] data) {
2   if (data == Data) {
3     return Next;
4   }
5
6   if (Data != null && data != null &&
7       Data.Length == data.Length) {
8     bool equals = true;
9
10    for (int i = 0; i < Data.Length; ++i) {
11      equals = Data[i] == data[i];
12
13      if (!equals) {
14          break;
15      }
16    }
17
18    if (equals) {
19      return Next;
20    }
21  }
22
23  Next = Next?.Delete(data);
24  return this;
25 }
```

Program B.9: Kód metody `Segment::Delete`, součást třídy definované v 6.10, konkrétní data.

```
1 int Count() {
2   int cnt = Data?.Length ?? 0;
3   if (Next != null) {
4     cnt += Next.Count();
5   }
6   return cnt;
7 }
```

Program B.10: Kód metody `Segment::Count`, součást třídy definované v 6.10, konkrétní data.

```
1 int NoArgs(IBar bar) {
2   if (2 * bar.GetValue() - 3 == value) {
3     return 4;
4   }
5   return 5;
6 }
```

Program B.11: Kód metody `NoArgs`, abstraktní data.

```
1 int WithArgs(IBar bar, double x, double y) {
2   if (bar.GetValue(x, y) == Value) {
3     return 1;
4   }
5   return 0;
6 }
```

Program B.12: Kód metody `WithArgs`, abstraktní data.

Příloha C

Seznam použitých zkratek

A

AAA: arrange – act – assert
API: application programming interface

B

BFS: breadth first searching, prohledávání do hloubky
BLI: bounded lazy initialization
BLISS: bounded lazy initialization with SAT support

C

C/DC: condition/decision coverage
CC: condition coverage
CFG: control flow graph, graf toku řízení
CIL: common intermediate language
CLI: common language infrastructure
CLP: constrained logic program
CLR: common language runtime
CLS: common language specification
CoreCLR: core common language infrastructure
CRC: cyclic redundancy chceck
CTL: computational tree logic
CTL*: computational tree logic*
CTS: common type system
CUTE: concolic unit testing engine
CVC4: cooperating validity checker, v4
CVC5: cooperating validity checker, v5

D

DART: directed automated random testing
DC: decision coverage
DFS: depth first searching, prohledávání do hloubky

F

FSA: finite state automata

H

HEX: heap explarotary language

I

IDE: integrated development evnironment

J

JBSE: java bytecode symbolic evaluator
JIT: just in time
JPF: java pathfinder
JSON: javascript object notation
JVM: java virtual machine

L

LEARCH:
LI: lazy initialization, odložená inicializace
LTL: linear temporal logic

M

MC/DC: modified condition/decision coverage
MCC: multiple condition coverage
ML: meta language
MPI: message passing interface

N

NASA: national aerospace agency

P

ParaDySe: parametric dynamic symbolick execution
PET: partial evaluation-based test case generation tool
POR: partial order reduction, redukce parciálního pořadí

Q

qf: quantifier free

S

SAGE: scalable automated guided execution
SAT: satisfiability
SDL: specification and description language
SMART: systematic modular automated random testing
SMT: satisfiability modulo theories
SQL: structured query language

T

TDL: test depth level

V

VES: virtual execution system

W

WASM: web assembly
WASP: web assembly symbolic processor

X

XML: extensible markup language
XRT: exploring runtime

Y

YAML: YAML ain't markup language

Příloha D

Literatura

[1] ALBERT, Elvira, Miguel G-MEZ-ZAMALLOA a Germán PUEBLA. PET: Partial Evaluation-based Test Case Generator for Bytecode. In: *Proceedings of the ACM SIGPLAN 2010 workshop on Partial evaluation and program manipulation - PEPM '10* [online]. New York, New York, USA: ACM Press, 2010, 2010, 25- [cit. 2022-03-07]. ISBN 9781605587271. Dostupné z: doi:10.1145/1706356.1706363

[2] ALBERT, Elvira, Israel CABANAS, Antonio FLORES-MONTOYA, Miguel GOMEZ-ZAMALLOA a Sergio GUTIERREZ. JPET: An Automatic Test-Case Generator for Java. In: *2011 18th Working Conference on Reverse Engineering* [online]. IEEE, 2011, 2011, s. 441-442 [cit. 2021-10-12]. ISBN 978-1-4577-1948-6. Dostupné z: doi:10.1109/WCRE.2011.67

[3] ANAND, Saswat, Corina S. PĂSĂREANU a Willem VISSER. Symbolic Execution with Abstract Subsumption Checking. In: VALMARI, Antti, ed. *Model Checking Software* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 2006, s. 163-181 [cit. 2022-03-05]. Lecture Notes in Computer Science. ISBN 978-3-540-33102-5. Dostupné z: doi:10.1007/11691617_10

