VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZVOJ NÁSTROJE COMBINE

DEVELOPMENT OF COMBINE TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PAVEL NOVÁČEK**
AUTHOR

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

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

147622

Ústav: Ústav inteligentních systémů (UITS)
Student: **Nováček Pavel, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Verifikace a testování software

Název: **Rozvoj nástroje Combine**
Kategorie: Analýza a testování softwaru
Akademický rok: 2022/23

Zadání:

1. Seznamte se s nástrojem Combine pro tvorbu kombinací hodnot testovacích dat. Nastudujte techniky kombinatorického testování. Nastudujte možnosti a použití SMT řešičů.
2. Analyzujte možnosti úpravy frontendu i backendu nástroje Combine. Navrhněte rozšíření pro generování náhodných dat složitějších typů (např. řetězce s daným omezením) a dalších metod pro generování testovacích dat (např. splňující kritérium pokrytí Base-Choice).
3. Implementujte navržené změny v nové verzi nástroje Combine.
4. Ověřte funkcionalitu pomocí automatizovaných testů. Vytvořte tutoriál k použití aplikace.

Literatura:

- P. Ammann, J. Offutt. 2008. Introduction to Software Testing, Cambridge University Press. ISBN 978-0-511-39330-3.
- Domovská stránka projektu Combine: http://combine.testos.org/

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání

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

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 3.11.2022

Abstrakt

Tato práce se zabývá reimplementací webového nástroje Combine, který pro uživatelem specifikované parametry a specifikovaná omezení vygeneruje použitím algoritmu IPOG kombinatorickou testovací sadu splňující požadované kombinatorické kritérium pokrytí. Práce je řešena v kontextu platformy Testos, která cílí na automatizaci softwarového testování. Cílem této práce je analyzovat aktuální stav nástroje, identifikovat jeho chyby a nedostatky, diskutovat vhodná rozšíření jeho funkcionality a na základě těchto poznatků vytvořit nový návrh a implementaci nástroje Combine. Vytvořené řešení zachovává veškerou funkcionalitu původního nástroje a zároveň ji rozšiřuje na všech úrovních architektury. Nástroj je nejen lépe ovladatelný díky uživatelsky přívětivějšímu webovému rozhraní, ale nabízí i nové možnosti pro generování testovacích sad. Pro jeho implementaci byly zvoleny vhodnější technologie, díky kterým je nástroj přenositelný a generuje testovací sadu značně rychleji než předchozí implementace a dokáže konkurovat existujícím řešením pro generování kombinatorické testovací sady.

Abstract

This thesis deals with the reimplementation of the web tool Combine, which generates combinatorial test sets satisfying user-specified parameters and constraints using the IPOG algorithm. The thesis is a part of Testos platform, which aims at software testing automation. The goal of this thesis is to analyze the current state of the tool, identify its errors and deficiencies, discuss suitable extensions to its functionality, and based on these findings, create a new design and implementation of the Combine tool. The new solution preserves all the functionality of the previous tool and at the same time expands it on all levels of the architecture. The tool is not only more manageable thanks to a more user-friendly web interface but also offers new possibilities for test set generation. More suitable technologies have been chosen for its implementation, making the tool portable and significantly faster in generating test sets compared to the previous implementation, capable of competing with existing solutions for combinatorial test set generation.

Klíčová slova

Combine, Testos, React, .NET 7, C++, SMT Solver, kombinatorické testování, generování testovací sady, algoritmus IPOG, testování T-Wise, Base-Choice algorithm, ekvivalenční dělení vstupní domény, webová aplikace

Keywords

Combine, Testos, React, .NET 7, C++, SMT Solver, combinatorial testing, test set generation, IPOG algorithm, T-Wise testing, Base-Choice, equivalence partitioning of input domain, web application

Citace

NOVÁČEK, Pavel. *Rozvoj nástroje Combine*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Rozvoj nástroje Combine

Prohlášení

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

.....
Pavel Nováček
17. května 2023

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Alešovi Smrčkovi, Ph.D. za jeho čas, odborné vedení, ochotu a cenné rady, které mi věnoval při tvorbě této diplomové práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

S adaptací nových technologií v oblasti vývoje softwaru, jako jsou software založený na komponentách a software orientovaný na služby, se výrazně začalo zvětšovat množství potřebných parametrů softwarových systémů. Především kvůli tomu, že se zvyšuje komplexnost funkcí softwaru a běhová prostředí jsou často komplikovaná a distribuovaná po síti, moderní softwarové systémy musí být vysoce konfigurovatelné, aby byly spustitelné a optimalizované pro běh na několika různých platformách a podporovaly různé scénáře použití. Nicméně, interakce těchto parametrů mohou způsobit selhání systému, a proto by měly být otestovány.

Jednou z možností, jak otestovat interakce parametrů, je prostřednictvím vyčerpávajícího testování, při kterém se testují všechny kombinace všech hodnot všech parametrů. Vzhledem k velkému počtu možných kombinací je vyčerpávající testování značně nepraktické. Navíc, tento přístup je značně neefektivní, jelikož většina kombinací žádnou poruchu nezpůsobí.

Kombinatorické testování představuje praktickou cestu, jak otestovat interakce parametrů a zároveň se vyhnout kombinatorické explozi. Definuje postupy pro systematické vybírání kombinací z kombinatorického prostoru tak, aby vytvořená testovací sada pokrývala specifikované kritérium pokrytí.

Cílem této práce je reimplementace nástroje Combine, který generuje testovací sadu pokrývající zadané kombinatorické kritérium pokrytí použitím algoritmu IPOG. Nástroj implementoval Radim Červinka v rámci své bakalářské práce [24]. Reimplementace nástroje je nutná především z důvodu, že současné řešení slouží především jako *Proof of Concept* algoritmu IPOG, a proto je prototypové, obsahuje spoustu chyb a nedostatků a není použitelné pro reálné problémy. Dalším důvodem je plánované použití nástroje v jiných aktuálně vyvíjených projektech, konkrétně v evropském projekt *VALU3S*. Nástroj, ve kterém se plánuje využití nástroje Combine, se nazývá *netloiter* (aktuálně je ve vývoji). Combine se bude využívat pro minimalizaci počtu experimentů, resp. kombinací různých parametrů tzv. injektovaných vad (angl. *injected faults*) jako například doba pozdržení paketu, ztrátovost paketu, výběr doby injekce vad s ohledem na dobu trvání netflow.

Nová implementace nástroje bude využívat vhodné technologie, bude efektivní, multiplatformní a bude otestovaná jednotkovými i integračními testy. Nástroj bude obsahovat všechny funkcionality původní implementace a zároveň ji značně rozšíří na všech úrovních architektury. Výsledné řešení bude možné používat nejen skrze webové rozhraní, ale také jako službu skrze veřejné REST API.

Nástroj patří do oblasti zabývající se problematikou kombinatorického testování. Kapitola 2 proto popisuje teoretické základy kombinatorického testování, definuje potřebný matematický aparát, pojednává o různých přístupech kombinatorického testování a představuje vybrané existující algoritmy a programy z této oblasti. Kapitola 3 se zabývá podrobnou analýzou aktuální implementace nástroje Combine a identifikuje jeho chyby a nedostatky. Kapitola 4 pak již přechází k návrhu nové implementace nástroje Combine. Z počátku rozebírá požadavky na výsledný nástroj a popisuje jeho obecný návrh. Na obecný návrh navazuje detailní popis jednotlivých komponent nástroje, a to především jejich rozhraní, architektury, hlavních funkcionalit a průběh komunikace s ostatními částmi nástroje.

Kapitola 2

Teorie kombinatorického testování

Tato kapitola představuje teoretické základy kombinatorického testování a definuje potřebný matematický aparát. Následně přechází k přehledu technik pro generování kombinatorické testovací sady, vybraným algoritmům a existujícím programovým řešením.

2.1 Kombinatorické testování

Kombinatorické testování (angl. *Combinatorial Testing*, dále jen CT), také nazývané *Kombinatorické testování interakcí* (angl. *Combinatorial Interaction Testing*), je typ dynamického testování[1] (s původem ve statistické oblasti *Navrhování experimentů* (angl. *Design of Experiments*)[2]), který poskytuje praktický způsob, s dobrým kompromisem mezi náklady na testování a jeho účinností, pro detekci selhání způsobených interakcí parametrů v testovaném softwaru. Tato selhání se označují jako *interakční selhání* (angl. *interaction failures*), protože se projeví pouze tehdy, když dvě nebo více hodnot vstupních parametrů spolu interagují a způsobí, že program dosáhne nesprávného výsledku. [12]

Přestože domény hodnot vstupních parametrů mohou být obsáhlé, většina jejich kombinací nezpůsobí selhání SUT, proto je *vyčerpávající testování*[3] (angl. *exhaustive testing*) neefektivní a nepraktické kvůli příliš velkému *kombinatorickému prostoru* (angl. *combinatorial space*). CT se vyhýbá problému *kombinatorické exploze*[4] (angl. *combinatorial explosion*) vyčerpávajícího testování výběrem testovacích případů prostřednictvím specifického vzorkovacího mechanismu tak, aby se systematicky pokryly kombinace hodnot parametrů malou testovací sadou, kterou lze relativně snadno spravovat a spustit. [16]

CT je všestranná metodika, jež je užitečná v široké škále testovacích situací. Nejzákladnější použití CT je pro testování různých konfigurací systému nebo různých vstupů do systému od uživatele nebo z jiného systému. CT lze také použít pro testování databází a stavových modelů.

[1] Ověřování vlastností SW na základě provádění kódu.

[2] Statistická metodologie pro provádění řízených experimentů, ve kterých je systém spouštěn s pečlivě vybranými hodnotami parametrů, které umožňují výpočet užitečných statistických informací o vztahu mezi vstupními parametry ovlivňující systém a sledovaným výstupním parametrem systému.

[3] Testování všech kombinací všech hodnot všech vstupních parametrů.

[4] Složitost daného problému silně vzrůstá spolu s tím, jak se vzhledem k rostoucímu vstupu velice rychle rozšiřuje kombinatorické jádro problému, typicky počet kombinací, které by mohly být řešením.

Podle [4] bylo CT využito v těchto situacích:

- Testování souběžných systémů,
- testování webových aplikací,
- bezpečnostní testování implementací řízení přístupu,
- navigace dynamických webových struktur,
- optimalizace modelů simulace diskrétních událostí,
- analýza pokrytí stavového prostoru systému,
- detekce uváznutí pro různé konfigurace sítě,
- detekce zranitelnosti při přetečení vyrovnávací paměti,
- testování sekvence událostí a
- prioritizace testovacích sad pro webové aplikace založené na uživatelských sezeních.

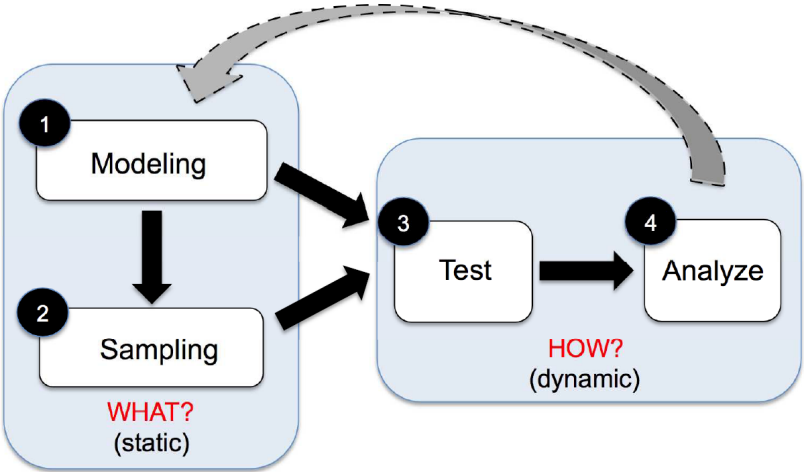
Problematika CT zahrnuje několik výzev: [4]

1. Modelování testovacího prostoru včetně specifikace testovacích parametrů, nastavení testu a jejich omezení.
2. Efektivní vytváření τ-cestných (angl. τ-*way*) testovacích sad, zejména sad podporujících specifikování omezení.
3. Určení očekávaného chování systému pro každý možný testovací případ a zkontrolování, zda skutečné chování souhlasí s očekávaným chováním.
4. Identifikace kombinací hodnot způsobujících selhání na základě procházejících/selhávajících výsledků CT.
5. Integrace CT do stávajících infrastruktur pro testování.

CT lze rozdělit do čtyř hlavních fází, jak zobrazuje obrázek 2.1. První dvě fáze, **modelování** a **vzorkování**, se typicky zabývají tím, *co se testuje*, tedy jaké jsou charakteristiky SUT a proti jakým vstupům by mělo být SUT testováno. V rámci první fáze se modelují aspekty SUT, jako jsou vstupy, konfigurace a sekvence operací. Vzorkování označuje proces nebo algoritmus, kterým se určí prostředky k pokrytí modelu vygenerovaného v první fázi, např. všechny páry všech parametrů atd. Tyto fáze jsou obvykle statické, tzn. že se provedou pouze jednou na začátku CT (ačkoli mohou mít zpětnou vazbu z pozdějších fází). [29]

Druhé dvě fáze, **testování** a **analýza**, se obvykle zabývají tím, *jak se testuje* — provádění testů a následně zkoumání jejich výsledků. Tyto fáze mají větší tendenci být řízeny procesem a vyvíjejí se po delší časové období. Testeři nakonec analyzují výsledky testů přinejmenším alespoň pro pochopení, které testovací případy prošly a které selhaly. V některých případech mohou testeři využít testovací a analytickou fázi k poskytnutí zpětné vazby ke zlepšení a upřesnění pozdějšího modelování a vzorkování. [29]

Podrobněji jsou rozebrány fáze ze statické části, protože oblastí dynamickou se tato práce nezabývá.



Obrázek 2.1: Fáze kombinatorického testování (převzato z [29]).

2.2 Modelování SUT a vzorkovací mechanismy

Tato podkapitola pojednává o prvních dvou fázích CT. Nejdříve definuje model SUT a diskutuje problémy při modelování. Následně se zabývá různými vzorkovacími mechanismy, které pro vstupní prostor modelu vygenerují nejmenší možnou množinu konfigurací, kterou jsou schopny najít, a která splňuje specifikované kritérium pokrytí.

2.2.1 Model SUT

Prvním krokem CT je modelování SUT a jeho *vstupního prostoru* (angl. *input space*). Za vstup se považuje vše, co může ovlivnit chování systému a je možné nad tím držet kontrolu, např. konfigurační parametry, interní nebo externí události, vstupy od uživatele atd.

Definice 2.2.1 (SUT model [16])**.** Model SUT s n vstupními parametry je čtveřice $Model_{SUT}(P, V, R, C)$ kde:

- P je neprázdná konečná množina parametrů SUT, $P = \{c_1, c_2, \ldots, c_n\}$.
- V je neprázdná konečná množina obsahující množiny hodnot, $V = \{V_1, V_2, \ldots, V_n\}$, kde V_i je množina hodnot pro parametr c_i $(i = 1, 2, \ldots, n)$, kterých může parametr nabývat.
- $R \subseteq 2^P$ je množina interakcí mezi parametry. Například pro množinu $R = \{\{c_1\}, \{c_1, c_2\}\}$ prvek $\{c_1\} \in R$ vyjadřuje, že všechny hodnoty V_1 parametru c_1 mohou ovlivnit SUT a mohou způsobit jeho selhání. $\{c_1, c_2\} \in R$ vyjadřuje, že existují interakce mezi parametry c_1 a c_2, což znamená, že všechny dvojice hodnot z $V_1 \times V_2$ můžou způsobit selhání SUT. R jistým způsobem vyjadřuje požadavky na pokrytí pro CT specifikující, které kombinace by měly být pokryty testováním.
- C je množina omezení, jež existují mezi hodnotami různých parametrů. Tyto hodnoty pak mohou být použity na vyřazení kombinací, které nejsou smysluplné ze sémantiky domény.

Jak naznačuje definice 2.2.1, v rámci modelování SUT je nutné vyřešit následují problémy: [16]

1. Identifikace parametrů ovlivňujících SUT – parametry mohou představovat konfigurační parametry, vstupy od uživatele, funkcionality SUT, rozhraní nebo GUI, komponenty nebo objekty.

2. Zvolení hodnot pro každý parametr – pokud doména parametru obsahuje velké množství diskrétních hodnot či dokonce je spojitá, je nutné z ní vybrat pouze několik typických hodnot, které budou použity při vytváření testovacích případů. *Ekvivalenční dělení vstupní domény* (angl. *Equivalence partitioning*), *Analýza hraničních hodnot* (angl. *Boundary value analysis*) a *Výběr primárních elementů* (angl. *Primary element selection*) jsou obvykle používané metody pro tento účel. Výběrem typických reprezentativních hodnot ze třídy ekvivalence doplněných některými hraničními a dalšími důležitými či relevantními hodnotami je možné množiny hodnot $V_i (1 \leq i \leq n)$ v SUT udržet na malých velikostech (pouze s několika reprezentativními hodnotami).

3. Identifikace interakcí a zařazení parametrů podle jejich interakcí:

   (a) Parametry, které nebudou interagovat s žádnými jinými parametry.

   (b) Parametry, mezi kterými budou silné vzájemné interakce.

   (c) Interakce, které existují mezi malým počtem parametrů.

4. Identifikace omezení mezi parametry a jejich hodnotami – často některé hodnoty jednoho parametru kolidují s některými hodnotami jiného parametru. Například parametr pro internetový prohlížeč nemůže nabývat hodnoty `Internet Explorer` pokud parametr pro operační systém nabývá hodnoty `Linux`.

2.2.2 Pole pokrytí a jeho kritéria pokrytí

Vstupní prostor modelu SUT je určen definovanými množinami hodnot parametrů. Přístupy CT systematicky vzorkují tento vstupní prostor a vytvářejí sadu konfigurací (tj. τ-tice představující testovací případy), které budou testovány v další fázi. Vzorkování se provádí výpočtem efektivního kombinatorického objektu zvaného *pole pokrytí*, který je konstruován podle specifikovaného kritéria vzorkování/pokrytí. Pro SUT je pole pokrytí matice $N \times \tau$, ve které jeden řádek je τ-cestná kombinace hodnot τ parametrů, přičemž každá validní kombinace hodnot parametrů SUT se musí objevit alespoň v jednom z N řádků.

Definice 2.2.2 (Testovací případ [27])**.** Nechť P je množina n parametrů $P = \{p_1, p_2, \ldots, p_n\}$ a $V = \{V_1, V_2, \ldots, V_n\}$, kde každý prvek V_i je konečná množina diskrétních hodnot a každý parametr p_i může nabývat hodnot z V_i pro všechny $1 \leq i \leq n$. Pak n-tice $(x_1, x_2, \ldots, x_n)$ je *testovací případ* t, kde každý prvek x_i je validní instance p_i, tj. $x_i \in V_i$ pro $1 \leq i \leq n$.

Definice 2.2.3 (τ-cestná kombinace (angl. *τ-way combination*) [27])**.** Nechť $t = (x_1, x_2, \ldots, x_n)$ je testovací případ nad $P = \{p_1, p_2, \ldots, p_n\}$ parametry. τ-cestná kombinace je kombinace kteréhokoliv z τ parametrů testovacího případu t, kde $1 \leq \tau \leq n$. *τ-cestná kombinace* bude používat následující notaci: $(-, x_{k_1}, \ldots, x_{k_\tau}, \ldots)$ kde každý z τ parametrů má fixní hodnoty a ostatním parametrům jsou přiřazeny jakékoliv validní hodnoty z jejich domény reprezentované jako "$-$".