[4] ANAND, Saswat, Corina S. PĂSĂREANU a Willem VISSER. JPF—SE: A Symbolic Execution Extension to Java PathFinder. In: GRUMBERG, Orna a Michael HUTH, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 134-138 [cit. 2022-11-03]. Lecture Notes in Computer Science. ISBN 978-3-540-71208-4. Dostupné z: doi:10.1007/978-3-540-71209-1_12

[5] ANAND, Saswat, Patrice GODEFROID a Nikolai TILLMANN. Demand-Driven Compositional Symbolic Execution. In: RAMAKRISHNAN, C. R. a Jakob REHOF, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 367-381 [cit. 2022-03-05]. Lecture Notes in Computer Science. ISBN 978-3-540-78799-0. Dostupné z: doi:10.1007/978-3-540-78800-3_28

[6] ANAND, Saswat, Mayur NAIK, Mary Jean HARROLD a Hongseok YANG. Automated concolic testing of smartphone apps. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12* [online]. New York, New York, USA: ACM Press, 2012, 2012, 1- [cit. 2021-10-10]. ISBN 9781450316149. Dostupné z: doi:10.1145/2393596.2393666

[7] BARBOSA, Haniel, Clark BARRETT, Martin BRAIN, et al. Cvc5: A Versatile and Industrial-Strength SMT Solver. In: FISMAN, Dana a Grigore ROSU, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Cham: Springer International Publishing, 2022, 2022-03-30, s. 415-442 [cit. 2022-12-13]. Lecture Notes in Computer Science. ISBN 978-3-030-99523-2. Dostupné z: doi:10.1007/978-3-030-99524-9_24

[8] BARRETT, Clark W., Roberto SEBASTIANI, Sanjit A. SESHIA a Ceasare TINELLI. Satisfiability Modulo Theories. In: BIERE, Armin, Marijn HEULE, Hans van MAAREN a Toby WALSH, ed. *Handbook of Satisfiability*. Amsterdam: IOS Press, c2009, s. 825-885. ISBN 978-1-58603-929-5.

[9] BERRETT, Clark, Pascal FONTAINE a Ceasare TINELLI. SMT-LIB The Satisfiability Module Theories Library. *The University of Iowa* [online]. Iowa City, Iowa: The University of Iowa, 2016 [cit. 2022-12-27]. Dostupné z: `https://smtlib.cs.uiowa.edu/`

[10] BOEHM. Software Engineering. *IEEE Transactions on Computers* [online]. 1976, **C-25**(12), 1226-1241 [cit. 2022-12-21]. ISSN 0018-9340. Dostupné z: doi:10.1109/TC.1976.1674590

[11] BOEHM, Barry W. Software Engineering-as It Is. In: *CSE '79: Proceedings of the 4th International Conference on Software Engineering*. Munich, Germany: IEEE Press, 1979, s. 11-21. Dostupné z: doi:10.5555/800091.802916

[12] BOEHM, B.W. a P.N. PAPACCIO. Understanding and controlling software costs. *IEEE Transactions on Software Engineering* [online]. 1988, **14**(10), 1462-1477 [cit. 2022-12-21]. ISSN 00985589. Dostupné z: doi:10.1109/32.6191

[13] BOEHM, Barry W. Software Engineering Economics. In: BROY, Manfred a Ernst DENERT, ed. *Pioneers and Their Contributions to Software Engineering* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, 2001, s. 99-150 [cit. 2022-12-21]. ISBN 978-3-540-42290-7. Dostupné z: doi:10.1007/978-3-642-48354-7_5

[14] BOONSTOPPEL, Peter, Cristian CADAR a Dawson ENGLER. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In: RAMAKRISHNAN, C. R. a Jakob REHOF, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 351-366 [cit. 2022-03-05]. Lecture Notes in Computer Science. ISBN 978-3-540-78799-0. Dostupné z: doi:10.1007/978-3-540-78800-3_27

[15] BORGES, Mateus, Marcelo D'AMORIM, Saswat ANAND, David BUSHNELL a Corina S. PASAREANU. Symbolic Execution with Interval Solving and Meta-heuristic Search. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* [online]. IEEE, 2012, 2012, s. 111-120 [cit. 2022-11-08]. ISBN 978-0-7695-4670-4. Dostupné z: doi:10.1109/ICST.2012.91

[16] BRAIONE, Pietro, Giovanni DENARO a Mauro PEZZÈ. Symbolic execution of programs with heap inputs. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* [online]. New York, NY, USA: ACM, 2015, 2015-08-30, s. 602-613 [cit. 2021-10-12]. ISBN 9781450336758. Dostupné z: doi:10.1145/2786805.2786842

[17] BRAIONE, Pietro, Giovanni DENARO a Mauro PEZZÈ. JBSE: a symbolic executor for Java programs with complex heap inputs. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* [online]. New York, NY, USA: ACM, 2016, 2016, s. 1018-1022 [cit. 2021-10-12]. ISBN 9781450342186. Dostupné z: doi:10.1145/2950290.2983940

[18] BRAIONE, Pietro, Giovanni DENARO, Andrea MATTAVELLI a Mauro PEZZÈ. Combining symbolic execution and search-based testing for programs with complex heap inputs. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* [online]. New York, NY, USA: ACM, 2017, 2017-07-10, s. 90-101 [cit. 2021-10-12]. ISBN 9781450350761. Dostupné z: doi:10.1145/3092703.3092715

[19] BROTHERSTON, James, Nikos GOROGIANNIS a Rasmus L. PETERSEN. A Generic Cyclic Theorem Prover. In: JHALA, Ranjit a Atsushi IGARASHI, ed. *Programming Languages and Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, 2012, s. 350-367 [cit. 2022-04-28]. Lecture Notes in Computer Science. ISBN 978-3-642-35181-5. Dostupné z: doi:10.1007/978-3-642-35182-2_25

[20] BROTHERSTON, James, Carsten FUHS, Juan A. Navarro PÉREZ a Nikos GOROGIANNIS. A decision procedure for satisfiability in separation logic with inductive predicates. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* [online]. New York, NY, USA: ACM, 2014, 2014-07-14, s. 1-10 [cit. 2022-04-28]. ISBN 9781450328869. Dostupné z: doi:10.1145/2603088.2603091

[21] BURES, Miroslav a Bestoun S. AHMED. On the Effectiveness of Combinatorial Interaction Testing: A Case Study. In: *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)* [online]. IEEE, 2017, 2017, s. 69-76 [cit. 2022-10-10]. ISBN 978-1-5386-2072-4. Dostupné z: doi:10.1109/QRS-C.2017.20

[22] BURNIM, Jacob a Koushik SEN. Heuristics for Scalable Dynamic Test Generation. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* [online]. IEEE, 2008, 2008, s. 443-446 [cit. 2022-03-05]. ISBN 978-1-4244-2187-9. Dostupné z: doi:10.1109/ASE.2008.69

[23] CADAR, Cristian, Daniel DUNBAR a Dawson ENGLER. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. San Diego, California: USENIX Association, 2008, s. 209-224. Dostupné z: doi:10.5555/1855741.1855756

[24] CADAR, Cristian, Vijay GANESH, Peter M. PAWLOWSKI, David L. DILL a Dawson R. ENGLER. EXE. *ACM Transactions on Information and System Security* [online]. 2008, **12**(2), 1-38 [cit. 2022-11-03]. ISSN 1094-9224. Dostupné z: doi:10.1145/1455518.1455522

[25] CLARKE, Edmund M., Thomas A. HENZINGER a Helmut VEITH. Introduction to Model Checking. In: CLARKE, Edmund M., Thomas A. HENZINGER, Helmut VEITH a Roderick BLOEM, ed. *Handbook of Model Checking* [online]. Cham: Springer International Publishing, 2018, 2018-05-19, s. 1-26 [cit. 2022-11-25]. ISBN 978-3-319-10574-1. Dostupné z: doi:10.1007/978-3-319-10575-8_1

[26] CHA, Sooyoung, Seongjoon HONG, Junhee LEE a Hakjoo OH. Automatically generating search heuristics for concolic testing. In: *Proceedings of the 40th International Conference on Software Engineering* [online]. New York, NY, USA: ACM, 2018, 2018-05-27, s. 1244-1254 [cit. 2023-01-03]. ISBN 9781450356381. Dostupné z: doi:10.1145/3180155.3180166

[27] CONTIERI, Maximiliano. NULL: The Billion Dollar Mistake. *HACKERNOON* [online]. Edwards, Colorado, USA: Hacker Noon, 2019, 14.4. 2020 [cit. 2022-12-27]. Dostupné z: `https://hackernoon.com/null-the-billion-dollar-mistake-8t5z32d6`

[28] DAKA, Ermira, José CAMPOS, Gordon FRASER, Jonathan DORN a Westley WEIMER. Modeling readability to improve unit tests. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* [online]. New York, NY, USA: ACM, 2015, 2015-08-30, s. 107-118 [cit. 2022-03-05]. ISBN 9781450336758. Dostupné z: doi:10.1145/2786805.2786838

[29] DAVENPORT, James H. a Joos HEINTZ. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation* [online]. 1988, **5**(1-2), 29-35 [cit. 2022-12-11]. ISSN 07477171. Dostupné z: doi:10.1016/S0747-7171(88)80004-X

[30] DE MOURA, Leonardo a Nikolaj BJØRNER. Z3: An Efficient SMT Solver. In: RAMAKRISHNAN, C. R. a Jakob REHOF, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, 2008, s. 337-340 [cit. 2022-11-11]. Lecture Notes in Computer Science. ISBN 978-3-540-78799-0. Dostupné z: doi:10.1007/978-3-540-78800-3_24

[31] DENG, Xianghua, Jooyong LEE a ROBBY. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)* [online]. IEEE, 2006, 2006, s. 157-166 [cit. 2021-10-12]. ISBN 0-7695-2579-2. Dostupné z: doi:10.1109/ASE.2006.26

[32] DHOK, Monika, Murali Krishna RAMANATHAN a Nishant SINHA. Type-aware concolic testing of JavaScript programs. In: *Proceedings of the 38th International Conference on Software Engineering* [online]. New York, NY, USA: ACM, 2016, 2016-05-14, s. 168-179 [cit. 2022-12-22]. ISBN 9781450339001. Dostupné z: doi:10.1145/2884781.2884859

[33] DJIKSTRA, Edsger W. Notes on Structured Programming. In: DJIKSTRA, Edsger W. *Structured Programming*. GBR: Academic Press, 1972, s. 1 - 89. ISBN 0122005503.