Vyčerpávající testování dle definice pokrývá všechny n-cesté kombinace. Takové testovací sady jsou však obvykle neúnosně velké. Místo toho CT poskytuje systematický přístup k výběru podmnožiny všech možných vstupů s cílem pokrýt každou τ-cestnou kombinaci alespoň jednou. Koncepce CT vychází ze skutečnosti, že pokud se na žádném selhání nepodílí více než τ parametrů, pak pokrytí všech k-cestných kombinací ($k \leq \tau$) je v podstatě ekvivalentní vyčerpávajícímu testování. K reprezentaci testovacích sad CT se používá matematický objekt nazývající se τ-cestné pole pokrytí.

Definice 2.2.4 (Pole pokrytí (angl. *Covering array*) [27])**.** *τ-cestné pole pokrytí* je $N \times n$ pole, které splňuje následující vlastnosti:

1. Každý sloupec i ($1 \leq i \leq n$) obsahuje pouze elementy z množiny V_i.
2. Řádky každého $N \times \tau$ podpole pokrývají všech $|V_{k_1}| \times |V_{k_2}| \times \cdots \times |V_{k_\tau}|$ kombinací τ sloupců alespoň jednou, kde $1 \leq \tau \leq n$ a $1 \leq k_1 < \cdots < k_\tau \leq n$.

τ-cestné pole pokrytí se často vyjadřuje zápisem $CA\left(N; \tau, v_1^{k_1}, v_2^{k_2}, \ldots, v_m^{k_m}\right)$, kde $v_i^{k_i}$ označuje k_i parametrů se stejným počtem v_i hodnot, přičemž platí $\sum k_i = n$. Hodnota τ se označuje jako *síla pokrytí* (angl. *covering strength*).

Testování pomocí testovací sady τ-cestného pole pokrytí se nazývá *τ-cestné testování* (angl. *τ-way testing*). τ-cestné testování je druh CT, který vyžaduje, aby každá kombinace hodnot τ parametrů byla testována alespoň jednou.

Některé hodnoty τ mají specifické označení: [16]

- $\tau = 1$ – kombinační strategie *Each Choice*, která vyžaduje, aby se každá hodnota každého parametru vyskytla v poli pokrytí alespoň jednou.
- $\tau = 2$ – testování párů (*Pair-Wise*) vyžaduje pokrytí takových kombinací, kde každá hodnota každého parametru musí být zkombinována s každou hodnotou ostatních parametrů.
- $\tau = n$ – kombinační strategie *All Combinations*, která vyžaduje, aby ze všech parametrů byly použity všechny kombinace jejich hodnot.

2.2.3 Vzorkovací metoda Base Choice

Metoda *Base Choice*, také označovaná jako *One Factor One Time*, začíná identifikací jednoho bázového testovacího případu $t = (v_1, v_2, \ldots, v_n)$. *Bázový testovací případ* (angl. *base test case*) lze určit na základě jakéhokoli předem definovaného kritéria, např. nejjednodušší, nejmenší, první atd. Navrhované kritérium je zvolit testovací případ s nejpravděpodobnějšími hodnotami z pohledu uživatele. [3]

Z bázového testovacího případu jsou vytvářeny nové testovací případy tak, že se nahradí hodnota v_i ostatními hodnotami parametru c_i jedna po druhé, přičemž hodnoty ostatních parametrů zůstanou nezměněny. Pro každou další hodnotu vzniká nový testovací případ. Jakmile jsou použity všechny hodnoty aktuálního parametru, je mu nazpět přiřazena bázová hodnota v_i a celý postup se opakuje pro další parametr v pořadí.

Base Choice splňuje 1-cestné pokrytí (Each Choice), protože každá hodnota každého parametru je zahrnuta v nějakém testovacím případě.

2.2.4 Přehled vzorkovacích přístupů

Různé vzorkovací přístupy se liší v používaných modelech a kritériích, které se snaží pokrýt. Tato podkapitola čerpá z [29].

Testování interakcí s proměnlivou silou (angl. *Variable-Strength Interaction Testing*) je praktičtější a v některých případech flexibilnější přístup ke zkoumání interakčního pokrytí než τ-cestné testování s pevně danou silou pokrytí τ. Cílem této metody je pokrýt kombinace hodnot parametrů pouze v rámci skupin, které obsahují navzájem silně interagující parametry, jelikož síly interakcí mezi parametry často nejsou uniformní a některé skupiny parametrů mohou způsobit poruchu s větší pravděpodobností. Metoda využívá *pole pokrytí s proměnlivou silou* (angl. *Variable-Strength Covering Arrays*), jež umožňuje různé síly pokrytí pro různé skupiny parametrů (při zachování τ-cestného pokrytí celého systému).

Pole lokalizující chyby (angl. *Error-Locating Arrays*) pomáhají vývojářům detekovat a izolovat chybné interakce s pomocí poli pokrytí, jež jsou vytvořena na základě kritéria pokrytí, které systematicky zavádí redundanci do testovací sady, jež umožňuje izolovat konkrétní kombinaci hodnot parametrů vedoucí k selhání.

Pole pokrytí zohledňující testovací případy (angl. *Test Case-Aware Covering Arrays*) obsahují množinu konfigurací SUT, z nichž každá je asociována se sadou testovacích případů, které budou v rámci dané konfigurace provedeny. Žádný testovací případ nesmí být asociován s takovou konfigurací, která by porušovala omezení specifická pro tento testovací případ, a zároveň každá validní τ-cestná kombinace daného testovacího případu musí být součástí alespoň jedné množiny konfigurací, s níž je testovací případ asociován.

Pole pokrytí zohledňující cenu (angl. *Cost-Aware Covering Arrays*) řeší situaci, kdy cena některých konfigurací je vyšší než cena jiných konfigurací, například konfigurace vyžadující instalaci nebo kompilaci softwaru mají výrazně delší dobu provedení. Pokud se bere v potaz pouze závislost cen na jednotlivých konfiguracích, pak snížením počtu konfigurací nebo testovacích běhů nemusí nutně dojít ke snížení celkové ceny testování. Pole pokrytí zohledňující cenu bere v úvahu odhad skutečné ceny na testování při vytváření interakčních testovacích sad. Konkrétně τ-cestné pole pokrytí zohledňující cenu je tradiční τ-cestné pole pokrytí, které minimalizuje danou funkci ceny.

Pole pokrytí sekvencí (angl. *Sequence-Covering Arrays*) jsou vytvořeny tak, aby pokryly sekvence událostí, které jsou implicitně specifikovány daným kritériem pokrytí v „minimálním“ počtu sekvencí událostí s pevnou délkou. V systémech řízených událostmi sekvence předchozích událostí určuje, jakým způsobem bude zpracována aktuální událost. Různá pořadí stejné množiny událostí pak mohou odhalit různá selhání.

2.3 Kombinační strategie pro pole pokrytí

Kombinační strategie (angl. *Combination Strategies*) vybírají testovací případy kombinováním hodnot různých parametrů testovaného objektu na základě nějaké kombinatorické strategie. Zahrnuje čtyři prvky: [16]

1. Určení síly pokrytí,

2. seeding k přiřazení některých konkrétních testovacích případů předem,
3. omezení, která je třeba vzít v úvahu při generování testovací sady,
4. metodu pro generování testovací sady.

2.3.1 Volba síly pokrytí

Pro volbu vhodné síly pokrytí je nutná znalost a zkušenosti s daným SUT. V případech, kdy jsou dostupné znalosti o SUT značně omezené, je volba τ obtížná. Volba τ vyžaduje kompromis mezi náklady na testování (určenými velikostí testovací sady) a potenciálními přínosy testování s vyšší silou pokrytí.

Tým výzkumníků z amerického Národního institutu pro standardy a technologie (NIST) zkoumal data v 15letém rozmezí za účelem získání poznatků o typech testování softwaru, které mohly detekovat základní chyby a zabránit selháním při používání systému. Následně výzkumníci NIST spočítali počty jednotlivých parametrů, které se podílely na poruchách. Ve čtyřech systémech zkoumaných týmem NIST zahrnovalo [11, obr. 1]

- $29 - 68\%$ chyb jediný faktor,
- $70 - 97\%$ chyb zahrnovalo jeden nebo dva faktory,
- $89 - 99\%$ chyb zahrnovalo tři nebo méně faktorů,
- $96 - 100\%$ chyb zahrnovalo čtyři nebo méně faktorů,
- $96 - 100\%$ chyb zahrnovalo pět nebo méně faktorů a
- žádná chyba nezahrnovala více než šest faktorů.

Na základě těchto výsledků je definováno empiricky odvozené pravidlo charakterizující rozložení interakčních chyb nazývané *interakční pravidlo* (angl. *interaction rule*) [4]:

> *„Většina selhání je vyvolána jednofaktorovými chybami nebo společným kombinatorickým účinkem (interakcí) dvou faktorů, přičemž postupně ubývá poruch způsobených interakcemi mezi třemi, čtyřmi nebo více faktory.“*

Maximální stupeň interakce u skutečných dosud pozorovaných poruch je šest. Pravidlo interakce vyjadřuje, že párové testování je užitečné, ale nemusí být dostatečné. Někdy může být zapotřebí kombinační (τ-cestné) testování pro τ větší než 2. Jelikož $96 - 100\%$ chyb zahrnovalo čtyři nebo méně faktorů, je rozumné zvolit sílu pokrytí τ čtyři a menší.

2.3.2 Seeding

Seeding znamená zařadit vybrané testovací případy do testovací sady před jejím vygenerováním. Takto zařazené testy se pak označují jako *seed testy*. Seed testy zůstanou součástí vygenerované testovací sady bez jejich úprav. Hodnoty seed testů mohou být specifikované pouze částečně, neboli hodnoty některých parametrů nebudou zadány. Metoda generující testovací sadu doplní chybějící hodnoty částečně specifikovaných seed testů. [16]

Seeding se používá zejména kvůli:

1. Explicitní specifikaci důležitých kombinací. Tester si může být vědom kombinací, které budou pravděpodobně použity v reálném provozu systému, a proto může určit testovací sadu, která bude tyto kombinace obsahovat.
2. Minimalizaci změn v testovací sadě při jejím opětovném vytvoření. V případech, kdy dojde k malé úpravě popisu testovací domény, jako je přidání parametrů nebo jejich hodnot, může vygenerovaní nové testovací sady způsobit její nežádoucí změny.

2.3.3 Omezení

τ-cestné pole pokrytí lze přímo použít jako testovací sadu k systematickému testování interakcí mezi τ nebo méně parametry. Nicméně, v reálných programech obvykle existují závislosti mezi parametry, tj. omezení. Omezení vyjadřují, že dané interakce nelze v testovacím případě dosáhnout, a limitují konstrukci pole pokrytí zakázáním některých kombinaci. [27]

Omezení prostředí (angl. *Environment Constraints*) jsou dána běhovým prostředím testovaného systému (např. OS Linux nelze nikdy kombinovat s prohlížečem Internet Explorer). Obecně platí, že kombinace, které porušují omezení prostředí, by se za běhu nikdy nemohly objevit, a proto musí být vyloučeny z testovací sady. [5]

Systémová omezení (angl. *System Constraints*) jsou dána sémantikou SUT (např. uživatel nemůže vybrat hodnotu menší než 10). Tato omezení lze vynutit například za účelem snížení počtu testovacích případů a tím i ceny, nebo je lze ignorovat například za účelem testování robustnosti systému (zda systém řádně odmítá neplatné vstupní kombinace). [5]

Omezení mohou být reprezentována buď jako zakázané n-tice, povolené n-tice, nebo formálně specifikované pomocí výrokové logiky nebo logických výrazů pomocí booleovských, relačních nebo aritmetických operátorů.

Omezení lze řešit následujícími způsoby: [5]

- Zpracování omezení ještě před spuštěním samotného algoritmu generujícího testovací sadu. Tento mechanismus lze použít pouze tehdy, pokud jsou na vstupu zadány pouze povolené n-tice a lze zabránit změně algoritmu generování testu.
- Nahrazení nevalidních testovacích případů validními po vygenerování testovací sady.
- Integrace zpracování omezení skrze ad-hoc proceduru do algoritmu generujícího testovací sadu.
- Integrace výběru validních n-tic podle specifikovaných omezení integrací SAT řešiče do algoritmu generujícího testovací sadu.

2.3.4 Techniky pro generování pole pokrytí

Konstrukce optimálního pole pokrytí (tj. z hlediska minimálního počtu řádků/testovacích případů) je obtížný problém kombinatorické optimalizace a bylo dokázáno, že se jedná o nedeterministicky polynomiální (tj. NP-těžký (angl. *NP-hard*)) výpočetní problém. Výpočetní

čas a složitost problému exponenciálně rostou s nárůstem počtu vstupních parametrů. V důsledku toho byly hledány způsoby, jak tento problém účinně řešit. [16]

Algoritmy pro generování kombinatorické testovací sady/pole pokrytí lze rozdělit podle toho, jaký stavební blok používají pro generování testovací sady.

Generování založené na testu (angl. *Test Based Generation*), také označované jako *one-test-at-a-time* nebo *one-row-at-a-time*, dále jen OTAT, používá jako základní blok testovací případ, kdy v každé iteraci rozšíří pole pokrytí o jeden nový test (řádek), který pokrývá co nejvíce možných zatím nepokrytých kombinací. Pro zvolení nejlepší kombinace v každé iteraci se používají optimalizační techniky, které se dále dělí na *náhodné* (angl. *random*), *hladové* (angl. *greedy*), *heuristické vyhledávání* a *metaheuristické* algoritmy. [5]

Generování založené na parametru (angl. *Parameter Based Generation*), také označované jako *one-parameter-at-a-time* nebo *one-factor-at-a-time*, dále jen OPAT, používá jako stavební blok vstupní parametr a začíná vygenerováním všech možných kombinací pro prvních τ parametrů. Tuto sadu následně v každé iteraci rozšiřuje horizontálně přidáním jednoho parametru do každého řádku a pokud je potřeba, rozšíří sadu vertikálně o nové řádky. Tato strategie byla poprvé použita v algoritmu *In-parameter-order* a dále rozvinuta v modifikacích IPO algoritmu jako jsou `IPOG`, `IPOG-D`, `IPOG-F` a `IPO-s`. [5]

2.4 Kombinatorické testování s omezeními

Tato podkapitola čerpá z [27, 28].

Pole pokrytí a jeho definice 2.2.4 uvažují, že každá možná τ-cestná kombinace je proveditelná a může vyvolat selhání SUT. Nicméně, v reálných systémech často není možné některé hodnoty některých parametrů spolu kombinovat kvůli omezením SUT. Omezení mohou být zavedena kvůli nekonzistentnosti hardwarových komponent, limitaci možných konfigurací SUT nebo zkrátka kvůli rozhodnutím pří návrhu SUT.

Aby bylo možné začlenit omezení do CT, je třeba definici pole pokrytí rozšířit na definici omezeného pole pokrytí, které je definováno takto:

Definice 2.4.1 (Omezené pole pokrytí (angl. *Constrained Covering Array*) [27])**.** *τ-cestné omezené pole pokrytí* s ohledem na množinu omezení C je $N \times n$ pole, ve kterém

1. každý sloupec $i\,(1 \leq i \leq n)$ obsahuje pouze elementy z množiny V_i,
2. každý řádek splňuje všechna omezení v množině C a
3. pokrývá každou τ-cestnou kombinaci splňující všechna omezení v množině C alespoň jednou.

Omezení mohou být buď *tvrdá* (angl. *hard*), nebo *měkká* (angl. *soft*). Tvrdá omezení vyžadují, aby se určité kombinace parametrů nikdy neobjevily v žádném testovacím případě, protože brání v proveditelnosti testu, neboli tvrdá omezení jsou taková omezení, jež musí splňovat všechny testovací případy.

Naopak, měkká omezení určují kombinace, které, na základě znalostí a zkušeností testerů, není třeba testovat, neboli měkká omezení jsou taková omezení, jež by měla být splněna

co největším možným počtem testovacích případů. Do testovací sady je možné zahrnout testovací případy porušující měkká omezení, ale obecně jsou nežádoucí a nepřínosné pro efektivitu testování.

Mohou také nastat situace, ve kterých je potřeba provést vybrané testovací případy ve vybraných konfiguracích SUT. V takových případech se omezení dále rozlišují na *celosystémová omezení* (angl. *System-Wide Constraints*) a *omezení specifická pro testovací případ* (angl. *Case-Specific Constraints*). Celosystémová omezení určují validní prostor, ze kterého jsou vybírány konfigurace, zatímco omezení specifická pro testovací případ určují množinu konfigurací, ve kterých je testovací případ možné provést. Tato omezení mohou být současně zohledněna v objektu pole pokrytí zohledňující testovací případy (viz podkapitolu 2.2.4).

2.4.1 SAT a SMT solvery

Omezení jsou často vyjádřena pomocí výrokové logiky nebo logických výrazů použitím booleovských, relačních nebo aritmetických operátorů. K řešení takto reprezentovaných omezení se používají SAT a SMT solvery, proto tato kapitola provádí jejich rešerši.

Problém splnitelnosti booleovské formule a SAT solvery

Otázka, zda je daná formule výrokové logiky splnitelná, tedy zda existuje přiřazení binárních hodnot jednotlivým proměnným tak, aby tato formule byla pravdivá, se nazývá *problém splnitelnosti* (angl. *Boolean satisfiability problem*, zkráceně SAT). Problém splnitelnosti patří mezi nejznámější NP-úplné problémy. Programy, které tuto rozhodovací proceduru implementují, se nazývají *SAT solvery*. [14]

Vstup do většiny solverů je uváděn v *konjunktivní normální formě* (angl. *conjunctive normal form*), neboli jako konjunkce klauzulí, kde klauzule je disjunkce literálů. Standardně se používá DIMACS-CNF formát. V praxi jsou nejvíce v dnešních SAT solverech používány dvě třídy nejvýkonnějších algoritmů pro řešení SAT problémů: moderní verze DPLL algoritmu a stochastické lokální prohledávání. Většina současných SAT solverů používá proceduru DPLL v různých optimalizovaných verzích. Algoritmus Davis–Putnam–Logemann–Loveland (DPLL) používá několik hlavních operací: [21]

- základní simplifikace klauzulí,
- substituce – přiřazení hodnoty proměnné,
- propagace – aplikace deduktivních pravidel, zejména pravidla jednotkové klauzule,
- návrat – navrácení do nějakého předchozího bodu substituce při nalezení konfliktních ohodnocení.

Známými SAT řešiči jsou např. MiniSAT, PicoSAT a CryptoMiniSAT.

Problém splnitelnosti formulí v teoriích predikátové logiky a SMT solvery

Problém splnitelnosti v teorii (angl. *Satisfiability Modulo Theories*, SMT) rozšiřuje SAT problém o splnitelnost výrazů predikátové logiky prvního řádu s rovností a prvky z různých

teorií predikátové logiky určitého univerza (jako jsou lineární aritmetika, teorie polí, bitové vektory ad.). Mezinárodní iniciativa SMT-LIB zaměřující se na výzkum a vývoj v oblasti SMT vydává standard formátu SMT-LIB, jazyku pro zápis SMT, pro využití v programech řešících SMT, a definující sémantiku v rámci SMT. [23]

SMT solver dostává na vstup vhodně zapsanou formuli. Vyřešení problému splnitelnosti v teorii pro zadanou formuli zadané teorie, případně jejího fragmentu, znamená rozhodnout, zda existuje přiřazení prvků univerza jednotlivým proměnným formule tak, aby tato formule byla pravdivá. Výsledkem běhu programu solveru je pak odpověď SAT (z zkratka anglického *satisfiable* tedy splnitelné), pokud existuje nějaké takové ohodnocení proměnných, v opačném případě vrací odpověď UNSAT (zkratka anglického *unsatisfiable* tedy nesplnitelné). Pokud se mu podařilo najít aspoň jedno řešení, je možné získat výstup ohodnocení jednotlivých proměnných ve formě kódovaného výstupu. [23]

Jazyk používá plně uzávorkovanou prefixovou notaci. Základní příkazy jazyka jsou deklarace či definice konstant a funkcí, nových druhů, a přidávání formulí do asercí, neboli podmínek, které musí být splněny. Ověření splnitelnosti pak spočívá v hledání ohodnocení všech konstant a funkcí splňující konjunkci všech asercí, podobně jako v SAT řešičích. Jazyk lze použít i pro inkrementální řešiče, pro které lze využít operací přidávání a odebírání asercí ze zásobníku. Ověření splnitelnosti pak vždy probíhá nad vrcholem zásobníku.

```
1 (set-option :produce-models true)
2 (set-logic QF_LIA)
3 (declare-fun x () Int)
4 (declare-fun y () Int)
5 (assert (= (+ x y) 9))
6 (assert (= (+ (* 2 x) (* 3 y)) 22))
7 (check-sat)
8 (get-model)
```

Výpis 1: Příklad zápisu v jazyce SMT-LIB 2.