[34] ENDERTON, Herbert B. *A mathematical introduction to logic*. Second edition. San Diego: Harcourt/Academic Press, [2001]. ISBN 9780122384523.

[35] FLOYD, Robert W. Assigning meaning to programs. *Proceedings of Symposium on Applied Mathematics*. 1967, (19), 19-31.

[36] FORTZ, Sophie, Fred MESNARD, Etienne PAYET, Gilles PERROUIN, Wim VANHOOF a Germán VIDAL. An SMT-Based Concolic Testing Tool for Logic Programs. In: NAKANO, Keisuke a Konstantinos SAGONAS, ed. *Functional and Logic Programming* [online]. Cham: Springer International Publishing, 2020, 2020-09-02, s. 215-219 [cit. 2021-10-10]. Lecture Notes in Computer Science. ISBN 978-3-030-59024-6. Dostupné z: doi:10.1007/978-3-030-59025-3_13

[37] FOWLER, Martin. Mocks Aren't Stubs. *MartinFowler* [online]. Chicago: Thoughtworks, 1996, 2007 [cit. 2022-11-04]. Dostupné z: `https://martinfowler.com/articles/mocksArentStubs.html`

[38] FRASER, Gordon a Andrea ARCURI. EvoSuite. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11* [online]. New York, New York, USA: ACM Press, 2011, 2011, 416- [cit. 2022-12-28]. ISBN 9781450304436. Dostupné z: doi:10.1145/2025113.2025179

[39] GLASS, R.L. Persistent Software Errors. *IEEE Transactions on Software Engineering* [online]. 1981, **SE-7**(2), 162-168 [cit. 2022-12-21]. ISSN 0098-5589. Dostupné z: doi:10.1109/TSE.1981.230831

[40] GODEFROID, Patrice, Nils KLARLUND a Koushik SEN. DART. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05* [online]. New York, New York, USA: ACM Press, 2005, 2005, 213- [cit. 2022-01-12]. ISBN 1595930566. Dostupné z: doi:10.1145/1065010.1065036

[41] GODEFROID, Patrice. Compositional dynamic test generation. In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '07* [online]. New York, New York, USA: ACM Press, 2007, 2007, 47- [cit. 2022-03-05]. ISBN 1595935754. Dostupné z: doi:10.1145/1190216.1190226

[42] GODEFROID, Patrice, Michael Y. LEVIN a David MOLNAR. Automated Whitebox Fuzz Testing. In: *NDSS*. 2008, s. 151-166.

[43] GODEFROID, Patrice, Michael Y. LEVIN a David MOLNAR. SAGE. *Communications of the ACM* [online]. 2012, **55**(3), 40-44 [cit. 2022-11-30]. ISSN 0001-0782. Dostupné z: doi:10.1145/2093548.2093564

[44] GOLD, Robert. Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science* [online]. 2010, **20**(4), 739-749 [cit. 2022-10-14]. ISSN 1641-876X. Dostupné z: doi:10.2478/v10006-010-0056-9

[45] GRIESKAMP, Wolfgang, Nikolai TILLMANN a Wolfram SCHULTE. XRT— Exploring Runtime for .NET Architecture and Applications. *Electronic Notes in Theoretical Computer Science* [online]. 2006, **144**(3), 3-26 [cit. 2022-03-05]. ISSN 15710661. Dostupné z: doi:10.1016/j.entcs.2006.01.002

[46] HAYHURST, Kelly J., Dan S. VEERHUSEN, John j. CHILENSKI a Leanna K. RIERSON. *A Practical Tutorial on Modified Condition/Decision Coverage*. NASA Langley Technical Report Server, 2001.

[47] HE, Jingxuan, Gishor SIVANRUPAN, Petar TSANKOV a Martin VECHEV. Learning to Explore Paths for Symbolic Execution. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* [online]. New York, NY, USA: ACM, 2021, 2021-11-12, s. 2526-2540 [cit. 2022-11-08]. ISBN 9781450384544. Dostupné z: doi:10.1145/3460120.3484813

[48] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM* [online]. 1969, **12**(10), 576-580 [cit. 2022-11-07]. ISSN 0001-0782. Dostupné z: doi:10.1145/363235.363259

[49] HOLZMANN, Gerard J. Explicit-State Model Checking. In: CLARKE, Edmund M., Thomas A. HENZINGER, Helmut VEITH a Roderick BLOEM, ed. *Handbook of Model Checking* [online]. Cham: Springer International Publishing, 2018, 2018-05-19, s. 153-171 [cit. 2022-11-25]. ISBN 978-3-319-10574-1. Dostupné z: doi:10.1007/978-3-319-10575-8_5

[50] HOWAR, Falk, Dimitra GIANNAKOPOULOU a Zvonimir RAKAMARIĆ. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis* [online]. New York, NY, USA: ACM, 2013, 2013-07-15, s. 268-279 [cit. 2022-12-16]. ISBN 9781450321594. Dostupné z: doi:10.1145/2483760.2483783

[51] CHA, Sooyoung, Seongjoon HONG, Jiseong BAK, Jingyoung KIM, Junhee LEE a Hakjoo OH. Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics. *IEEE Transactions on Software Engineering* [online]. 2022, **48**(9), 3640-3663 [cit. 2022-10-13]. ISSN 0098-5589. Dostupné z: doi:10.1109/TSE.2021.3101870

[52] CHIN, Wei-Ngan, Cristina DAVID, Huu Hai NGUYEN a Shengchao QIN. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming* [online]. 2012, **77**(9), 1006-1036 [cit. 2022-03-06]. ISSN 01676423. Dostupné z: doi:10.1016/j.scico.2010.07.004

[53] JAFFAR, Joxan, Vijayaraghavan MURALI a Jorge A. NAVAS. Boosting concolic testing via interpolation. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* [online]. New York, New York, USA: ACM Press, 2013, 2013, 48- [cit. 2021-10-10]. ISBN 9781450322379. Dostupné z: doi:10.1145/2491411.2491425

[54] KHURSHID, Sarfraz, Corina S. PĂSĂREANU a Willem VISSER. Generalized Symbolic Execution for Model Checking and Testing. In: GARAVEL, Hubert a John HATCLIFF, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, 2003-2-28, s. 553-568 [cit. 2022-03-07]. Lecture Notes in Computer Science. ISBN 978-3-540-00898-9. Dostupné z: doi:10.1007/3-540-36577-X_40

[55] KING, James C. Symbolic execution and program testing. *Communications of the ACM* [online]. 1976, **19**(7), 385-394 [cit. 2022-01-12]. ISSN 0001-0782. Dostupné z: doi:10.1145/360248.360252

[56] KUHN, Richard D., Raghu N. KACKER a Yu LEI. *Introduction to Combinatorial testing*. Miami: CRC Press, 2013. ISBN 978-1-4665-5230-2.

[57] LE, Quang Loc, Jun SUN a Wei-Ngan CHIN. Satisfiability Modulo Heap-Based Programs. In: CHAUDHURI, Swarat a Azadeh FARZAN, ed. *Computer Aided Verification* [online]. Cham: Springer International Publishing, 2016, 2016-07-13, s. 382-404 [cit. 2022-03-06]. Lecture Notes in Computer Science. ISBN 978-3-319-41527-7. Dostupné z: doi:10.1007/978-3-319-41528-4_21

[58] LE, Quang Loc, Makoto TATSUTA, Jun SUN a Wei-Ngan CHIN. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In: MAJUMDAR, Rupak a Viktor KUNČAK, ed. *Computer Aided Verification* [online]. Cham: Springer International Publishing, 2017, 2017-07-13, s. 495-517 [cit. 2022-03-06]. Lecture Notes in Computer Science. ISBN 978-3-319-63389-3. Dostupné z: doi:10.1007/978-3-319-63390-9_26

[59] LI, Hongbo, Zizhong CHEN a Rajiv GUPTA. Efficient concolic testing of MPI applications. In: *Proceedings of the 28th International Conference on Compiler Construction - CC 2019* [online]. New York, New York, USA: ACM Press, 2019, 2019, s. 193-204 [cit. 2021-10-10]. ISBN 9781450362771. Dostupné z: doi:10.1145/3302516.3307353

[60] LI, Hongbo, Sihuan LI, Zachary BENAVIDES, Zizhong CHEN a Rajiv GUPTA. COMPI: Concolic Testing for MPI Applications. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* [online]. IEEE, 2018, 2018, s. 865-874 [cit. 2022-11-30]. ISBN 978-1-5386-4368-6. Dostupné z: doi:10.1109/IPDPS.2018.00096

[61] LIU, Dongge, Gidon ERNST, Toby MURRAY a Benjamin I. P. RUBINSTEIN. Legion. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* [online]. New York, NY, USA: ACM, 2020, 2020-12-21, s. 54-65 [cit. 2021-10-10]. ISBN 9781450367684. Dostupné z: doi:10.1145/3324884.3416629

[62] LUCKOW, Kasper, Marko DIMJAŠEVIĆ, Dimitra GIANNAKOPOULOU, Falk HOWAR, Malte ISBERNER, Temesghen KAHSAI, Zvonimir RAKAMARIĆ a Vishwanath RAMAN. JDart: A Dynamic Symbolic Analysis Framework. In: CHECHIK, Marsha a Jean-François RASKIN, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, 2016-04-09, s. 442-459 [cit. 2022-03-06]. Lecture Notes in Computer Science. ISBN 978-3-662-49673-2. Dostupné z: doi:10.1007/978-3-662-49674-9_26

[63] MAJUMDAR, Rupak a Koushik SEN. Hybrid Concolic Testing. In: *29th International Conference on Software Engineering (ICSE'07)* [online]. IEEE, 2007, 2007, s. 416-426 [cit. 2021-10-10]. ISBN 0-7695-2828-7. Dostupné z: doi:10.1109/ICSE.2007.41

[64] MARICK, Brian. How to Misuse Code Coverage. In: *Exampler Consulting* [online]. Illinois, USA: Brian Marick, 2014, 1997, s. 1-13 [cit. 2022-12-21]. Dostupné z: `http://www.exampler.com/testing-com/writings/coverage.pdf`

[65] MARQUES, Filipe, José Fragoso SANTOS a Nuno SANTOS. Concolic Execution for WebAssembly (Artifact). *Dagstuhl Artifacts Series*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, **8**(2), 20:1-20:3. ISSN 2509-8195. Dostupné z: doi:10.4230/DARTS.8.2.20

[66] MESZAROS, Gerard. *XUnit test patterns: refactoring test code*. Boston, MA: Pearson Education, 2007. ISBN 978-0-13-149505-0.