Příklad zápisu programu v jazyku SMT-LIB 2 je uveden ve výpisu 1. Program začíná direktivou, která specifikuje nastavení samotného SMT solveru. Pokračuje příkazem pro specifikaci teorie `QF_LIA` (z anglického *quantifier-free linear integer arithmetic*, tedy lineární teorie celých čísel bez kvantifikátorů), definuje proměnné `x` a `y`, následně specifikuje dvě aserce a nakonec zadá příkaz pro dotázání se na splnitelnost asercí a získání modelu.

Existují dva základní přístupy k SMT solvingu, a to *Eager* a *Lazy*, s tím, že v praxi se častěji využívá Lazy přístup. Eager přístup spočívá v tom, že se snaží převést formuli prvního řádu, která obsahuje atomy určitých teorií, na splnitelnou formuli ve výrokové logice a následně položí dotaz SAT solveru, zda je formule splnitelná. Nevýhodou tohoto přístupu je, že způsob překladu je specifický pro každou teorii. Navíc ne vždy je tento překlad možný, neboť přeložená formule je velmi velká a těžko řešitelná pro SAT solver.

SMT solver založený na Lazy přístupu se skládá ze dvou typů komponent. Využívá SAT solver a potom pro každou teorii, jejíž atomy se v dané formuli vyskytují, používá speciální tzv. T-solver (neboli solver dané teorie), který umí pracovat s formulemi v CNF. [21]

Použití SMT solverů se do značné míry kryje se SAT solvery, často je nahradily, resp. rozšířily.

CVC5[5] je inkrementální SMT solver s otevřenými zdrojovými kódy, který je navržen pro snadné rozšiřování a poskytuje rozhraní v C++ a také rozhraní textové přes vstupní jazyk, tzn. že nástroj lze použít jak jako knihovnu, tak samostatně, tj. jako černou skříňku (angl. *black box*). Podporuje řadu teorií (resp. logik), např. číselné aritmetiky, bit vektory, pole, řetězce. Podporuje také kvantifikátory a nelineární logiky. [8]

OpenSMT[6] je inkrementální SMT solver s otevřenými zdrojovými kódy napsanými v jazyce C++, který podporuje standardní iniciativu SMT-LIB. Je postaven nad SAT solverem MiniSAT2. Nástroj byl implementován s důrazem na snadnou rozšiřitelnost o nové T-solvery, současně však zůstává efektivní. OpenSMT používá Lazy přístup. [8]

z3[7] je důkazní nástroj pocházející od Microsoft R©Research, nicméně má otevřené zdrojové kódy a je publikován pod licencí MIT. Je napsán převážně v jazyce C++ a podporuje většinu rozšířených OS. V Microsoftu se používá pro účely softwarové analýzy a verifikace a generování testů. Nástroj přijímá několik vstupních textových formátů, mj. vlastní formát a SMT-LIB. z3 provádí poměrně důkladné předzpracování vstupu a podle povahy vstupu používá různé nástroje. Ovládá oba přístupy k řešení SMT: Eager i Lazy, a zvolí ten, který usoudí jako vhodnější. Např. v teorii bit vektorů bývá mnohdy vhodnější použít Eager přístup, protože zakódování do Booleovských proměnných je snadné. [8]

2.4.2 Reprezentace omezení

Omezení v CT mohou být reprezentována různými formami a jejich kombinacemi. Tato část popisuje čtyři nejběžnější reprezentace omezení (zakázané n-tice, implikace, numerické omezení a shielding) a jednu neobvyklou reprezentaci určenou pro lepší schopnost testera vyjadřovat omezení (vestavěné funkce).

Zakázané n-tice

Přístup využívající *zakázané n-tice* (angl. *forbidden tuples*) je nejjednoznačnější reprezentace omezení. Každé omezení je vyjádřeno k-cestnou kombinací určující hodnoty parametrů, které se nesmí spolu vyskytnout v testovacím případě.

Implikace

Přípustné hodnoty některých parametrů mohou být určeny přiřazenými hodnotami jiným parametrů. Takové omezení může být reprezentováno pomocí implikace $p \rightarrow q$. Zakázané n-tice a implikační omezení jsou nejběžnější reprezentace omezení v CT a lze je snadno mezi sebou převést – pro zakázanou n-tici obsahující k parametrů je možné zkonstruovat k implikací, z nichž každá používá jednu $(k-1)$-cestnou kombinaci této zakázané n-tice jako svůj předpoklad.

[5] https://github.com/cvc5/cvc5
[6] https://github.com/usi-verification-and-security/opensmt
[7] https://github.com/Z3Prover/z3

Numerické omezení

Tato omezení se používají, pokud hodnoty parametrů musí splňovat daný aritmetický vztah. Numerické omezení je vyjádřeno dvojicí (f, P_f), kde $P_f \subset P$ je množina parametrů a f je funkce nad P_f, která musí být splněna. Například, pro testovaný model s parametry $P = \{a, b, c\}$, kde každý parametr může nabývat hodnot $\{0, 1, 2\}$ může být numerické omezení vyjádřeno jako (f, P) kde $f : a + b + c \geq 3$. Numerická omezení také mohou být převedena na omezení reprezentovaná zakázanými n-ticemi tak, že se vyzkouší všechny kombinace všech parametrů v P_f a vyberou se ty, které vztah f nesplňují.

Shielding

Obecně CT předpokládá, že všechny parametry mají vždy specifikovanou hodnotu. Může však nastat situace, kdy specifikace určité hodnoty parametru vylučuje specifikaci jakékoliv hodnoty jiného parametru, neboli daný parametr nesmí nabývat žádné hodnoty. Takové omezení se pak nazývá *shielding* a označuje se a/P_a, kde a je k-cestná kombinace nazývaná *shielding kombinace* a P_a je množina parametrů, označovaných jako závislé parametry na a, které nesmí mít specifikovanou hodnotu v případě kombinace a. Převod tohoto omezení na jiná omezení je možný například tak, že se zavede speciální hodnota reprezentující chybějící hodnotu daného parametru.

Vestavěné funkce

Pro ještě flexibilnější omezení je možné omezení reprezentovat pomocí *vestavěných funkcí* (angl. *Embedded Function*). Myšlenkou tohoto přístupu je definovat omezení pomocí funkce a vložit ji do testovacího modelu. Tato funkce je napsána v programovacím jazyce systému a generátor testů ji využije pro vytvoření testovacích případů. Například, pokud SUT má tři parametry $Year$, $Month$ a Day, a Day může nabývat hodnot $\{1, 10, last\_day\}$, kde hodnota $last\_day$ je určena kombinací $Year$ a $Month$. Pak je možné definovat funkci $f(year, month)$, která vrací konkrétní $last\_day$ jako třetí hodnotu parametru Day.

2.5 Algoritmy pro generování testovací sady

Tato podkapitola prezentuje metody a algoritmy pro tvorbu kombinatorické testovací sady. V úvodu jsou algoritmy prezentovány přehledově a v následující části je podrobně popsán algoritmus `FIPOG`.

2.5.1 Přehled algoritmických přístupů

Tato kapitola čerpá z [5].

Hladové (angl. *greedy*) **algoritmy** generují testy nalezením lokálně optimálního řešení a zajištěním, že každý nový test využívá maximální možný počet zatím nepokrytých kombinací. Tyto algoritmy jsou obvykle rychlejší než metaheuristické techniky, ale ne vždy vytvářejí nejmenší možné testovací sady. Tyto algoritmy vracejí spíše lokální než globální optimum. Do této kategorie se řadí algoritmy založené na *zpětném vyhledávání* (angl. *backtracking*),

metodě větví a mezí (angl. *branch and bound*), vyčerpávajícím vyhledávání, algoritmy založené na IPO atd.

Pro řešení optimalizačních problémů se hojně využívají **metaheuristické techniky**. Tyto algoritmy prohledávají okolí v oblasti řešení a vyberou nejvíce vyhovující[8] kandidátní řešení. Heuristické prohledávací metody, jako je *simulované žíhání* (angl. *Simulated Annealing*), vytváří menší sady než hladové algoritmy, ale vyžadují více času na nalezení řešení. Typickým reprezentantem jsou evoluční algoritmy a přírodou orientované algoritmy, např. Genetický algoritmus, Optimalizace mravenčích kolonií (angl. *Ant Colony Optimization*), Optimalizace hejnem částic (angl. *Particle Swarm Optimization*), Simulované žíhání, Harmony Search a Great Flood. Patří sem i heuristické algoritmy jako jsou Horolezecký algoritmus (angl. *Hill Climbing*) nebo Tabu Search,

Kategorie **adaptivních náhodných** (angl. *Adaptive random*) nebo **ad-hoc technik** obsahuje dva typy technik. Adaptivní náhodný algoritmus využívá vzdálenosti mezi hodnotami parametrů (např. Hammingovu vzdálenost) ke generování co nejvzdálenějších testovacích sad. Mezi Ad-hoc algoritmy se řadí ty algoritmy, které nevyužívají žádné další techniky a typicky vybírají testovací případy náhodně nebo na bázi nějaké vstupní distribuce.

Pro dosažení lepších výsledků se využívají i **hybridní přístupy**. Kombinováním vhodných technik lze snížit velikost i dobu generování pole pokrytí a zároveň zvýšit pokrytí a tím i detekci chyb. Byl použit například hybridní přístup hladového algoritmu s heuristickým vyhledáváním nebo matematický přístup se simulovaným žíháním.

Matematické metody pro konstrukci pole pokrytí jsou široce zkoumané v matematické komunitě. Existují dva přístupy matematických metod k vytváření testovacích sad. V prvním přístupu se sady vytvářejí přímo na základě matematické funkce, která spočte hodnotu každé buňky podle indexů řádky a sloupce. Tento přístup je většinou rozšířením matematických metod pro konstrukci ortogonálních polí. Druhý přístup je založený na rekurzivním konstruování větších testovacích sad z menších. Pro použití matematických metod musí systémové konfigurace splňovat určitá omezení (např. počet hodnot parametrů musí být uniformní), kvůli čemu je použitelnost těchto metod značně omezená. [13]

Výhodou hladové a metaheuristické techniky je, že je lze aplikovat na libovolnou velikost systémových konfigurací, tj. neexistuje žádné omezení počtu parametrů nebo počtu hodnot, kterých může každý parametr nabývat. Nevýhodou je, že vytvoření pole pokrytí zabere více času. Na druhou stranu jsou algebraické techniky extrémně rychlé, ale fungují pouze na podmnožinu systémových konfigurací.

2.5.2 Vybrané algoritmy pro generování testovací sady

Podle provedeného průzkumu v [5] se v praxi v drtivé většině uplatňují nástroje implementující hladové algoritmy pro generování testovacích sad. Jejich značnou rozšířenost lze odůvodnit tím, že generují testovací sadu poměrně rychle a jsou aplikovatelné pro libovolnou sílu pokrytí. Vygenerované testovací sady jsou často větší než testovací sady generované s použitím například metaheuristických algoritmů, avšak vykonání testů je většinou rychlé, proto se raději akceptují větší testovací sady namísto použití metaheuristického algoritmu, jehož doba běhu je často daleko delší než hladového algoritmu.

[8] Výběr je podle fitness funkce, která ohodnocuje vhodnost každého dostupného řešení.

Z tohoto důvodu jsou dále jsou představeny algoritmy ze skupiny IPO, které jsou typickými představiteli hladových algoritmů, přičemž podrobnější popis je věnován nejnovější verzi zvané `FIPOG`.

2.5.3 Přehled IPO algoritmů

Tato podkapitola čerpá z [6].

Rodina algoritmů *In-Parameter-Order* (`IPO`) je sada hladových algoritmů pro konstrukci pole pokrytí o přijatelné velikosti. Při konstrukci je v každé iteraci pole rozšířeno o jeden sloupec (horizontální rozšíření) a volitelně o další řádky (vertikální rozšíření).

Zatímco původní algoritmus byl omezen na konstrukci polí se silou pokrytí o hodnotě 2 (Pair-Wise), nové varianty algoritmus zobecnily pro generování polí s vyšší silou pokrytí (`IPOG`) a také do něj integrovaly zpracování omezení (`IPOG-C`). V dalších variantách `IPOG-F`, `IPOG-F2` došlo k rozšíření prohledávaného prostoru při horizontálním rozšíření. Varianta `IPOG-D` zahrnuje rekurzivní konstrukční metodu zaměřenou na snížení počtu kombinací, které musí být prozkoumány.

Všechny varianty začínají vytvořením počátečního pole pokrytí o síle τ obsahující všechny kombinace prvních τ parametrů. Poté je pole v každé iteraci rozšířeno o jeden sloupec dvourozměrným způsobem.

Nejdříve je nový sloupec přidán do pole horizontálním rozšířením. Tento krok přiřadí hodnoty do nového sloupec hladovým způsobem tak, aby se maximalizovalo pokrytí. Horizontální rozšíření skončí buď ve chvíli, kdy jsou pokryty všechny n-tice, nebo došlo k přiřazení hodnot ve všech řádcích v novém sloupci. Ve druhém případě nedošlo k pokrytí všech zbývající n-tic, a proto jsou přidány do kroku vertikálního rozšíření. V tomto kroku dojde k růstu pole přidáním dalších řádků, které obsahují chybějící n-tice.

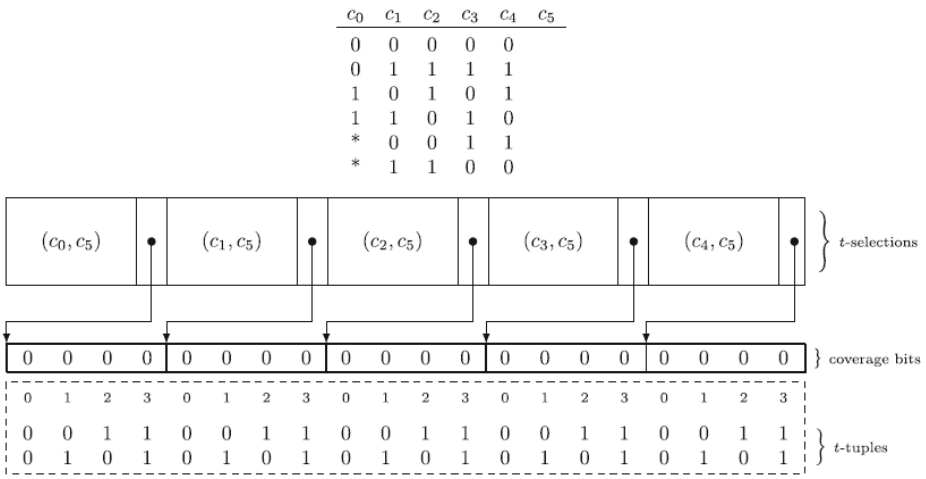
Při vertikálním rozšíření jsou všechny zbývající nepokryté n-tice přidány do pole, čímž se zajistí, že prvních i sloupců tvoří pole pokrytí. N-tice lze přidat buď jako nový řádek pole nebo vložením do již existujícího řádku, který vyhovuje n-tici. Druhý případ nastane tehdy, když řádek pole obsahuje *don't-care* hodnotu či hodnoty a n-tice je může nahradit tak, že nedojde k porušení pokrytí.

`IPOG-F` a `IPOG-F2` rozšiřují prohledávaný prostor při horizontálním rozšíření optimalizací pořadí, ve kterém jsou řádky rozšiřovány. V každé iteraci jsou navíc procházeny zatím neobsazené řádky za účelem nalezení nejlepší pozice a hodnoty k přiřazení. Zatímco `IPOG-F` při určování nejvhodnější hodnoty bere v úvahu skutečné pokrytí, `IPOG-F2` se snaží odhadnout množství pokrytých n-tic. Tato optimalizace sice urychluje implementaci ve srovnání s `IPOG-F`, ale snižuje se její přesnost.

Podrobnější popis algoritmů `IPOG` a `IPOG-C` včetně názorných příkladů je možné nalézt v [24].

2.5.4 Algoritmus FIPOG

Nejnovější varianta popsaná v [7] nazvaná *Fast In-Parameter-Order-General* (dále jen `FIPOG`) zavádí několik optimalizací do původního algoritmu `IPOG` a značně zrychluje tvorbu testovací sady. Nicméně, nejdříve je nutné definovat datovou strukturu `covering map`, kte-



Obrázek 2.2: Dvouúrovňová struktura `coverage map` pro ukládání kombinací parametrů v první úrovni a pro každou kombinaci parametrů seznam jejich n-tic ve druhé úrovni (převzato z [7]).

rou navrhli autoři algoritmu `IPOG`, pro efektivní uložení a vyhledávání dosud nepokrytých n-tic.

Při horizontálním rozšiřování je nutné si uchovávat záznam o všech dosud pokrytých a nepokrytých n-ticích. K tomu slouží dvouúrovňová struktura `covering map`, která má v první úrovni uložené ukazatele pro všechny kombinace mezi parametry již zpracovaných sloupců a parametrem právě rozšiřovaného sloupce. Každý ukazatel pak odkazuje do druhé úrovně struktury, kde jsou uložené informace o již zpracovaných n-ticích pro danou dvojici parametrů (viz obrázek 2.2). Bitové hodnoty na druhé úrovni vyjadřují, zda n-tice již byla (hodnota 1) nebo ještě nebyla (hodnota 0) pokryta. Index každého bitu se jednoznačně mapuje na jednu n-tici, proto není nutné mít explicitně uložené n-tice ve struktuře, ale namísto toho lze prostřednictvím bijektivní funkce *pack* spočíst pro danou n-tici index odpovídajícího bitu.

Při přiřazování hodnot do nového sloupce se postupuje řádek po řádku od shora dolů. `IPOG` v každém kroku maximalizuje pokrytí a zvolí takovou hodnotu, která pokryje co nejvíce zatím nepokrytých n-tic. Následně se pro každou hodnotu z domény hodnot nového sloupce spočítá, kolik n-tic by bylo pokryto, pokud by hodnota byla do zpracovávaného řádku přiřazena. Kvůli tomu je nutné vypočítat index odpovídajícího bitu pomocí funkce *pack* postupně pro všechny kombinace mezi aktuálně zpracovávanou hodnotou v novém sloupci a všemi hodnotami již zpracovaných sloupců v daném řádku. Po zjištění indexu se přistoupí na daný bit a pokud je jeho hodnota 0, tj. kombinace ještě nebyla pokryta, pak se inkrementuje počet pokrytelných n-tic. Na konci se vybere ta hodnota, jejímž přiřazením do řádku dojde k pokrytí největšího počtu n-tic a aktualizují se bitové hodnoty pro nově pokryté n-tice.

Současný výpočet počtu pokrytých n-tic pro n-tice se stejným prefixem

První optimalizace spočívá v odstranění redundantních výpočtů při určování indexů bitů odpovídajících n-ticím. Optimalizace vychází ze skutečnosti, že při rozšiřování pole o nový

sloupec i jsou hodnoty ve všech prvních $i-1$ sloupcích konstantní během celého horizontálního rozšíření. N-tice $(x_{i_1}, \ldots, x_{i_{t-1}}, x_i)$ pro t-vybraných sloupců $\{i_1, \ldots, i_{t-1}, i\}$ vždy obsahuje hodnotu z nového sloupce a zároveň prefix $(x_{i_1}, \ldots, x_{i_{t-1}})$ je vždy konstantní pro daný řádek a mění se pouze hodnota pro nový sloupec. Proto stačí spočítat index bitu pouze pro prefix, tedy pro $(x_{i_1}, \ldots, x_{i_{t-1}})$, a to navíc pouze jednou. Tento index se pak nazývá *bázový index*. Následně se zkontrolují bity na indexech $b, \ldots, b + v_i - 1$, kde b značí bázový index a v_i hodnotu pro nový sloupec. Dané indexy reprezentují hledané n-tice díky skutečnosti, že n-tice jsou uloženy v `coverage map` v lexikograficky vzestupném pořadí.

Ořezání prohledávání první úrovně `coverage map`

Druhá optimalizace se zaměřuje na odstranění redundantního procházení bitů v případě, kdy všechny n-tice pro daný výběr t sloupců $\{i_1, \ldots, i_{t-1}, i\}$ již byly pokryty, a proto je možné všechny jím odpovídající bity zcela přeskočit. K tomu je nutné si pro každý výběr sloupců uchovávat ve struktuře `coverage map` na její první úrovni celočíselnou hodnotu udávající, kolik n-tic již bylo pokryto. Při pokrytí n-tice patřící do daného výběru sloupců se hodnota inkrementuje a jakmile bude stejná jako počet všech n-tic patřících do daného výběru, pak je jisté, že došlo k pokrytí všech n-tic a je možné přeskočit kontrolu bitů daného výběru.

Rozdělení kandidátních řádků

Třetí a zároveň poslední optimalizace se zabývá rychlejším vyhledáváním vyhovujících řádků při vertikálním rozšiřování pole. Na konci horizontálního rozšíření mají všechny řádky v novém sloupci i přiřazenou hodnotu. Při hledání kandidátních řádků (tj. řádků s alespoň jednou *don't care* hodnotou) pro nepokrytou n-tici lze přeskočit všechny řádky, jejichž hodnota v novém sloupci je jiná, než hodnota v nepokryté n-tici, protože tyto řádky nikdy nebudou nepokryté n-tici kvůli rozdílné hodnotě vyhovovat. Podle hodnoty v novém sloupci lze kandidátní řádky rozdělit do skupin a následně pro každou nepokrytou n-tici prohledat pouze tu skupinu, jejíž řádky mají v novém sloupci stejnou hodnotu, jako je hodnota v nepokryté n-tici.

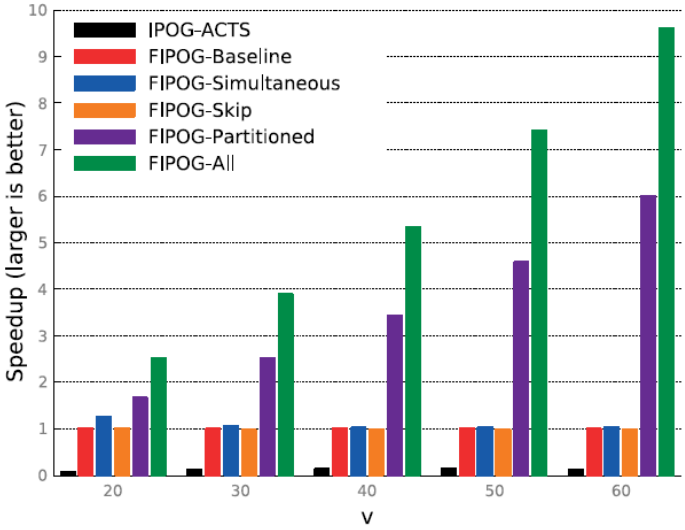
Vyhodnocení navržených optimalizací bylo provedeno na několika příkladech a ve všech si algoritmus `FIPOG` vedl výrazně lépe než původní algoritmus `IPOG`. Výsledky jednoho experimentu zobrazuje graf na obrázku 2.3.

2.6 Existující řešení

Webová stránka www.pairwise.org obsahuje seznam všech dostupných nástrojů pro generování pole pokrytí (více než 50 nástrojů). Nicméně, z tohoto množství se v praxi uplatnilo pouze několik málo nástrojů a ostatní sloužily spíše pro demonstraci nově navrženého algoritmu než pro praktické využití.

Pairwise Independent Combinatorial Testing[9] (dále jen PICT) je populární testovací nástroj založený na přístupu OTAT (viz podkapitolu 2.3.4). Je dostupný jako program

[9] https://github.com/microsoft/pict



Obrázek 2.3: Graf závislosti velikosti domén parametrů na zrychlení (převzato z [7]). Vyhodnocení bylo provedeno pro 3-cestné kombinace 6 parametrů. Hodnoty jsou uvedeny pro `IPOG` v nástroji ACTS, následně tři `IPOG` verze, kde každá implementuje jednu z optimalizací (Simultaneous, Skip, Partitioned) a nakonec verzi implementující všechny optimalizace (All).

příkazové řádky i jako webová služba. Ačkoliv název nástroje napovídá, že umí generovat pouze párové kombinace, není tomu tak a nástroj podporuje libovolně velké kombinace. Nástroj podporuje omezení parametrů jak v podobě implikace tak v podobě zakázaných n-tic. Datový typ parametru musí být buď řetězec nebo číslo. Parametry mohou mít aliasy a definované parametry mohou být použity pro definování nového parametru. Nástroj podporuje i seeding, tedy specifikace počáteční sady testovacích případů, a váhování hodnot parametrů, díky čemu lze specifikovat, které hodnoty mají být při generování upřednostněny.

Combinatorial Testing Web-based Editor and Generator[10] (dále jen CTWedge) je webový nástroj postavený nad programy ACTS a CASA a umožňuje uživateli zvolit si, který z programů se použije pro vygenerování testovací sady, ale už neumožňuje zvolení konkrétního algoritmu (nástroj ACTS implementuje několik variant IPO algoritmu). Podporovanými datovými typy parametrů jsou pravdivostní hodnota (`boolean`), rozsah celočíselných hodnot (`range`) a výčet hodnot (`enumerative`). Omezení jsou podporovaná a zapisují se v podobě implikace. Do nástroje je integrovaný textový editor Xtext, který umožňuje například zvýraznění syntaxe, našeptávání a označení řádků s chybnou syntaxí, což značně ulehčuje zápis testovaného modelu. [2]

CAgen[11] je další webový nástroj, který využívá ke generování testovací sady nejaktuálnější verzi z rodiny IPO algoritmů a to algoritmus `FIPOG` a ještě jeho dvě varianty `FIPOG-F` (kombinace `FIPOG` a `IPOG-F`) a `FIPOG-F2` (kombinace `FIPOG` a `IPOG-F2`). Jedná se o čistě klientskou aplikaci bez jakékoliv služby zajišťované backendem. Jádro aplikace generující tes-

[10] http://foselab.unibg.it/ctwedge
[11] https://matris.sba-research.org/tools/cagen

tovací sadu je napsáno v programovacím jazyce Rust a je kompilováno do WebAssembly[12], díky čemuž je možný jeho běh v internetovém prohlížeči. Nástroj dále umožňuje správu definovaných testovacích modelů pomocí *pracovních prostorů* (angl. *workspace*), které se ukládají do lokální paměti internetového prohlížeče. Podporované datové typy jsou stejné jako v případě nástroje CTWedge. Omezení se specifikují opět v podobě implikace. [26]

Kombinatorické testování bylo integrováno i do některých testovacích frameworků, například JCUnit[13] (algoritmus `IPOG`) nebo NUnit[14] (podpora pouze Pair-Wise).

Advanced Combinatorial Testing System[15] (dále jen ACTS), dříve známý jako FireEye, je pravděpodobně nejznámější nástroj pro generování t-cestných kombinatorických testovacích sad. Jde o akademický nástroj vyvinutý výzkumníky z Národního institutu standardů a technologie (angl. *National Institute of Standards and Technology*) v USA. ACTS podporuje generování maximálně 6-cestných kombinatorických testů. Dostupné datové typy parametrů jsou opět stejné jak u nástrojů CTWedge a CAgen. Pro generování si lze vybrat algoritmus `IPOG` nebo některou z jeho variant (`IPOG-F`, `IPOG-F2`, `IPOG-D`, `IPOG-C`) či Base Choice. Podporovaná jsou jak omezení, tak i seeding či generování pokrývajícího pole s proměnlivou silou. Podle uživatelské příručky ACTS je `IPOG-D` preferovanou strategií pro větší systémy, zatímco ostatní algoritmy jsou vhodnější pro středně velké systémy s méně než 20 parametry a 10 hodnotami na parametr v průměru. ACTS je implementovaný v programovacím jazyce Java a tudíž nezávislý na platformě. Nástroj lze použít několika způsoby, protože ACTS poskytuje grafické uživatelské rozhraní, aplikační programové rozhraní i rozhraní příkazového řádku. Testovací sadu lze uložit do souboru ve formátu CSV. [30, 1]

[12] Webový standard, který definuje binární formát a odpovídající pseudo-jazyk symbolických adres pro přenositelný strojový kód spustitelný na webových stránkách.

[13] https://github.com/dakusui/jcunit

[14] https://docs.nunit.org/articles/nunit/writing-tests/attributes/pairwise.html

[15] https://csrc.nist.rip/groups/SNS/acts/documents/comparison-report.html#acts

Kapitola 3

Nástroj Combine a analýza jeho současné implementace

V této kapitole je popsána analýza aktuální implementace nástroje Combine. Nejdříve je představena platforma Testos, v rámci které je práce řešena. Následující analýza se zabývá architekturou a funkcemi nástroje, přičemž poukazuje především na jejich chyby a důležité nedostatky.

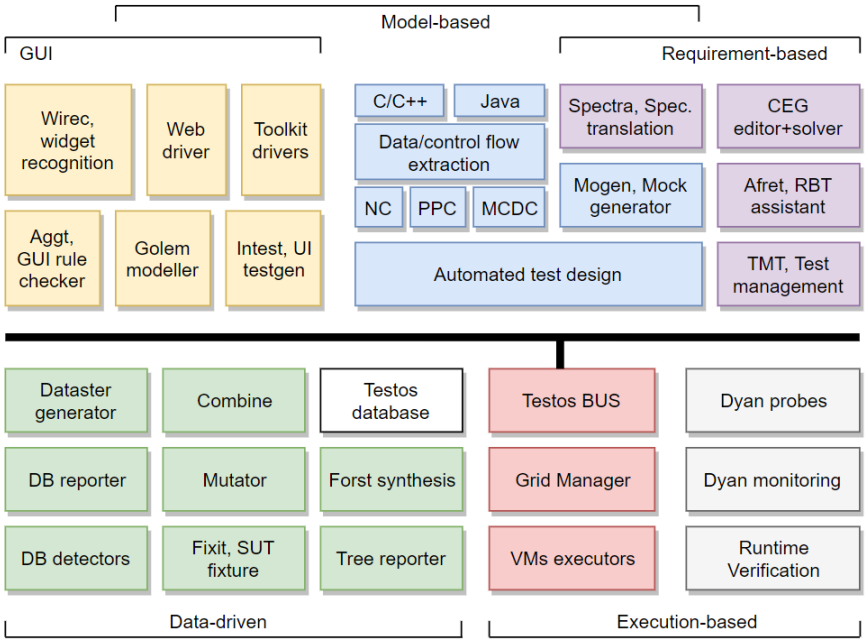
3.1 Platforma Testos

Testos (Test Tool Set) [22] je projekt vyvíjený na Fakultě informačních technologií VUT v Brně, jehož hlavním cílem je vytvoření sady nástrojů podporující automatizované testování softwaru. Nástroje v platformě Testos (viz obrázek 3.1) kombinují různé úrovně testování a lze je řadit do několika kategorií:

- testování založené na modelech (*Model-based*),
- testování založené na požadavcích (*Requirement-based*),
- testování grafického uživatelského rozhraní (*GUI*),
- daty řízené testování (*Data-driven*),
- dynamická analýza (*Execution-based*).

Tato práce patří do kategorie testování řízené daty, která, kromě nástroje Combine, obsahuje nástroje pro

- detekci dat uložených v relačních databázích (db-detectors [18], db-reporter [10], DeCon [17]),
- generování testovacích dat pro relační databáze (dbgenx [9], dataster [15]),
- detekování závislostí datových struktur ve strukturovaných datech (s-detector [19]),
- automatizovanou syntézu stromových struktur z reálných dat (treaper [25]),
- generování strukturovaných testovacích dat (gestr [20]).



Obrázek 3.1: Platforma Testos (převzato z [22]).

3.2 Představení nástroje Combine

Combine je webový nástroj[1] určený pro vygenerování testovací sady splňující zadané kombinační kritérium pro specifikované vstupní parametry. Konkrétně, nástroj vygeneruje pole pokrytí o zadané síle pokrytí obsahující všechny τ-cestné kombinace bloků vstupních parametrů (viz podkapitolu 2.2.2).

Nástroj aktuálně podporuje:

- generování až 6-cestných kombinatorických testů,
- 10 datových typů vstupních parametrů,
- specifikaci omezení bloků v podobě implikace,
- uložení vygenerované testovací sady ve formátech CSV, JSON a XML,
- randomizaci *don't care* hodnot ve vygenerované testovací sadě a
- generování testovací sady obsahující buď konkrétní hodnoty parametrů anebo pouze indexy odpovídajících bloků.

Nástroj má typickou architekturu webového nástroje, tedy skládá se z částí frontend a backend. Frontend je v podobě jednostránkové aplikace nevyužívající žádný framework a komunikující asynchronně s backend částí skrze technologii AJAX. Backend je implementovaný

[1] Dostupný na https://combine.testos.org/.

v programovacím jazyce Python a je přístupný skrze aplikační rozhraní vytvořené skrze framework *Flask*[2].

3.3 Analýza webového rozhraní

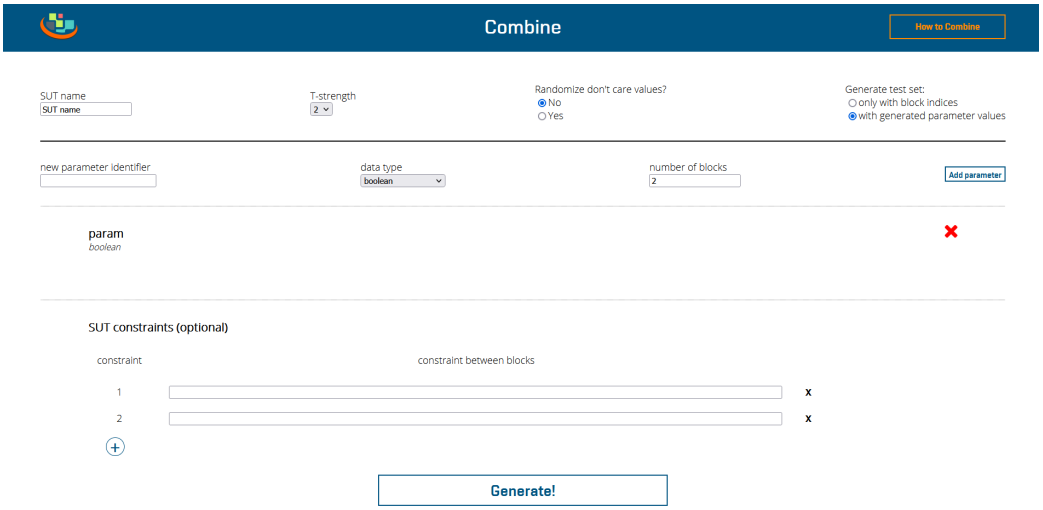
Webové rozhraní je tvořeno jednou hlavní HTML stránkou (viz obrázek 3.2), která slouží pro specifikaci parametrů a omezení modelu, pro který se má vytvořit testovací sada. Kromě hlavní stránky obsahuje ještě jednu stránku vedlejší, která zobrazuje nápovědu pro použití aplikace. Hlavní stránka má jednoduchou strukturu a obsahuje tři formuláře pro specifikaci

- nastavení algoritmu, který bude generovat testovací sadu (síla pokrytí, náhodné generování *don't care* hodnot, generování s konkrétními hodnotami nebo s indexy bloků),
- nového parametru, jeho datového typu a počtu bloků,
- bloků vytvořených parametrů a
- omezení mezi bloky parametrů.

Při používání aplikace lze narazit na několik chyb a zpozorovat několik nedostatků:

- Struktura stránky a rozložení jejich prvků:
  - Nevhodné rozložení vytvořených parametrů – formulář obsahující bloky vytvořeného parametru zabírá skoro celou šířku obrazovky, ale přitom jsou textové definice bloků většinou velmi krátké.
  - Nevhodně umístěné tlačítko pro zahájení generování – nachází se až za všemi parametry a omezeními.
  - Chybějící navigace – těžká orientace mezi definovanými parametry a omezeními při jejich větším počtu.
- Vytvořené parametry a definovaná omezení:
  - Nelze měnit pořadí vytvořených parametrů.
  - Nelze měnit pořadí definovaných bloků v rámci parametru.
  - Nelze přejmenovat vytvořený parametr.
  - Nelze změnit datový typ vytvořeného parametru.
  - Nelze pro daný parametr zobrazit pouze ta omezení, která s ním souvisí (tj. obsahují některý z jeho bloků).
  - Nelze duplikovat parametry nebo bloky v rámci parametru.
  - Nejasné chybové hlášky při špatně definovaném bloku parametru nebo omezení.
  - Nelze provádět hromadné akce (např. odstranění několika parametrů najednou, odstranění několika bloků najednou, ...).

[2]Mikro webový framework napsaný v programovacím jazyce Python nevyžadující konkrétní nástroje ani další vnitřní knihovny, nemá žádnou vrstvu abstrakce databáze, ověřování formulářů ani žádné jiné komponenty třetích stran poskytující běžné funkce.



Obrázek 3.2: Webové rozhraní nástroje Combine.

- Specifikace nového parametru:
  - Pole pro specifikaci počtu bloků je aktivní i pro datové typy, u kterých to nedává smysl (např. pro `boolean`).
  - Nelze vytvořit několik nových parametrů stejného datového typu se stejným počtem bloků najednou.
- Ostatní:
  - Nelze specifikovaný model (tj. parametry a omezení) uložit/nahrát do/ze souboru.
  - Nápověda není integrovaná do hlavní stránky, ale nachází se jinde.

Po vzoru nástrojů popsaných v podkapitole 2.6 by bylo vhodné do aplikace navíc přidat tyto funkce:

- Organizace pomocí *workspace* – každý model má svůj pracovní prostor, díky čemuž je možné mít několik rozpracovaných modelů najednou.
- Automatické ukládání modelů do lokální paměti prohlížeče.
- Našeptávání při specifikování bloků parametru nebo omezení.
- Zvýrazňování syntaxe.
- Explicitní označení nevalidních bloků a omezení.
- Předpřipravené datasety.

Zřejmě je nutné webovou část reimplementovat a to nejlépe pomocí vhodně zvoleného frameworku běžící na klientovi a zachovávající architekturu jednostránkové aplikace, protože webové stránky vytvořené pomocí základního HTML, CSS a JavaScriptu jsou často těžce spravovatelné a rozšiřovatelné. Moderní JavaScriptové frameworky nabízejí modularitu, lepší rozšiřovatelnost a správu zdrojového kódu a lepší podporu pro testování.