[67] MYERS, Glenford J., Tom BADGETT a Corey SANDLER. *The art of software testing*. 3rd ed. Hoboken, New Jersey: Wiley, c2012. ISBN 978-1-118-03196-4.

[68] NAVARRO PÉREZ, Juan Antonio a Andrey RYBALCHENKO. Separation Logic Modulo Theories. In: SHAN, Chung-chieh, ed. *Programming Languages and Systems* [online]. Cham: Springer International Publishing, 2013, 2013, s. 90-106 [cit. 2022-04-13]. Lecture Notes in Computer Science. ISBN 978-3-319-03541-3. Dostupné z: doi:10.1007/978-3-319-03542-0_7

[69] NGUYEN, Viet Yen. *Optimising Techniques for Model Checkers*. Enschede, 2007. Master's Thesis. University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, Formal Methods and Tools. Vedoucí práce J.P. Katoen.

[70] PALSHIKAR, G.K. Applying formal specifications to real-world software development. *IEEE Software* [online]. 2001, **18**(6), 89-97 [cit. 2022-12-16]. ISSN 07407459. Dostupné z: doi:10.1109/52.965810

[71] PĂSĂREANU, Corina S., Willem VISSER, David BUSHNELL, Jaco GELDENHUYS, Peter MEHLITZ a Neha RUNGTA. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* [online]. 2013, **20**(3), 391-425 [cit. 2022-03-07]. ISSN 0928-8910. Dostupné z: doi:10.1007/s10515-013-0122-2

[72] PELED, Doron. Partial-Order Reduction. In: CLARKE, Edmund M., Thomas A. HENZINGER, Helmut VEITH a Roderick BLOEM, ed. *Handbook of Model Checking* [online]. Cham: Springer International Publishing, 2018, 2018-05-19, s. 173-190 [cit. 2022-11-25]. ISBN 978-3-319-10574-1. Dostupné z: doi:10.1007/978-3-319-10575-8_6

[73] PHAM, Long H., Quang Loc LE, Quoc-Sang PHAN a Jun SUN. Concolic Testing Heap-Manipulating Programs. In: TER BEEK, Maurice H., Annabelle MCIVER a José N. OLIVEIRA, ed. *Formal Methods — The Next 30 Years* [online]. Cham: Springer International Publishing, 2019, 2019-09-23, s. 442-461 [cit. 2021-10-10]. Lecture

Notes in Computer Science. ISBN 978-3-030-30941-1. Dostupné z: doi:10.1007/978-3-030-30942-8_27

[74] PHAM, Long H., Quang Loc LE, Quoc-Sang PHAN, Jun SUN a Shengchao QIN. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In: CHEN, Yu-Fang, Chih-Hong CHENG a Javier ESPARZA, ed. *Automated Technology for Verification and Analysis* [online]. Cham: Springer International Publishing, 2019, 2019-10-21, s. 209-227 [cit. 2022-03-06]. Lecture Notes in Computer Science. ISBN 978-3-030-31783-6. Dostupné z: doi:10.1007/978-3-030-31784-3_12

[75] PISKAC, Ruzica, Thomas WIES a Damien ZUFFEREY. Automating Separation Logic Using SMT. In: SHARYGINA, Natasha a Helmut VEITH, ed. *Computer Aided Verification* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, 2013, s. 773-789 [cit. 2022-03-06]. Lecture Notes in Computer Science. ISBN 978-3-642-39798-1. Dostupné z: doi:10.1007/978-3-642-39799-8_54

[76] REYNOLDS, John C. Intuitionistic reasoning about shared mutable data structure. In: DAVIES, Jim, Bill ROSCOE a Jim WOODCOCK, ed. *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*. 1. London: Red Globe Press London, 2000, s. 303-321. ISBN 0333922301.