3.4 API a framework Flask

Definované API pro komunikaci mezi frontend a backend částí obsahuje pouze tři veřejné akce:

- Výchozí akce (metoda GET) zpřístupňující webové rozhraní s funkcemi v JavaScriptu.
- Akce `/help` (metoda GET) navracející stránku s nápovědou.
- Akce `/generate` (metoda POST), která je volána asynchronně z webové části tehdy, když uživatel požaduje vytvoření testovací sady. V případě POST metody jsou data požadavku uložena ve formátu JSON.

Vzhledem k tomu, že rozhraní není složité, pak neobsahuje žádné nedostatky a chyby. Nicméně, mohlo by být po vzoru moderních API orientováno na REST přístup. Navíc, framework Flask je jednoduchý a slouží spíše pro vývojové a testovací účely než pro produkční aplikace. Mezi nevýhody frameworku patří

- méně dostupných nástrojů – důležité funkce většinou zajišťují až knihovny třetích stran,
- větší náchylnost k bezpečnostním rizikům kvůli nutnosti instalovat knihovny třetích stran,
- náročnější údržba projektu s přibývajícím počtem knihoven třetích stran a zvětšující se implementací,
- nepodporování autentizace,
- nepodporování práce s databázemi a ORM,
- nekompatibilní pro vývoj složitých a větších aplikací a horší podpora pro MVP vývoj,
- nutnost Full-Stack znalostí,
- obtížné migrace,
- není asynchronní – příchozí požadavky jsou zpracovávány synchronně v pořadí, ve kterém přišly.

Ačkoliv je rozhraní veřejně přístupné, postrádá implementaci některých funkcionalit z hlediska bezpečnosti a dostupnosti služby:

- Počet aktuálně zpracovávaných požadavků by měl být omezený.
- Možnost volání metod rozhraní (jiným způsobem než skrze webové rozhraní) z jiných lokací než je webové rozhraní by mělo být omezené.
- Příliš dlouho zpracovávané požadavky by měly být zrušeny.

3.5 Backend a generátor testovací sady

Jádro aplikace obsahuje implementaci algoritmu `IPOG`, resp. `IPOG-C`, pro generování testovací sady. Jádro je implementované v jazyce Python, využívá moduly *PySMT*, který tvoří jednotné API pro různé solvery (z3, MathSAT a další) a *ply*, jenž je implementací nástrojů *lex* a *yacc* pro lexikální a syntaktickou analýzu. Skrze modul PySMT nástroj využívá z3 solver pro vyhodnocování omezení. Pomocí modulu ply jsou vytvořeny překladače jazyků pro definování bloků parametrů. Každý datový typ má definovaný svůj vlastní jazyk a tudíž má i svůj překladač.

Hlavním úskalím této části aplikace je volba programovacího jazyka. Použití jazyka Python s sebou sice nese spoustu výhod z hlediska času potřebného pro vytvoření řešení (velká úroveň abstrakce, objektově orientovaný, dynamicky typovaný, přenositelnost řešení, ...), ale zároveň jsou tyto výhody důsledkem řady omezení a nevýhod především z hlediska rychlosti a paměťových nároků výsledného řešení:

- Vysoká abstrakce, dynamické typování a interpretace kódu značně výkonnostně limitují výsledný program.
- *Globální zámek interpretu*[3] (angl. *Global Interpreter Lock*) znemožňuje vláknovou paralelizaci programu.
- Automatická správa paměti skrze Garbage collector a velké množství metadat, které si s sebou nesou vytvořené objekty, způsobují nadměrnou spotřebu paměti.

Generování testovací sady by mělo být co nejrychlejší, a proto by měl být algoritmus implementovaný v programovacím jazyku, který nepoužívá tak velkou abstrakci a je blíže hardwaru.

Z hlediska návrhu je aplikace přímo navázána na algoritmus `IPOG`, což brání rozšíření aplikace o další algoritmus pro generování testovací sady. Tato vazba by měla být nahrazena vhodným rozhraním.

Dalšími nedostatky jsou:

- Program implicitně používá z3 solver a neumožňuje použití jiného solveru.
- Algoritmus nepodporuje generování testovací sady s omezeními mezi hodnotami parametrů.

Další funkce, které by program mohl podporovat jsou

- seeding (viz podkapitolu 2.3.2),
- generování pole pokrytí s proměnlivou silou (viz podkapitolu 2.2.4,
- specifikace omezení v další podobách (např. pomocí zakázaných n-tic, viz podkapitolu 2.4.2).

[3] https://wiki.python.org/moin/GlobalInterpreterLock

3.6 Závěr analýzy a shrnutí identifikovaných nedostatků

Aktuální implementace trpí zásadními nedostatky ve všech svých částech. Ačkoliv se jejich přítomnost dala očekávat především z toho důvodu, že se jedná o prototypové řešení jakožto *Proof of concept* algoritmu `IPOG`, znemožňují použití nástroje pro řešení praktických a reálných problémů.

Frontend v podobě jednostránkové webové aplikace nenásleduje moderní způsob vývoje webových aplikací a nepoužívá žádný moderní framework. Navíc, webová stránka postrádá několik zásadních funkcí pro lepší ovládání a uživatelskou zkušenost.

Jednoduchý frontend následuje jednoduché Flask API, které je z pohledu dostupných funkcí značně limitované a neřeší požadavky z oblasti bezpečnosti a dostupnosti služby. Backend je implementovaný v jazyce Python, který není vhodný pro výkonnostně zaměřené programy a není navržený pro možnost budoucího rozšíření o další algoritmy pro generováno testovací sady.

Kapitola 4

Analýza a návrh nového řešení nástroje Combine

V této kapitole je prezentován celkový návrh výsledného produktu. Počáteční podkapitola se věnuje specifikaci požadavků, na základě kterých je dále prezentována architektura nástroje s detailním popisem každé jeho komponenty.

4.1 Specifikace požadavků

Mnoho požadavků vyplývá z analýzy aktuální implementace nástroje Combine reprezentované v kapitole 3. Z těchto požadavků byla následně provedena podrobná analýza. Uvedené požadavky spadají jak do funkčních[1], tak i nefunkčních[2] typů požadavků. Tabulka 4.1 uvádí hlavní požadavky, které jsou dále rozvedeny a podrobněji popsány.

Identifikátor	Název	Kategorie
req_web	Webová aplikace	Funkcionalita, UI
req_admin	Webová administrace	Funkcionalita, UI
req_backend	Backend aplikace	Funkcionalita
req_gen	Generátor testovací sady	Funkcionalita
req_testing	Testování kritických částí programu	Kód
req_code	Konzistentní styl kódu	Čitelnost, Udržovatelnost
req_cli	Konfigurace prostřednictvím rozhraní příkazové řádky	Funkcionalita

Tabulka 4.1: Základní požadavky.

Identifikátor	Název
req_web_params	Správa parametrů
req_web_params_blocks	Správa bloků parametrů
req_web_constraints	Omezení mezi bloky
req_web_validation	Validace definic bloků a omezení
req_web_projects	Správa projektů
req_web_projects_persist	Ukládání/nahrávání projektů
req_web_projects_persist_localstorage	Využití lokálního úložiště
req_web_projects_persist_restore	Automatické načtení posledního stavu
req_web_tutorial_app	Tutoriál na používání aplikace
req_web_tutorial_language	Tutoriál na syntaxi jazyků
req_web_keyboard	Ovládání prostřednictvím klávesnice
req_web_async	Asynchronní komunikace
req_web_testsuite_download	Uložení testovací sady

Tabulka 4.2: Požadavky týkající se webové aplikace.

Webová aplikace

Webová aplikace bude umožňovat správu parametrů, pro které se bude generovat testovací sada (`req_web_params`). Parametry bude možné přidávat, odebírat, pojmenovávat, specifikovat jejich datový typ, určit jejich pořadí, duplikovat a provádět akce s několika vybranými parametry najednou. Parametry budou obsahovat bloky, které budou specifikovat rozdělení vstupní domény parametru (`req_web_blocks`). Bloky bude možné přidávat, odebírat, určit pořadí, duplikovat a provádět akce s několika vybranými bloky najednou. Mezi bloky parametrů bude možné definovat omezení, jež bude vygenerovaná testovací sada splňovat (`req_web_constraints`). Syntaxe i sémantika definovaných bloků a omezení budou validovány online, tedy okamžitě bez komunikace s backend částí (`req_web_validation`).

Každý definovaný model SUT (tj. jeho parametry a omezení) bude uložen v samostatném *projektu* (`req_web_projects`). Projekty bude možné stahovat do souboru a nahrávat ze souboru ve formátu JSON (`req_web_projects_persist`). Navíc bude aplikace automaticky průběžně ukládat projekty do lokální paměti webového prohlížeče, pokud jí to bude umožněno, tj. budou jí přidělena potřebná oprávnění (`req_web_projects_persist_localstorage`). Pokud aplikace po své inicializaci zjistí, že lokální úložiště prohlížeče obsahuje uložená validní data projektů, pak je automaticky načte (`req_web_projects_persist_restore`).

Při prvním spuštění aplikace bude uživatel seznámen s jejími funkcemi a strukturou prostřednictvím interaktivního tutoriálu, který bude možné spustit i kdykoliv později (`req_web_tutorial_app`). Dostupný bude také podrobný tutoriál zabývající se jazyky pro definování bloků a omezení (`req_web_tutorial_language`).

Ovládání aplikace bude možné pomocí klávesnice, např. pro pohyb mezi formuláři nebo potvrzování akcí klávesou enter (`req_web_keyboard`). Vygenerovanou testovací sadu bude možné stáhnout v několika formátech (JSON, XML, CSV), přičemž bude možné zvolit si, zda součástí souboru mají být i patřičná metadata, např. název SUT, síla pokrytí, defino-

[1] Požadavky specifikující, jaké chování nebo jaké služby bude systém nabízet.

[2] Požadavky, které kladou kladná omezení na design a provedení (např. požadavky na výkonnost, standardy kvality, ...).

vané parametry, definovaná omezení a další (`req_web_testsuite_download`). Komunikace se serverem bude probíhat asynchronně a bude možné zažádat o vygenerování testovací sady, zastavení nebo zrušení generování testovací sady. V průběhu generování bude uživatel informován o stavu a průběhu generování (`req_web_async`).

Webová administrace

Identifikátor	Název
req_admin_authentication	Autentizace uživatelů
req_admin_overview	Přehled statistik
req_admin_jobs	Správa úloh
req_admin_keys	Správa API klíčů
req_admin_api	Nastavování API

Tabulka 4.3: Požadavky týkající se webové administrace.

Do webové administrace budou moci přistupovat pouze oprávnění uživatelé, kteří se budou muset autentizovat svým přihlašovacím jménem a heslem (`req_admin_authentication`). Webová administrace bude zobrazovat statistiky o nástroji, např. kolik je právě probíhajících úloh, kolik úloh bylo úspěšně či neúspěšné dokončeno, jaká je průměrná doba zpracování jedné úlohy, kolik parametrů modely SUT průměrně obsahují a další (`req_admin_overview`).

Administrace také bude zobrazovat údaje o aktuálně zpracovávaných úlohách a bude umožňovat provádět akce nad těmito úlohami, např. je zrušit či pozastavit (`req_admin_jobs`). Dostupná bude také správa API klíčů a bude možné generovat nové či aktuální mazat (`req_admin_keys`).

Poslední funkcí bude možnost nastavování parametrů API, jako jsou maximální čas zpracování požadavku nebo maximální možný počet současně zpracovávaných požadavků (`req_admin_api`).

Backend a API

Identifikátor	Název
req_backend_api	Aplikační rozhraní backendu
req_backend_api_rest	Architektura REST API
req_backend_api_keys	API klíče
req_backend_api_doc	Dokumentace API
req_backend_database	Komunikace s databází
req_backend_jobs	Správa úloh
req_backend_jobs_async	Asynchronní operace
req_backend_jobs_timeout	Omezení doby zpracování úlohy
req_backend_jobs_max	Omezení počtu současně zpracovávaných úloh

Tabulka 4.4: Požadavky týkající backendu a API.

Backend část aplikace bude veřejně přístupná prostřednictvím HTTP API (`req_backend_api`). API bude navrženo v architektonickém stylu REST (`req_backend_rest`). Jelikož bude

API veřejně přístupné, pak budou zamítány požadavky, které nebudou obsahovat validní *API klíč*[3], který se bude shodovat s některým z API klíčů uložených v databázi (`req_backend_api_keys`). Akce API, očekávané formáty požadavků a návratové kódy budou dostatečně zdokumentovány (`req_backend_api_doc`). Backend bude spravovat běžící procesy generátoru (`req_backend_jobs_async`).

Pro příchozí požadavek na vygenerování testovací sady backend spustí nový proces generátoru a předá mu na vstup data z požadavku (`req_backend_jobs`). Backend spravuje tyto běžící procesy generátoru a může je na vyžádání ukončit nebo pozastavit. Také měří dobu běhu zpracovávaných úloh a pokud doba dané úlohy přesáhne stanovený limit, pak úlohu ukončí (`req_backend_jobs_timeout`). Posledním kontrolovaným aspektem je počet aktuálně zpracovávaných úloh a pokud tato hodnota přesáhne daný limit, pak jsou další požadavky na vygenerování testovací sady zamítány (`req_backend_jobs_max`).

Zpracování příchozích požadavků bude probíhat asynchronně. Odpovědí na požadavek na vygenerování testovací sady proto nebude vygenerovaná testovací sada, ale pouze úloze přiřazený číselný identifikátor. Jakmile bude úloha dokončena, výsledná data budou předána prvnímu požadavku, který o ně zažádá správně specifikovaným identifikátorem úlohy a tajemstvím v podobě náhodně vygenerovaného řetězce, které mu přidělilo API. Pro zjištění, zda již byla úloha dokončena, bude nutné opakovaně se dotazovat se na stav úlohy (`req_backend_jobs_async`).

Generátor kombinatorické testovací sady

Identifikátor	Název
req_gen_smt	Volba SMT solveru
req_gen_constraints	Omezení kombinací bloků
req_gen_fipog	Implementace algoritmu `FIPOG`
req_gen_cli	Konfigurace nástroje skrze CLI
req_gen_input_json	Vstupní data ve formátu JSON
req_gen_input_valid	Ošetření nevalidních vstupů
req_gen_multiplatform	Kompatibilita s různými OS
req_gen_basechoice	Kritérium pokrytí Base Choice

Tabulka 4.5: Požadavky týkající se generátoru kombinatorické testovací sady.

Pro generování testovací sady bude použit algoritmus `FIPOG`(viz podkapitolu 2.5.4). Algoritmus bude při generování zohledňovat definovaná omezení (`req_gen_constraints`). Také Také bude podporovat vytvoření testovací sady splňující kritérium pokrytí Base Choice (`req_gen_basechoice`, viz podkapitolu 2.2.3). Pro řešení splnitelnosti omezení bude možné zvolit z více možných SMT solverů (`req_gen_smt`).

Generátor bude možné ovládat skrze vstupní přepínače příkazové řádky (`req_gen_cli`). Řešení musí být vytvořeno použitím takových nástrojů, které zajistí jeho snadnou přenositelnost mezi různými operačními systémy (`req_gen_multiplatform`).

Vstupní data reprezentující model SUT budou akceptována pouze ve formátu JSON (`req_gen_input_json`).

[3] Jedinečný identifikátor používaný k ověření uživatele, vývojáře nebo volajícího programu k API

Testování

Identifikátor	Název
req__testing__unit	Jednotkové testy
req__testing__integration	Integrační testy
req__testing__performance	Výkonnostní testy

Tabulka 4.6: Požadavky týkající se testování nástroje.

Základní funkcionalita programu bude pokryta jednotkovými testy (`req_testing_unit`). Správná komunikace frontend části s backend částí bude ověřena integračními testy (`req_testing_integration`). Generátor a ním implementovaný algoritmus `IPOG` bude porovnán z pohledu rychlosti generování a velikosti výsledné vygenerované testovací sady s původním řešením.

Konzistence kódu

Identifikátor	Název
req__code__style	Konzistentní styl kódu
req__code__struct	Konzistentní struktura kódu
req__code__patterns	Návrhové vzory
req__code__doc	Dokumentace kódu

Tabulka 4.7: Požadavky týkající se kódu implementace nástroje.

Styl kódu bude konzistentní a v souladu s běžným standardem pro daný programovací jazyk (`req_code_style`). Kód bude strukturovaný dle logických bloků a tak, aby byl přehledný (`req_code_struct`). Program bude plně dokumentovaný stylem, který je pro daný programovací jazyk běžný (`req_code_doc`). Na vhodných místech budou použity dobře známé návrhové vzory (`req_code_patterns`).

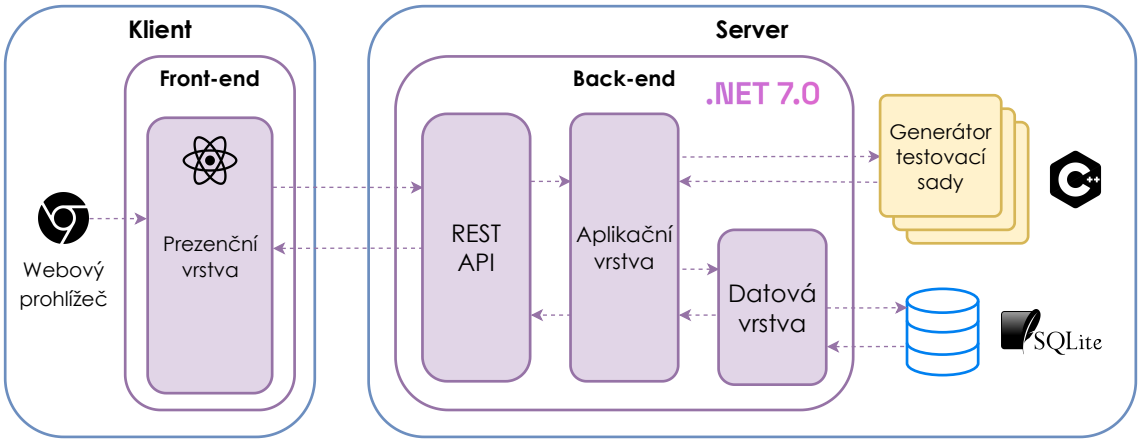
4.2 Návrh architektury nástroje Combine

Navržená architektura (viz obrázek 4.1) zachovává řešení původní implementace v podobě jednostránkové aplikace, která asynchronně komunikuje se serverem.

Pro tvorbu prezenční vrstvy na klientovi bude použita JavaScriptová knihovna pro tvorbu uživatelského rozhraní **ReactJS**[4]. Tento webový framework je vyvíjený Facebookem a komunitou samostatných vývojářů a společností a jeho hlavními výhodami jsou:

- Je deklarativní – React efektivně aktualizuje a vykresluje odpovídající komponenty při změně dat. Deklarativní přístup činí kód čitelnějším a ulehčuje jeho ladění.
- Umožňuje použití syntaxe JSX, která programátorům dovoluje vytvářet React komponenty podobné HTML kódu, který by jinak vkládali do webové stránky.

[4] https://reactjs.org/



Obrázek 4.1: Návrh architektury nástroj Combine.

- Může být použitý v obou architekturách SPA i SSR. Jeho vykreslování na straně serveru je lepší než u konkurenčních frameworků a nabízí velmi dobrou podporu SEO.
- Je flexibilní – vývojář si sám určí, jaká bude fungovat směrování, testování a jaká bude struktura souborů. Flexibilita je také podpořena jeho velkou modularitou.
- Implementace virtuálního DOM umožňuje jedno z nejrychlejších vykreslování na straně klienta.
- Je stabilní – jednosměrný tok dat zabraňuje potomkům, aby měnili své rodiče.
- React má dnes jednu z největších komunit návrhářů a vývojářů jakéhokoli programovacího jazyka. Také je open-source.
- Poměrně jednoduchá migrace mezi verzemi.
- React komponenty jsou znovupoužitelné, díky čemu urychlují vývoj a zajišťují lepší modularitu.

Backend část aplikace bude implementována pomocí **.NET 7**[5], což je open-source a multiplatformní (Windows, Mac a Linux) framework pro vytváření moderních cloudových aplikací připojených k internetu, jako jsou webové aplikace, aplikace IoT a mobilní backendy. .NET 7 byl navržen tak, aby poskytoval optimalizovaný framework pro vývoj aplikací, které se nasazují do cloudu nebo běží lokálně. Implementuje odlehčenou, vysoce výkonnou a modulární HTTP pipeline pro příchozí požadavky.

Mimoto nabízí širokou škálu snadno použitelných mechanismů pro autentizaci, autorizaci, ochranu dat a prevenci útoků, podporuje asynchronní programování, má vestavěné šablony pro integraci JavaScriptových frameworků (včetně Reactu) a obsahuje zabudovanou podpora dependency injection. Také podporuje testovatelnost vytvořených aplikací a má k dispozici velké množství knihoven třetích stran v podobě jednoduše spravovatelných NuGet balíčků.

[5] https://dotnet.microsoft.com/en-us/

Pro databázi bude použit **SQLite**[6], což je relační databázový systém obsažený v relativně malé knihovně napsané v C, na platformě .NET Core dostupný skrze `Entity Framework Core`[7].

Na rozdíl od databází založených na principu klient-server, kde je databázový server spuštěn jako samostatný proces, je SQLite pouze nevelká knihovna, která, po přilinkování k aplikaci, je k dispozici pomocí jednoduchého rozhraní. Každá databáze je uložena v samostatném souboru .dbm (Database Manager), kde se data ukládají za použití jednoduchého primárního klíče do stejně velkých bloků a používá hašovacích technik pro rychlý přístup k datům při vyhledávání podle klíče.

Charakteristickými prvky systému SQLite jsou:

- absence databázového systému ve formě abstrahovaného prostředí,
- absence konfigurace,
- absence serveru,
- databáze v jednom souboru, nezávislém na platformě – to s sebou nese výhody (migrace) i nevýhody (fragmentace).

Generátor testovací sady bude implementovaný v C++ především kvůli efektivitě kompilovaného kódu a velké schopnosti dostupných překladačů optimalizovat C++ kód, ale také kvůli jeho přenositelnosti, objektové orientaci, STL knihovně a podpoře šablon.

4.3 Generátor kombinatorické testovací sady

První podrobně popsanou komponentou celého nástroje je generátor kombinatorické testovací sady. Následující podkapitoly pojednávají o důvodech, které hrály roli při architektonickém návrhu komponenty a jejím rozhraní. Následně je popsána samotná architektura a také způsob zpracování příchozího požadavku na vygenerování testovací sady.

4.3.1 Definice vstupního formátu požadavku

Jedním z hlavních problémů původního řešení bylo zahrnutí generátoru testovací sady jako součást webového API (viz podkapitolu 3.5), což vedlo při novém návrhu generátoru k rozhodnutí, že komponenta bude plně samostatným programem, tzn. nezávislým na webové aplikaci. Z toho důvodu se musí pro generátor definovat nový obecný formát pro příchozí požadavky na vygenerování testovací sady, který bude definovat jeho rozhraní vůči backendu webové aplikace.

Příchozí požadavek v novém formátu bude postrádá některé vlastnosti či informace, které jsou v původní implementaci nástroje Combine dostupné díky tomu, že je logika generování součástí webového API:

[6] https://www.sqlite.org
[7] https://docs.microsoft.com/cs-cz/ef/core/

```
{
    "GenerationStrategy": {
        "Name": "IPOG",
        "Config": { "T": 5 }
    },
    "InputParameters": [{
            "Definition": "(declare-fun a () Int)",
            "Blocks": ["(< a 20))", "(>= a 20)"]
        }, {
            "Definition": "(declare-fun b () Int)",
            "Blocks": ["(< b 30))", "(>= b 30)"]
    }],
    "Solver": {
        "Name": "z3",
        "Preamble": "",
        "Config": {},
        "Constraints": [{
            "Definition": "(and (< a 20) (>= b 30))",
            "Parameters": [0, 1]
        }]
    }
}
```

Výpis 2: Formát vstupního požadavku na vygenerování testovací sady pro generátor testovací sady.

- Vzhledem k tomu, že je v původní implementaci generátoru přístupný celý kontext řešené úlohy, je vyhodnocování bloků a omezení prováděno přímo skrze rozhraní z3 solveru. Kvůli obecnosti nového formátu je nutné veškeré definice, které budou během generování testovací sady vstupovat do SMT solveru, definovat v jazyce SMT-LIB.
- Program nebude definice pro SMT solver nijak validovat ani zpracovávat. Jejich korektní specifikace je proto plně ponechána na zodpovědnosti uživatele. Na druhou stranu, díky tomu, že je program zcela nezávislý na sémantice bloků a omezení, může uživatel volně definovat a využívat v podstatě vše, co je součástí standardu SMT-LIB (pokud to zvolený SMT Solver podporuje).

Příchozí požadavek musí být ve formátu JSON (`req_gen_input_json`; viz výpis 4.3.2) a musí obsahovat tyto klíče:

- `GenerationStrategy` – objekt obsahuje informace o algoritmu či strategii, která bude použita pro vygenerování testovací sady. Každá strategie je definovaná svým unikátním názvem, který se musí uvést jako hodnota v klíči `Name`. Objekt `Config` pak musí obsahovat informace specifické pro jednotlivé algoritmy, např. pro `IPOG` musí objekt obsahovat klíč `T` s celočíselnou hodnotou.
- `InputParameters` – pole objektů, kde každý objekt reprezentuje jeden parametr SUT. Definice parametru v jazyce SMT-LIB musí být řetězcová hodnota klíče `Definition`. Součástí parametru je také pole definic jeho bloků v jazyce SMT-LIB v klíči `Blocks`.
- `Solver` – objekt určuje, který SMT solver se použije pro vyhodnocování splnitelnosti definovaných omezení. Stejně jako `GenerationStrategy` i tento objekt obsahuje klíč

`Name`, ve kterém je nutné specifikovat unikátní jméno požadovaného SMT solveru, a klíč `Config`, ve kterém lze určit nastavení specifická pro zvolený SMT Solver.

Dalším klíčem je `Preamble`, ve kterém lze specifikovat příkazy v jazyce SMT-LIB, které se mají přidat do každého vyhodnocení splnitelnosti omezení. Může jít například o definici rekurzivních funkcí, které mohou být využívány napříč vícero parametry, ale musí být načteny právě jednou, jinak by mohlo dojít ke kolizi jmen.

V posledním klíči `Constraints` se definují omezení pro generování testovací sady. Každé omezení obsahuje svoji definici v jazyku SMT-LIB a seznam indexů parametrů, kterých se dané omezení týká.

Kromě těchto syntaktických pravidel je nutné dodržet i několik sémantických:

- Definice bloků a omezení v jazyce SMT-LIB nesmí obsahovat příkaz `assert` (ačkoliv se o asertace jedná) a jejich výsledkem musí být pravdivostní hodnota `Boolean`.
- U omezení je nutné v klíči `Parameters` specifikovat indexy těch parametrů, kterých se dané omezení týká. Jelikož program neprovádí žádné parsování definovaných omezení v jazyku SMT-LIB, nemá žádnou jinou možnost, jak zjistit, při kterých kombinacích kterých parametrů je nutné zkontrolovat splnitelnost daného omezení.

4.3.2 Postup sestavení programu pro SMT solver

Během generování testovací sady se program dotazuje SMT solveru na splnitelnost definovaných omezení. Daný dotaz v SMT-LIB jazyku musí program sestavit ze vstupních definic parametrů, jejich bloků a omezení. Na následujícím příkladu bude vysvětleno, jak takové sestavení probíhá.

Nechť je vstupní úloha generátoru stejná jako je ve výpisu a nechť při generování dojde algoritmus do stavu, kdy musí ověřit splnitelnost kombinace prvního bloku prvního parametru a prvního bloku druhého parametru. Program začne sestavovat dotaz pro SMT solver následovně:

1. Zkonkatenuje všechny definice zúčastněných parametrů.

   ```
   (declare-fun a () Int)
   (declare-fun b () Int)
   ```

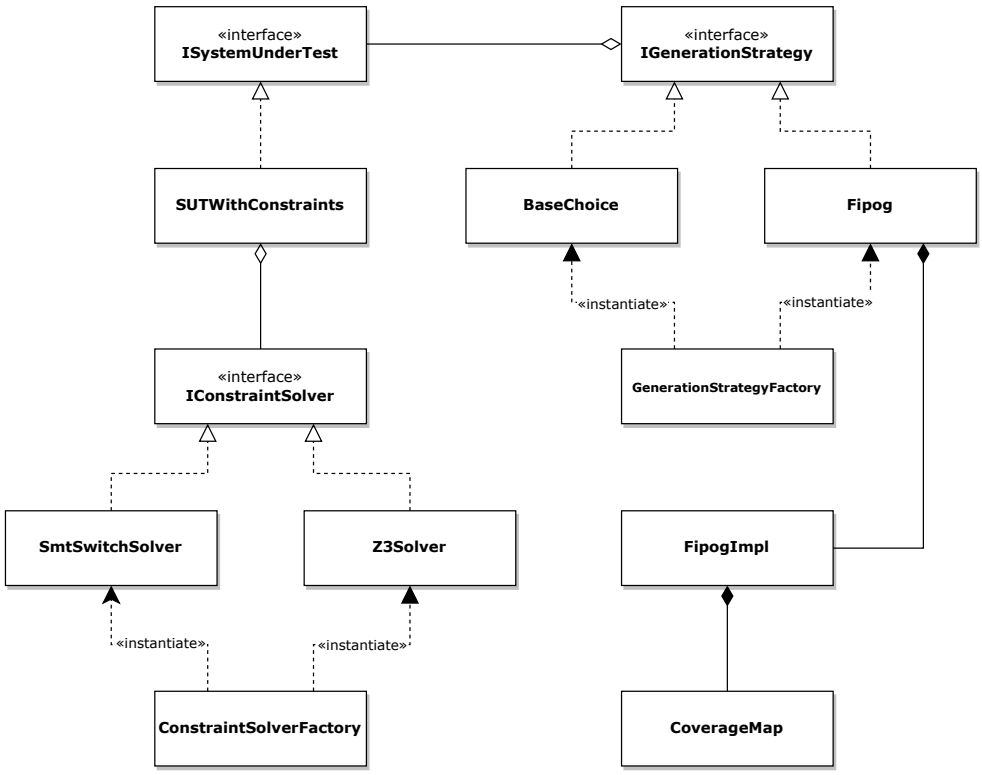
2. Spojí definice bloků operátorem `and` a výsledný řetězec nastaví jako parametr příkazu `assert`, čímž dojde k omezení hodnot parametrů, kterých mohou nabývat.

   ```
   (assert (and (< a 20) (< b 30))))
   ```

3. Všechna omezení, kterých se alespoň jeden z parametrů týká, tj. index parametru je specifikovaný v poli `Parameters`, jsou spojena operátorem `and` a výsledek je opět předán jako parametr příkazu `assert`.

   ```
   (assert (and (< a 20) (>= b 30))))
   ```

4. Nakonec se před odesláním dotazu do SMT solveru připojí na jeho začátek vše, co bylo specifikováno jako hodnota klíče `Preamble` v objektu pro definici SMT solveru.



Obrázek 4.2: Schéma architektury generátoru testovací sady.

Výsledný dotaz je zobrazen ve výpisu 3. Z příkladu je patrné, proč musí být uživatelem specifikované definice bloků a omezení bez `assert` příkazu a proč jejich výsledkem musí být `Boolean`.