[77] REYNOLDS, J.C. Separation logic: a logic for shared mutable data structures. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* [online]. IEEE Comput. Soc, 2002, s. 55-74 [cit. 2022-03-07]. ISBN 0-7695-1483-9. Dostupné z: doi:10.1109/LICS.2002.1029817

[78] ROSNER, Nicolas, Jaco GELDENHUYS, Nazareno AGUIRRE, Willem VISSER a Marcelo FRIAS. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering* [online]. 2015, **41**(7), 1-1 [cit. 2021-10-12]. ISSN 0098-5589. Dostupné z: doi:10.1109/TSE.2015.2389225

[79] RUYS, Theo C. a Niels H.M. Aan DE BRUGH. MMC: the Mono Model Checker. *Electronic Notes in Theoretical Computer Science* [online]. 2007, **190**(1), 149-160 [cit. 2021-09-30]. ISSN 15710661. Dostupné z: doi:10.1016/j.entcs.2007.02.066

[80] SEN, Koushik, Darko MARINOV a Gul AGHA. CUTE. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13* [online]. New York, New York, USA: ACM Press, 2005, 2005, 263- [cit. 2021-10-10]. ISBN 1595930140. Dostupné z: doi:10.1145/1081706.1081750

[81] SEN, Koushik a Gul AGHA. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: BALL, Thomas a Robert B. JONES, ed. *Computer Aided Verification* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 2006, s. 419-423 [cit. 2021-10-10]. Lecture Notes in Computer Science. ISBN 978-3-540-37406-0. Dostupné z: doi:10.1007/11817963_38

[82] SEN, Koushik. Concolic testing. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07* [online]. New York, New York, USA: ACM Press, 2007, 2007, 571- [cit. 2021-10-10]. ISBN 9781595938824. Dostupné z: doi: 10.1145/1321631.1321746

[83] SEN, Koushik, Swaroop KALASAPUR, Tasneem BRUTCH a Simon GIBBS. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* [online]. New York, New York, USA: ACM Press, 2013, 2013, 488- [cit. 2022-12-22]. ISBN 9781450322379. Dostupné z: doi:10.1145/2491411.2491447

[84] SEO, Hyunmin a Sunghun KIM. How we get there: a context-guided search strategy in concolic testing. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* [online]. New York, NY, USA: ACM, 2014, 2014-11-11, s. 413-424 [cit. 2022-10-13]. ISBN 9781450330565. Dostupné z: doi:10.1145/2635868.2635872

[85] SCHWARTZ, Edward J., Thanassis AVGERINOS a David BRUMLEY. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In: *2010 IEEE Symposium on Security and Privacy* [online]. IEEE, 2010, 2010, s. 317-331 [cit. 2022-10-11]. ISBN 978-1-4244-6894-2. Dostupné z: doi:10.1109/SP.2010.26

[86] SOMMERVILLE, Ian. *Software engineering*. 9th ed. Boston: Pearson, 2011. ISBN 978-0-13-703515-1.

[87] STEPHENS, Nick, John GROSEN, Christopher SALLS, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: *Proceedings 2016 Network and Distributed System Security Symposium* [online]. Reston, VA: Internet Society, 2016, 2016, - [cit. 2021-10-10]. ISBN 1-891562-41-X. Dostupné z: doi:10.14722/ndss.2016.23368

[88] SUN, Youcheng, Min WU, Wenjie RUAN, Xiaowei HUANG, Marta KWIATKOWSKA a Daniel KROENING. Concolic testing for deep neural networks. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* [online]. New York, NY, USA: ACM, 2018, 2018-09-03, s. 109-119 [cit. 2021-10-10]. ISBN 9781450359375. Dostupné z: doi:10.1145/3238147.3238172

[89] TATSUTA, Makoto, Quang Loc LE a Wei-Ngan CHIN. Decision Procedure for Separation Logic with Inductive Definitions and Presburger Arithmetic. In: IGARASHI, Atsushi, ed. *Programming Languages and Systems* [online]. Cham: Springer International Publishing, 2016, 2016-10-09, s. 423-443 [cit. 2022-03-07]. Lecture Notes in Computer Science. ISBN 978-3-319-47957-6. Dostupné z: doi:10.1007/978-3-319-47958-3_22

[90] THUMMALAPENTA, Suresh, Tao XIE, Nikolai TILLMANN, Jonathan DE HALLEUX a Zhendong SU. Synthesizing method sequences for high-coverage testing. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11* [online]. New York, New York, USA: ACM Press, 2011, 2011, 189- [cit. 2022-03-05]. ISBN 9781450309400. Dostupné z: doi:10.1145/2048066.2048083

[91] TILLMANN, Nikolai a Jonathan DE HALLEUX. Pex—White Box Test Generation for .NET. In: BECKERT, Bernhard a Reiner HÄHNLE, ed. *Tests and Proofs* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, 2008, s. 134-153 [cit. 2021-09-24]. Lecture Notes in Computer Science. ISBN 978-3-540-79123-2. Dostupné z: doi:10.1007/978-3-540-79124-9_10

[92] TORKAR, Richard. *Towards Automated Software Testing: Techniques, Classifications and Frameworks*. Karlskrona, Sweden: Blekinge Institute of Technology, 2006. ISBN 91-7295-089-7.

[93] TRTÍK, Marek. *Symbolic Execution and Program Loops*. Brno, 2013. Disertace. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Anotnín Kučera.

[94] MEUTER, Wolfgang De a Coen De ROOVER, VANDERCAMMEN, Maarten a Laurent CHRISTOPHE, ed. Concolic Testing of Full-Stack JavaScript Applications. In: GOUSIOS, Georgios a Joseph HEJDERUP. *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop, Delft, the Netherlands, December 10th - to - 11th, 2018*. CEUR-WS.org, 2018, s. 38-42.