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (and (< a 20) (< b 30))))
(assert (and (< a 20) (>= b 30))))
(check-sat)
```

Výpis 3: Sestavený dotaz na splnitelnost pro SMT solver v jazyce SMT-LIB.

4.3.3 Architektura generátoru testovací sady

V návaznosti na požadavky `req_gen_fipog`, `req_gen_smt` a `req_gen_basechoice` byla výsledná architektura navržena tak, aby byl generátor dobře rozšiřitelný o podporu dalších algoritmů pro generování testovací sady či SMT solverů.

Architektura generátoru je zobrazena na obrázku 4.2 a obsahuje:

- *ISystemUnderTest* – rozhraní, které musí implementovat každá třída určená pro uchování a správu parametrů ze vstupního požadavku.

- *IGenerationStrategy* – rozhraní, které musí implementovat každý algoritmus určený pro generování testovací sady.
- *GenerationStrategyFactory* – tovární metoda pro vytvoření instance třídy implementující algoritmus pro generování testovací sady, jehož jméno se shoduje s názvem v příchozím požadavku.
- *ConstraintSolverFactory* – tovární metoda pro vytvoření instance třídy zapouzdřující logiku požadovaného SMT solveru.
- *SUTWithConstraints* – třída spravující vstupní parametry i omezení z příchozího požadavku.
- *SmtSwitchSolver*, *Z3Solver* – třídy zapouzdřující konkrétní instance SMT solverů z externích knihoven.
- *BaseChoice*, *Fipog* – třídy implementující algoritmy pro generování testovací sady.
- *FipogImpl* – implementace algoritmu Fipog. Ukazatel na tuto implementaci je uložen v instanci třídy *Fipog*, tj. je zde použita technika *pimpl*[8].
- *CoverageMap* – implementace struktury pro uchování všech kombinací parametrů a jejich n-tic (viz kapitolu 2.5.4).

Na základě požadavku pro podporu různých SMT solverů (`req_gen_smt`) je do programu integrovaná knihovna *SmtSwitch*[9], která definuje jednotné aplikační rozhraní pro různé SMT solvery (Bitwuzla, cvc5, Z3, MathSAT a Yices2). Bohužel, knihovna je pouze kompatibilní s operačními systémy založenými na Unixu. Z toho důvodu je součástí programu vždy SMT solver z3, který je kompatibilní jak s operačním systémem Windows, tak s operačními systémy založenými na Unixu.

4.4 Backend webové aplikace

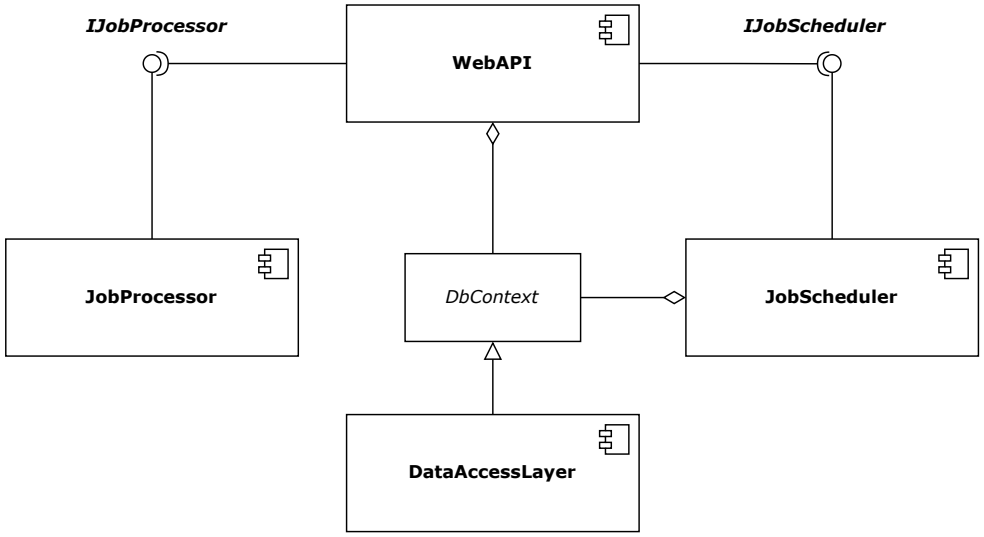
V rámci backendu webové aplikace se klade důraz na popis zprostředkování komunikace mezi klientem a generátorem testovací sady. Popsány jsou jednak komponenty podílející se na zaplánování příchozího požadavku ke zpracování, ale a také komponenty podílející se na překladu příchozí úlohy klienta na úlohu ve formátu pro generátor.

4.4.1 Architektura backendu

Následkem rozhodnutí o odstranění logiky generování testovací sady z webové aplikace do samostatného programu (viz podkapitolu 4.3.1) došlo k výrazné změně role backendu v celém nástroji a jeho zodpovědností. Backend nově musí

[8] Zkratka pro *pointer to implementation*; jedná se o techniku k oddělení rozhraní třídy od její implementace – veřejné rozhraní třídy se skládá pouze z deklarací a ukazatel na konkrétní implementaci třídy je skrytý pomocí privátního ukazatele

[9] https://github.com/stanford-centaur/smt-switch



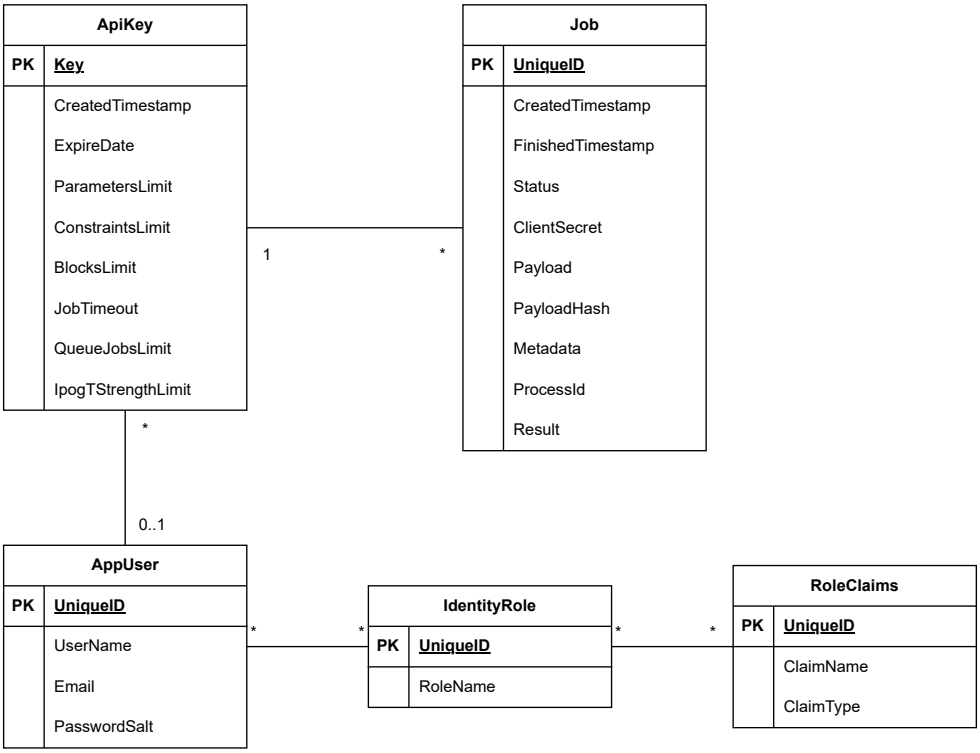
Obrázek 4.3: Schéma architektury komponent backendu webové aplikace.

- přijímat požadavky pro generování testovací sady od klienta a konvertovat je do formátu, který vyžaduje program pro generování testovací sady,
- spouštět generátor jako samostatný proces a spravovat jeho životní cyklus, tj. spuštění, předání dat, přečtení výsledku, ukončení a ošetření výjimek (`req_backend_jobs`),
- zpracovat více příchozích požadavků, resp. spouštět více instancí generátoru, proto, aby všechny ostatní požadavky nebyly blokovány aktuálně zpracovávaným požadavkem (`req_backend_jobs_async`) a
- překládat jazyk pro definici bloků a omezení do jazyka SMT-LIB.

Kromě těchto zmíněných důvodu musí backend zohlednit i bezpečností stránku, např. zamítnutí neidentifikovaných požadavků na generování testovací sady, prevence zahlcení příchozími požadavky, ošetření příliš dlouhého běhu generátoru, zahlcení databáze, atd.

Všechny zmíněné poznatky vedly ke konečnému návrhu, který je znázorněn na obrázku 4.3 a skládá se ze 4 hlavních komponent:

- *WebAPI* – obsahuje kontroléry pro požadavky na generovaní testovací sady, správu uživatelů a rolí. Dále obsahuje logiku identifikaci a autorizaci příchozích požadavků.
- *JobProcessor* – provádí konverzi příchozí úlohy do formátu požadovaného generátorem testovací sady. Obsahuje proto parsery pro všechny datové typy parametrů, parser pro omezení a pro strategie generování testovací sady. Také řeší převod zpět, tj. z výsledku obdrženého od generátoru testovací sady do formátu očekávaného klientem.
- *DataAccessLayer* – zapouzdřuje práci s databází a definuje modely pro objektově relační mapování.
- *JobScheduler* – komponenta spravuje kompletně celý životný cyklus běhu generátoru testovací sady, od naplánovaní úlohy, přes spuštění generátoru testovací sady v novém procesu, jeho ukončení a uložení výsledků.



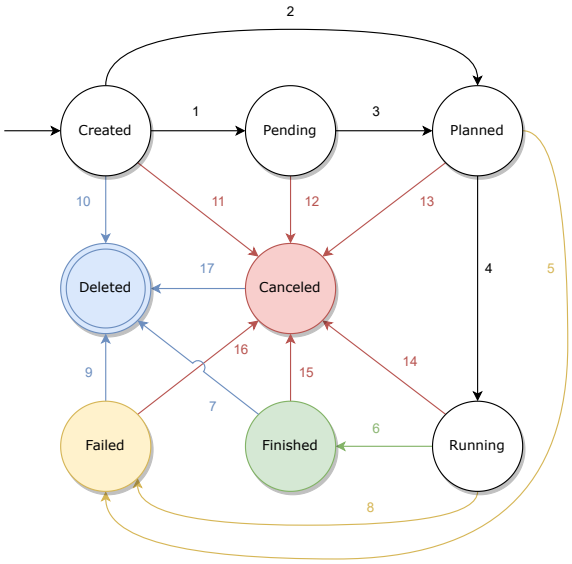
Obrázek 4.4: ER diagram databáze.

4.4.2 Databáze backendu

Backend používá pro perzistentní uložení informací o zpracovávaných nebo již zpracovaných úlohách databázi. Kromě informací o úlohách se v databázi nacházejí tabulky pro vytvořené API klíče a pro správu uživatelů a jejich rolí.

ER diagram databáze zobrazuje obrázek 4.4. Pro úlohu se v databázi uchovávají tyto informace:

- *CreatedTimestamp* – časové razítko vytvoření úlohy.
- *FinishedTimestamp* – časové razítko dokončení úlohy.
- *Status* – aktuální stav úlohy (viz kapitolu 4.5).
- *ClientSecret* – klientův unikátní kód pro rozlišení požadavků se stejným API klíčem (viz kapitolu 4.4.7).
- *Payload* – data zaslaná na vstup generátoru testovací sady.
- *PayloadHash* – spočtený hash pro data zaslaná na vstup generátoru testovací sady.
- *Metadata* – dodatečné interní informace o úloze potřebné pro její pozdější zpracování.
- *ProcessId* – číselný identifikátor procesu generátoru testovací sady.
- *Result* – výsledná data úlohy.



Obrázek 4.5: Životní cyklus požadavku.

Pro API klíč se v databázi uchovávají tyto informace:

- *CreatedTimestamp* – časové razítko vytvoření klíče.
- *ExpireDate* – datum kdy dojde k vypršení platnost klíče.
- *ParametersLimit* – maximální počet parametrů, které mohou obsahovat úlohy zasílané s tímto klíčem.
- *BlocksLimit* – maximální počet omezení.
- *JobTimeout* – Počet milisekund, po jejichž uplynutí dojde k vynucenému ukončení generátoru.
- *QueueJobsLimit* – maximální počet požadavků ve frontě.
- *IpogTStrengthLimit* – maximální hodnota parametru `T` pro algoritmus `IPOG` (viz definici 2.2.4).

Přístup k databázi je umožněn skrze *EntityFrameworkCore*[10], což je .NET framework pro objektově relační mapování. Celou práci s databází zapouzdřuje do třídy *DbContext*, který je následně zpřístupněn všem ostatním komponentám backendu.

4.4.3 Životní cyklus úlohy pro vygenerování testovací sady

Úloha pro vygenerování testovací sady, v databázi reprezentovaná entitou *Job* (viz kapitolu 4.4.2), má několik stavů a přechod mezi nimi ovlivňuje plánovač úloh, spouštěč úloh anebo uživatel. Životný cyklus úlohy je zobrazen na obrázku 4.5 a obsahuje tyto stavy:

[10] https://learn.microsoft.com/en-us/ef/core/

- *Created* – úloha byla úspěšně uložena do databáze.
- *Pending* – úloha čeká ve frontě na zpracování.
- *Planned* – některý z spouštěčů si úlohu vybral ke zpracování.
- *Running* – proces pro generování testovací sady byl úspěšně spuštěn a spouštěč čeká na jeho dokončení.
- *Canceled* – úloha byla zrušena ze strany klienta.
- *Failed* – vytvoření procesu nebo běh procesu skončil chybou.
- *Finished* – proces generování skončil a získaný výsledek byl uložen do databáze. Jakmile se úloha nachází v tomto stavu, klient si může vyžádat její výsledek.
- *Deleted* – záznam úlohy byl z databáze odstraněn.

Následující popis uvažuje konvenci, ve které *Stav* označuje jakýkoliv stav v životním cyklu, *Číslo* označuje jakýkoliv číselný identifikátor přechodu v životním cyklu (viz obrázek 4.5) a platí:

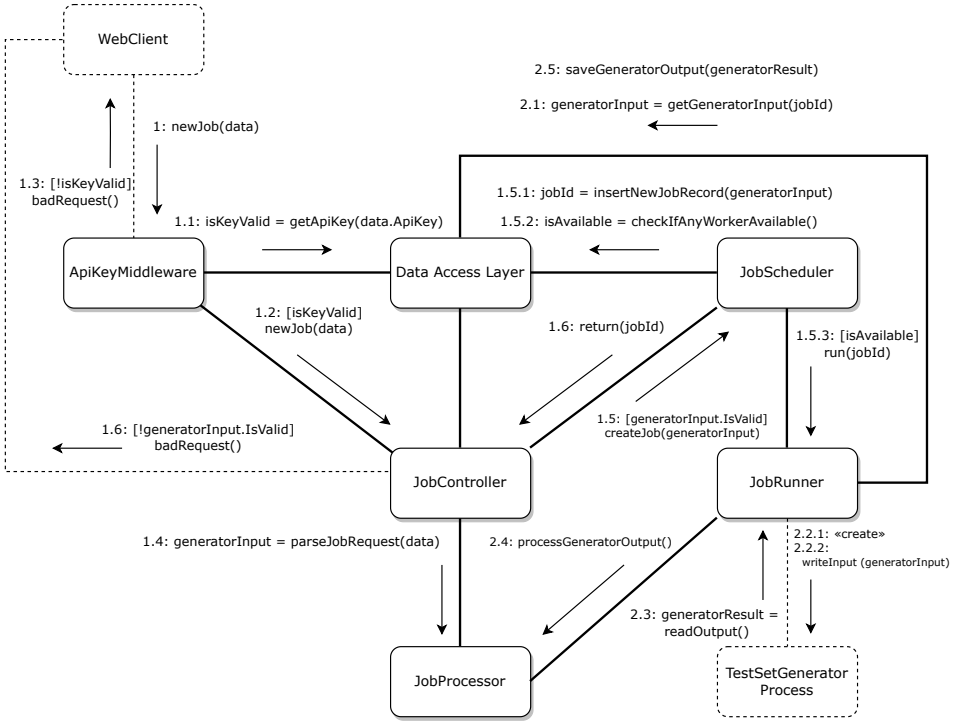
- „do stavu *Stav* (*Číslo*)“ vyjadřuje přechod do stavu *Stav* přes přechod s číselným označením *Číslo*.
- „ze stavu *Stav* (*Číslo*)“ vyjadřuje přechod ze stavu *Stav* přes přechod s číselným označením *Číslo*.

Úloha se po uložení do databáze nachází v počátečním stavu *Created*. Tento počáteční stav slouží pro ošetření kritické části plánovače, kdy může potenciálně dojít k vytvoření dvou a více stejných úloh, protože klient odeslal více stejných požadavků opakovaně za sebou a ve stejnou chvíli byly všechny uloženy do databáze. Proto po uložení úlohy do databáze následuje dotaz na počet existujících úloh v databázi, jejichž *hash* je stejný jako aktuálně uložená úloha. Pokud je takových úloh více, pak pokračuje ke zpracování úloha s nejnižším identifikátorem (pro ošetření této situace se využije vlastnost databáze generovat jednoznačný číselný identifikátor jako primární klíč ve vzestupném pořadí). Úlohy, které jsou duplicitní, a ve zpracování nepokračují, jsou z databáze odstraněny a přechází do stavu *Deleted (10)*.

Úloha přechází se stavu *Created* do stavu *Pending (1)*, pokud plánovač vyhodnotí, že nelze úlohu okamžitě zpracovat, a úlohu je nutné vložit do fronty (viz podkapitolu 4.4.6). Naopak, pokud lze úlohu okamžitě zpracovat, pak přechází ze stavu *Created* do stavu *Planned (2)*. Ze stavu *Pending* do stavu *Planned (2)* přechází úloha v momentě, kdy si ji některý ze spouštěčů zvolí jako další úlohu na zpracování (viz podkapitolu 4.4.6).

Po přechodu do stavu *Planned* začne spouštěč obsluhující úlohu alokokovat potřebné struktury pro spuštění procesu a po jejich alokaci se pokusí o spuštění procesu. Pokud byl proces úspěšně spuštěn, pak přechází úloha do stavu *Running (4)*, v opačném případě je uložena chybová hláška namísto výsledku úlohy a úloha přechází do stavu *Failed (5)*.

Pro úlohu ve stavu *Running* běží proces generující testovací sadu a spouštěč čeká na jeho dokončení. Pokud běh procesu skončí chybou anebo vyprší časový limit pro generování



Obrázek 4.6: Diagram spolupráce hlavních komponent backendu při obdržení nového požadavku.

testovací sady, pak úloha přechází do stavu *Failed (8)* a namísto jejího výsledku je uložena chybová hláška. Pokud proces úspěšně skončí, pak úloha přechází do stavu *Finished (6)*.

Ve všech stavech, kromě koncového stavu *Deleted*, může úloha přejít do stavu *Canceled (11, 12, 13, 14, 15, 16)*, pokud API příjme požadavek od klienta na zrušení úlohy.

Ze stavů *Canceled (17)*, *Finished (7)* a *Failed (9)* přechází úloha do koncového stavu *Deleted* v momentě, kdy dojde k odstranění záznamu úlohy z databáze.

4.4.4 Průběh zpracování úlohy

Tato podkapitola popisuje, jak spolu komponenty komunikují a jaké operace vykonávají při zpracovávání příchozího požadavku na vygenerování testovací sady.

Průběh zpracování úlohy je zobrazen na diagramu spolupráce 4.6 a je tvořen následujícími kroky:

- Webový klient zašle požadavek na vygenerování testovací sady (krok 1). Backendová část příjme požadavek a předtím, než začne zpracovávat jeho data, tak ověří, že pochází ze známého zdroje (krok 1.1). Kontrola se provede ověřením, že uvedený klíč v hlavičce požadavku byl nalezen v databázi a nevypršela jeho platnost (viz podkapitolu 4.4.7). Pokud je klíč validní, pak se data předají dále do kontroléru (krok 1.2). V opačném případě dojde k odmítnutí požadavku (krok 1.3)

- Kontrolér z požadavku extrahuje data a předá je komponentě pro transformaci vstupního požadavku do formátu vyžadovaného programem pro generování testovací sady (krok 1.4). Pokud jsou příchozí data validní a úspěšně se dokončí jejich parsování a transformace, pak kontrolér předá vytvořená vstupní data pro generátor testovací sady do komponenty pro plánování (krok 1.5). Pokud se nepodaří vstupní data zpracovat, pak se klientovi odešle informace o zamítnutí požadavku a o nalezené chybě ve vstupních datech (krok 1.6).
- Plánovač úloh uloží do databáze nově vytvořenou úlohu (krok 1.5.1) a zkontroluje, zda je možné úlohu okamžitě zpracovat (krok 1.5.2), tj. zda lze zadat úlohu pro zpracování novému vláknu, aby zároveň nedošlo k překročení maximálního počtu současně běžících úloh (`req_backend_jobs_max`). Pokud úlohu lze okamžitě zpracovat, pak se její identifikátor předá do nové instance spouštěče úlohy (krok 1.5.3). Nakonec dojde k odeslání identifikátoru úlohy na klienta, aby ho mohl použít pro další dotazování se na API (krok 1.6).
- Spouštěč úlohy si podle předaného identifikátoru načte data pro úlohu z databáze (krok 2.1) a spustí nový proces programu pro generování testovací sady (krok 2.2.1), kterému předá na vstup načtená data (krok 2.2.2). Během běhu procesu naslouchá na standardním výstupu procesu a střádá přijatá dat. Jakmile dojde k úspěšnému skončení procesu (krok 2.3), spouštěč přijatá data předá do komponenty pro transformaci vstupních/výstupních dat generátoru (krok 2.4), a výsledná data z generátoru přemapuje do formátu očekávaného na klientovi. Takto zpracovaná data jsou nakonec uložena do databáze (krok 2.5).

Uloženou vygenerovanou testovací sadu klient obdrží, až si sám o ní požádá, protože komunikace s backendem probíhá asynchronně. Pokud v případě generování testovací daty skončí proces chybou, pak je do databáze uložena přijatá chybová hláška, která je uživateli vrácena namísto vygenerované testovací sady.

4.4.5 Komunikační protokol frontendu a backendu

Komunikace frontendu a backendu probíhá asynchronně, což pro klienta znamená, že na svůj požadavek pro vygenerování testovací sady neobdrží v odpovědi vygenerovanou testovací sadu, ale pouze informaci o tom, že jeho požadavek byl zadán ke zpracování. Aby klient zjistil, v jakém stavu se zpracování úlohy právě nachází, je nutné se opakovaně dotazovat API, dokud klient neobdrží informaci, že byla úloha úspěšně dokončena, anebo že skončila chybou. V takovém případě si může klient vyžádat výsledek úlohy.

Díky asynchronní komunikaci dochází k lepšímu škálování požadavků, protože jejich zpracování je výrazně rychlejší. Naopak, jelikož není celý požadavek zpracován v rámci jednoho sezení, je nutné si informace o příchozích požadavcích ukládat perzistentně.

Kontrolér pro příchozí požadavky má tyto akce:

- Metoda: pro `/job/new` POST, jinak GET.
- Návratové kódy:
  - 200 – příchozí požadavek byl úspěšně zpracován.

  - 400 – dodaná vstupní data jsou špatného formátu.
  - 404 – úloha s daným identifikátorem neexistuje.
  - 500 – při zpracovávání požadavku nastala fatální chyba.
  - 429 – fronta s požadavky od klienta je plná.
- `/job/new` – nový požadavek pro vygenerování testovací sady.
- `/job/status/{id}` – vyžádání stavu úlohy. Pokud je stav Failed/Finished, pak je součástí odpovědi i výsledek úlohy.
- `/job/cancel/{id}` – vyžádá zrušení probíhající úlohy.

Kompletní komunikace mezi frontendem, backendem a generátorem testovací sady při zpracování úlohy je zobrazena pomocí sekvenčního diagramu, který se nachází v příloze A.

4.4.6 Plánovač úloh a spouštěč úloh

V momentě, kdy dojde k úspěšné validaci a přeložení příchozí úlohy od klienta do úlohy pro generátor testovací sady, předá se úloha plánovači úloh. Plánovač po obdržení úlohy provede tyto kroky:

1. Pokusí se najít v databázi úlohu, jejíž vstupní data jsou stejná a daná úloha čeká na zpracování nebo se aktuálně zpracovává. Tento krok je nutný pro vyloučení stejných požadavků od jednoho klienta, jež byly zaslány krátce po sobě. Taková situace může nastat tehdy, kdy odpověď od serveru trvá déle a klient po vypršení časového intervalu zasílá na backend požadavek znovu. Pokud se taková úloha najde, pak se klientovi zašle v odpovědi identifikátor této úlohy a zpracování duplicitní úlohy nepokračuje.
2. Zkontroluje platnost všech podmínek pro vytvoření nových úloh a zadání nových úloh ke zpracování. Aktuálně se kontroluje, zda:
   - Je možné spustit další proces generátoru testovací sady – maximální počet současně běžících procesů se definuje při startu webového API. Tento počet není možné překročit (aby nedošlo k vyčerpání prostředků systému).
   - Pokud není bod č. 1 splněn a aktuálně nelze spustit další proces, pak se zkontroluje, zda je možné požadavek na zpracování vložit do fronty požadavků.
3. Pokud nelze splnit ani jednu ze zmíněných podmínek, pak je požadavek odmítnut s informací, aby uživatel opakoval akci později. Velikost fronty požadavků je stejně jako maximální počet současně běžících procesů omezená na maximální počet pro daný API klíč (viz kapitola 4.4.2).
4. Pokud je splněna první podmínka, pak se úloha uloží do databáze a spustí se nové vlákno v podobě instance třídy *Spouštěče úloh* (angl. *JobRunner*), které je předán identifikátor vytvořené úlohy. Vlákno pak tuto úlohu zpracuje přednostně.
5. Při splnění druhé podmínky dojde opět k uložení požadavku do databáze, ale ke spuštění nového vlákna nedojde. Danou úlohu si z databáze později odbaví některý z aktuálně běžících spouštěčů. Úlohy jsou z fronty odbavovány v pořadí, ve kterém byly do databáze uloženy.

Spouštěč úloh při svém startu buď přednostně zpracuje úlohu, jejíž identifikátor obdržel, anebo vyhledá v databázi úlohy čekající ve frontě a vybere si ke zpracování tu nejstarší. Následně spustí proces generátoru testovací sady, předá mu na vstup data úlohy a poté čeká na jeho ukončení, přičemž zároveň čte a ukládá si výstup procesu na jeho standardním výstupu a standardním chybovém výstupu.

Jakmile dojde k úspěšnému ukončení procesu, pak spouštěč obdržený výsledek předá ke zpracování komponentě pro přeložení výsledku do formátu očekávaného klientem. Nakonec spouštěč výsledek uloží do databáze.

Pokud proces skončí chybou anebo vyprší stanovený časový limit pro maximální dobu běhu procesu, pak spouštěč uloží získanou chybovou hlášku do databáze.

Po dokončení zpracování úlohy pokračuje kontrolou úloh ve frontě. Jestliže žádná úloha ve frontě nečeká, vlákno se ukončí.

4.4.7 Bezpečnostní prvky API

Pro backend bylo navrhnuto a implementováno několik bezpečnostních prvků, které mají jednak zabránit neidentifikovaným požadavkům v dotazování se na API, ale také těm identifikovaným zabránit ve vyčerpání všech dostupných zdrojů systému. Některé tyto bezpečností prvky již byly zmíněny a popsány (např. maximální velikost fronty požadavků a maximální počet současně běžících procesů generátoru v podkapitole 4.4.6).

Ačkoliv původní implementace žádné takové bezpečností opatření neobsahuje a za dobu jejího vystavení na Internetu nedošlo k žádnému zneužití služby, je vhodné tuto oblast brát v potaz již v návrhu, aby bylo API připraveno na budoucí případy užití, tj. kdy má být využito pro projekt *VALU3S* (viz kapitolu 1), čímž se stane potenciálně kritičtější službou.

Identifikace skrze API klíč

Při popisu zpracování úlohy v kapitole 4.4.4 dochází v prvním kroku k identifikaci požadavku skrze API klíč. Při návrhu backendu bylo rozhodnuto (`req_backend_api_keys`), že každý příchozí požadavek musí mít validní a platný API klíč. Díky tomu, že je každý požadavek rozlišen podle API klíče, pak lze určit pro zpracování požadavku různé nastavení. Aktuálně API klíč určuje (viz podkapitolu 4.4.2):

- Maximální počet parametrů, které mohou obsahovat požadavky zasílané s tímto klíčem.
- Maximální počet omezení,které mohou obsahovat požadavky zasílané s tímto klíčem.
- Počet milisekund, po jejichž uplynutí dojde k vynucenému ukončení generátoru.
- Maximální počet požadavků ve frontě.
- Maximální hodnota parametru `T` pro algoritmus `IPOG` (viz definici 2.2.4).

Díky tomu lze uživatelům přidělovat různé klíče s různě definovaným nastavením pro zpracování požadavků podle jejich potřeb. Tento případ užití vyplývá i z požadavku, kdy by

mělo API sloužit pro generování testovací sady pro nástroj *netloiter*. Díky tomu lze přidělit požadavkům od tohoto nátroje třeba delší dobu na generování testovací sady, než požadavkům přicházejícím z frontendu, kterých může být potenciálně velké množství a mohly by vyčerpat dostupné zdroje systému.

Tajemství klienta

Z důvodu možného sdílený jednoho API klíč vícero uživateli, jak je to například pro webovou část nástroje, ve které jsou všechny požadavky odesílány pod stejným API klíčem, je nutné na klienta odesílat nejen identifikátor vytvořené úlohy, ale i vygenerovaný unikátní klíč jako další tajemství klienta, který bude sloužit pro další identifikaci uživatele v rámci všech uživatelů jednoho API klíče.

Bez tohoto tajemství by se mohli uživatelé se společným API klíčem dotazovat na API na úlohy, jejichž autory nejsou oni, ale někdo jiný se sdíleným API klíčem. Klient si proto musí toto tajemství uchovat a odesílat ho společně s požadavky na API. Pokud uživatel toto tajemství ztratí, pak ztratí i možnost se dotázat na již zpracované anebo stále zpracovávané úlohy, které byly pod tímto tajemstvím vytvořeny.

4.4.8 Procesor úloh

Úkolem této komponenty je překlad příchozího požadavku od klienta do formátu vyžadovaného generátorem testovací sady (viz podkapitolu 4.3.1) a následně výsledku obdrženého od generátoru do formátu vyžadovaného klientem.

Příchozí požadavek od klienta, jehož formát zobrazuje výpis 4.4.8, je nutné

- zvalidovat a přeložit omezení do jazyka SMT-LIB,
- zvalidovat a přeložit definice bloků parametrů do jazyka SMT-LIB,
- zvalidovat požadované datové typy parametrů a definici strategie pro generování testovací sady.

Při návrhu této komponenty byl především kladen důraz na rozšiřitelnost, aby bylo možné přidávat podporu dalších datových typu či dalších strategií pro generování. Konečný návrh je zobrazen na obrázku 4.7 a skládá se z

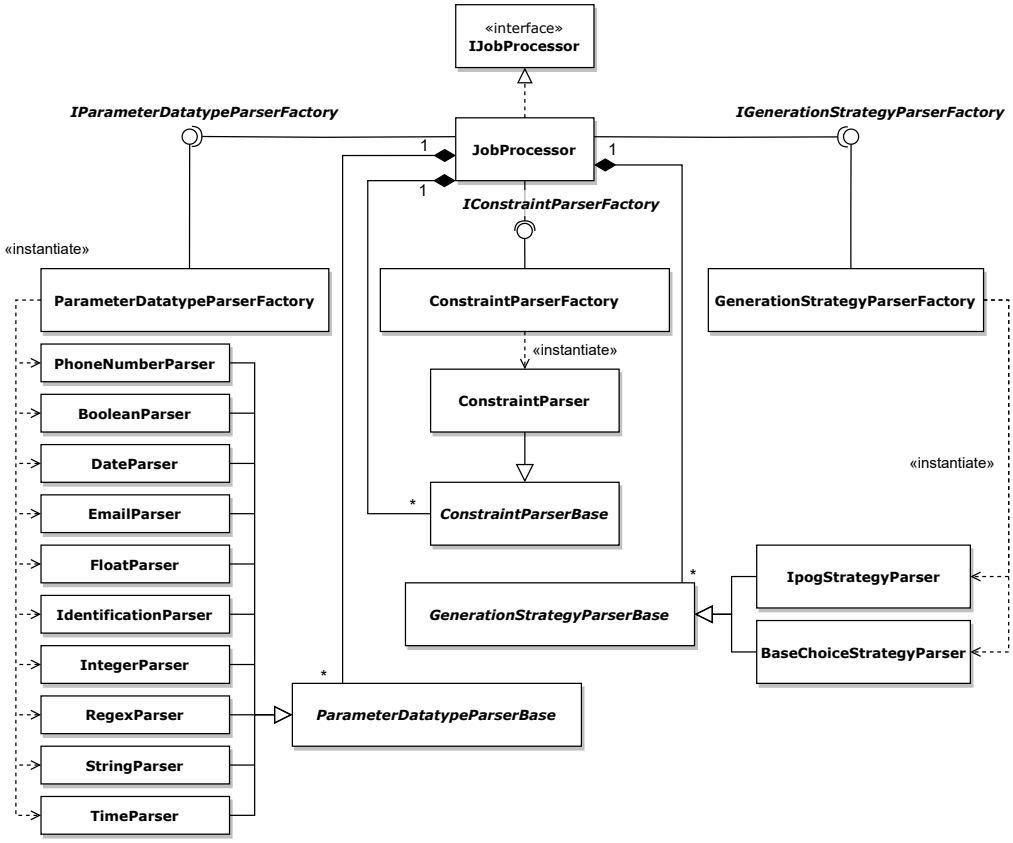
- *JobProcessor* – třída řídící překlad. Instanci třídy jsou předána data z příchozího požadavku, ta je postupně zpracovává a zasílá zodpovědným továrním metodám, které vytváří instance parseru odpovídající příchozím datům. Vytvořené instance parseru předává zpět instanci třídy JobProcess, která invokuje jejich parsovací metody nad příchozími daty a ukládá jejich výsledky. Třída je závislá na rozhraních všech továrních metod, nikoliv na jejich konkrétních implementacích. Konkrétní objekty implementující daná rozhraní přijímá instanci třídy JobProcessor při vytvoření své instance, tedy je zde uplatněn návrhový vzor *vkládání závislostí* (angl. *Dependency injection*)[11].

[11] Návrhový vzor pro oddělení vytváření objektů a jejich závislostí od samotné logiky tříd. Namísto toho, aby třída vytvářela a vlastnila objekty, které potřebuje, jsou tyto objekty injektovány do třídy zvenčí.