[95] VARTANOV, Sergey a Audrey DUTCHER. Enzet/symbolic-execution: History of symbolic execution (as well as SAT/SMT solving, fuzzing and taint data tracking). *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008 [cit. 2022-12-27]. Dostupné z: `https://github.com/enzet/symbolic-execution`

[96] VINEET, Nanda. What is Grey Box Testing?. *Online Tutorials Library* [online]. Hyderabad, Telangana, India: Tutorials Point, 2006, 2021 [cit. 2023-01-02]. Dostupné z: https://www.tutorialspoint.com/what-is-grey-box-testing#

[97] VISSER, Willem, Klaus HAVELUND, Guillaume BRAT, SeungJoon PARK a Flavio LERDA. Model Checking Programs. *Automated Software Engineering* [online]. 2003, **10**(2), 203-232 [cit. 2022-12-13]. ISSN 09288910. Dostupné z: doi:10.1023/A:1022920129859

[98] VISSER, Willem, Corina S. PĂSĂREANU a Sarfraz KHURSHID. Test input generation with java PathFinder. In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04* [online]. New York, New York, USA: ACM Press, 2004, 2004, 97- [cit. 2021-10-12]. ISBN 1581138202. Dostupné z: doi:10.1145/1007512.1007526

[99] VOCKE, Ham. The Practical Test Pyramid. *MartinFowler* [online]. Chicago: Thoughtworks, 1996, 26. 2. 2018 [cit. 2022-10-27]. Dostupné z: `https://martinfowler.com/articles/practical-test-pyramid.html`

[100] XIE, Tao, Darko MARINOV, Wolfram SCHULTE a David NOTKIN. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In: HALBWACHS, Nicolas a Lenore D. ZUCK, ed. *Tools and Algorithms for the Construction and Analysis of Systems* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, 2005, s. 365-381 [cit. 2022-03-05]. Lecture Notes in Computer Science. ISBN 978-3-540-25333-4. Dostupné z: doi:10.1007/978-3-540-31980-1_24

[101] *Common Language Infrastructure (CLI)*. Third Edition. Geneva: EIC, 2012.

[102] Diffblue Cover: Create complete JUnit tests with AI - IntelliJ IDEA & Aqua Plugin. *JetBrains: Essential tools for software developers and teams* [online]. Praha: JetBrains, 2000 [cit. 2022-12-29]. Dostupné z: `https://plugins.jetbrains.com/plugin/14946-diffblue-cover--create-complete-junit-tests-with-ai/`

[103] 0xd4d/dnlib: Reads and writes .NET assemblies and modules. *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008 [cit. 2022-12-27]. Dostupné z: `https://github.com/0xd4d/dnlib`

[104] dnWalker-project/dnWalker. *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008, 2022 [cit. 2023-01-05]. Dostupné z: `https://github.com/dnWalker-project/dnWalker`

[105] .NET: Free. Cross-Platform. Open Source. *Microsoft - Cloud, Computers, Apps & Gaming* [online]. Redmond, Washington, USA: Microsoft [cit. 2022-12-27]. Dostupné z: `https://dotnet.microsoft.com/`

[106] FakeItEasy/FakeItEasy: The easy mocking library for .NET. *Github* [online]. San Francisco, California, U.S.: GitHub, 2011 [cit. 2022-12-27]. Dostupné z: `https://fakeiteasy.github.io/`

[107] Overview of Microsoft IntelliTest. *Microsoft Docs* [online]. San Francisko: Microsoft, 22. 12. 2021 [cit. 2022-01-12]. Dostupné z: `https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2019`

[108] Isolator: Typemock. *Typemock* [online]. Typemock, 2006, 2016 [cit. 2022-12-27]. Dostupné z: `https://www.typemock.com/isolator-product-page/`

[109] Mono. *Mono Project* [online]. Redmond: .NET Foundation, 2013 [cit. 2022-01-12]. Dostupné z: `https://www.mono-project.com/`

[110] Moq/moq4: Repo for managing Moq4.x. *Github* [online]. San Francisco, California, U.S.: GitHub, 2008, 2010 [cit. 2022-12-27]. Dostupné z: urlhttps://github.com/moq/moq4

[111] Isolate code under test with Microsoft Fakes. *Microsoft - Cloud, Computers, Apps & Gaming* [online]. Redmond, Washington, USA: Microsoft, 2022 [cit. 2022-12-27]. Dostupné z: `https://learn.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes?view=vs-2022&tabs=csharp`

[112] Remoting Proxy Support for Rhino Mocks. *Ivan Krivyakov - Premature optimization is the root of all evil* [online]. 2002, 2007 [cit. 2022-12-27]. Dostupné z: `https://ikriv.com/dev/dotnet/RhinoMocks`

[113] SL-COMP: Competition of Solvers for Separation Logic. *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008, 2014 [cit. 2022-12-27]. Dostupné z: `https://sl-comp.github.io/`

[114] SMT-COMP. *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008 [cit. 2022-12-27]. Dostupné z: `https://smt-comp.github.io/`

[115] *The Satisfiability Modulo Theories Library (SMT-LIB)* [online]. 2016 [cit. 2022-12-27]. Dostupné z: `https://smtlib.cs.uiowa.edu/`

[116] Xunit/xunit: xUnit.net is a free, open source, community-focuesd unit testing tool for .NET. *GitHub* [online]. San Francisco, California, U.S.: GitHub, 2008 [cit. 2022-12-28]. Dostupné z: https://github.com/xunit/xunit