```
{
    "GenerationStrategy": {
        "GenerationStrategy": "Ipog",
        "StrategyConfig": {
            "T": 3
        }
    },
    "OnlyIndices": true,
    "Constraints": [{ "Definition": "a.1 -> b.2" }
    ],
    "Parameters": [
        {
            "Name": "a",
            "Datatype": "Integer",
            "Blocks": [{ "Definition": "(< a 20))" }, { "Definition": "(>= a 20)" }
            ]
        },
        {
            "Name": "b",
            "Datatype": "Integer",
            "Blocks": [{ "Definition": "}(< b 30))" }, { "Definition": "(>= b 30)" }
            ]
        }
    ]
}
```

Výpis 4: Formát požadavku od klienta.

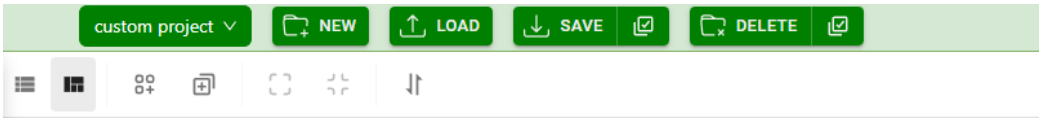
- *ParameterDatatypeParserFactory* – tovární metoda pro instanciaci tříd pro parsování a překlad definic parametrů a jejich bloků do SMT-LIB jazyka.
- *ConstraintParserFactory* – tovární metoda pro instanciaci tříd pro parsování a překlad definicí omezení do SMT-LIB jazyka.
- *GenerationStrategyParserFactory* – tovární metoda pro instanciaci tříd pro validaci a zpracování definic strategií pro generování testovací sady.
- *ParameterDatatypeParserBase* – abstraktní třída ze které dědí všechny třídy pro parsování definic parametrů a jejich bloků.
- *GenerationStrategyParserBase* – abstraktní třída ze které dědí všechny třídy pro zpracování definici strategií.
- *ConstraintParserBase* – abstraktní třída ze které dědí všechny třídy pro parsování definic omezení.
- *ConstraintParser* – třída pro parsování aktuálně podporovaného formátu definic omezení.
- *IpogStrategyParser*, *BaseChoiceStrategyParser* – třídy pro zpracování definice pro Ipog a BaseChoice strategie generování testovací sady.
- *DateParser*, *EmailParser*, *IntegerParser*, *FloatParser*, ... – třídy pro parsování konkrétních datových typů parametrů a jejich bloků



Obrázek 4.7: Architektura procesoru úloh.

4.5 Frontend webové aplikace

V této kapitole jsou popsány hlavní prvky nového uživatelského rozhraní a jejich hlavní funkce.

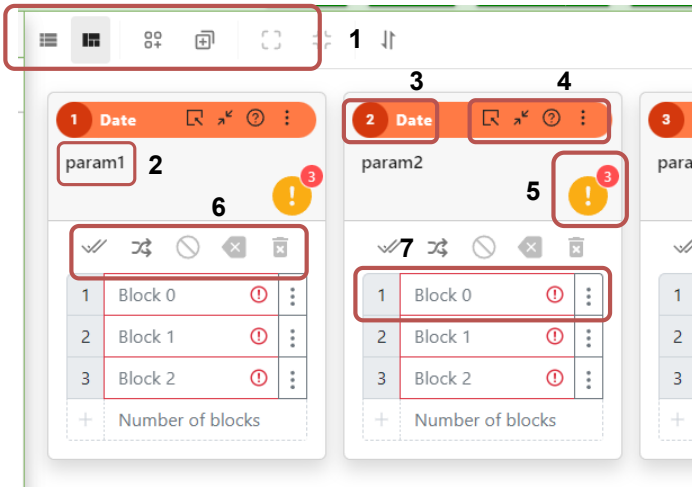


Obrázek 4.8: Uživatelské prvky pro ovládání projektů.

Parametry a omezení jsou nově sdružovány do projektů. Ty je možné vytvářet, načítat, ukládat a mazat pomocí ovládacích prvků nacházejících se v horní liště, tak jak je ukázáno na obrázku 4.8.

Parametry jsou zobrazeny pomocí dlaždic (viz obrázek 4.9) a obsahují

1. panel s akcemi pro vytváření, mazání, řazení nebo vybírání parametrů,
2. textové pole pro editaci jména,
3. zobrazení aktuálního datového typu a po rozkliknutí množnost změny na jiný datový typ,



Obrázek 4.9: Ovládání parametru a jeho bloků.

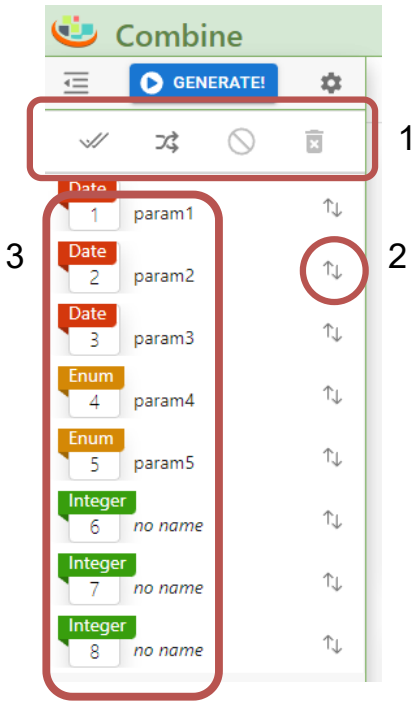
4. akce pro minimalizaci/maximalizaci dlaždice, rychlé zobrazení nápovědy pro daný datový typ,
5. indikace na přítomnost chyby v blocích parametru,
6. akce pro hromadnou selekci, deselekci, mazání, přidávání bloků,
7. textové pole pro zápis bloku a kontextové akce pro každý blok.

Postranní menu, zobrazené na obrázku 4.10, se nachází na levé straně, obsahuje seznam všech definovaných parametrů a

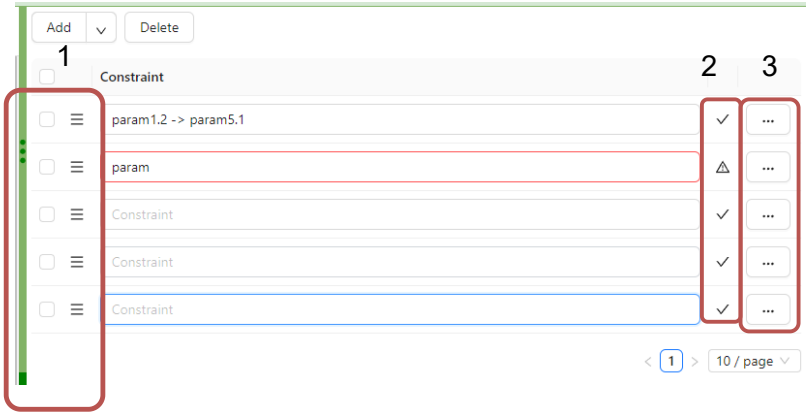
1. akce pro selekci, deselekci, mazání,
2. možnost změny pořadí,
3. orientaci podle datového typu a pořadí parametru.

Tabulka s definicemi omezení na obrázku 4.11 se nachází na pravé straně uživatelského rozhraní a umožňuje kromě zápisu omezení také

1. hromadně vybírat a měnit jejich pořadí,
2. upozornit uživatele, pokud je omezení definováno chybně,
3. manipulovat s omezeními pomocí kontextových akcí.



Obrázek 4.10: Levé menu se seznamem definovaných parametrů.



Obrázek 4.11: Tabulka s omezeními.

Kapitola 5

Závěr

Cílem této diplomové práce byla kompletní analýza nástroje Combine, nalezení jeho nedostatků, navrhnutí způsobu jejich odstranění a přidání další funkcionality do nástroje, která by zvýšila jeho kvalitu.

Po podrobné analýze nástroje bylo usouzeno, že je nutná reimplementace celého nástroje, tj. jak jeho uživatelského rozhraní, tak i webového API zahrnující logiku pro generování testovací sady. Pro reimplementaci byly zvoleny vhodnější technologie, lepší návrh architektury a navrženy nové funkce, které předchozí řešení postrádalo.

Nové řešení významně nabylo na komplexnosti, čímž došlo i k výraznému zpomalení vývoje a vynechání implementace některých požadavků, které by byly vhodným rozšířením původního nástroje. Vytvořené řešení proto značně strádá na jednotkových testech, není implementovaná webová administrace a nebyly provedeny výkonnostní testy pro porovnání nové implementace algoritmu `FIPOG` s implementací původní.

Nové řešení i přesto, že staví na lepších technologiích a celkovém návrhu, tak není připravené na nasazení a je nutné ho ještě doladit.

Literatura

[1] Ericsson, S. a Enoiu, E. Combinatorial Modeling and Test Case Generation for Industrial Control Software Using ACTS. In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2018, s. 414–425. DOI: 10.1109/QRS.2018.00055.

[2] Gargantini, A. a Radavelli, M. Migrating Combinatorial Interaction Test Modeling and Generation to the Web. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. April 2018, s. 308–317. DOI: 10.1109/ICSTW.2018.00066.

[3] Grindal, M., Lindström, B., Offutt, J. a Andler, S. F. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*. Dec 2006, sv. 11, č. 4, s. 583–611. DOI: 10.1007/s10664-006-9024-2. ISSN 1573-7616. Dostupné z: https://doi.org/10.1007/s10664-006-9024-2.

[4] Kacker, R. N., Richard Kuhn, D., Lei, Y. a Lawrence, J. F. Combinatorial testing for software: An adaptation of design of experiments. *Measurement*. 2013, sv. 46, č. 9, s. 3745–3752. DOI: https://doi.org/10.1016/j.measurement.2013.02.021. ISSN 0263-2241. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0263224113000596.

[5] Khalsa, S. K. a Labiche, Y. An Orchestrated Survey of Available Algorithms and Tools for Combinatorial Testing. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, s. 323–334. DOI: 10.1109/ISSRE.2014.15.

[6] Kleine, K., Kotsireas, I. S. a Simos, D. E. Evaluation of Tie-Breaking and Parameter Ordering for the IPO Family of Algorithms Used in Covering Array Generation. In: *IWOCA*. 2018.

[7] Kleine, K. a Simos, D. E. An Efficient Design and Implementation of the In-Parameter-Order Algorithm. *Mathematics in Computer Science*. Mar 2018, sv. 12, č. 1, s. 51–67. DOI: 10.1007/s11786-017-0326-0. ISSN 1661-8289. Dostupné z: https://doi.org/10.1007/s11786-017-0326-0.

[8] Kolárik, T. *SAT s diferenciálními rovnicemi*. Praha, CZ, 2018. Diplomová práce. České vysoké učení technické v Praze. Vypočetní a informační centrum. Dostupné z: http://hdl.handle.net/10467/76369.

[9] Kotyz, J. *Nástroj pro tvorbu obsahu databáze pro účely testování software*. Brno, CZ, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/18066/.

[10] Kropáč, F. *Nástroj pro analýzu obsahu databáze pro účely testování softwaru*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19446/.

[11] Kuhn, D., Wallace, D. a Jr, A. Software Fault Interactions and Implications for Software Testing. *Software Engineering, IEEE Transactions on*. Červenec 2004, sv. 30, s. 418 – 421. DOI: 10.1109/TSE.2004.24.

[12] Kuhn, D., Kacker, R. a Lei, Y. *Practical Combinatorial Testing*. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2010-10-07 2010. DOI: https://doi.org/10.6028/NIST.SP.800-142.

[13] Matějka, T. *Kombinatorické generování testů*. Plzeň, CZ, 2019. Diplomová práce. Západočeská univerzita v Plzni. Dostupné z: http://hdl.handle.net/11025/37422.

[14] MILATA, M. *Verifikace konečných systémů pomocí SMT-solveru*. 2010 [cit. 2022-01-29]. Dostupné z: https://is.muni.cz/th/ek0l1/.

[15] Naňo, A. *Frontend pro generátor testovacích dat*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21378/.

[16] Nie, C. a Leung, H. A Survey of Combinatorial Testing. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. feb 2011, sv. 43, č. 2. DOI: 10.1145/1883612.1883618. ISSN 0360-0300. Dostupné z: https://doi.org/10.1145/1883612.1883618.

[17] Oberreiter, M. *Kontejnerizace detektorů nad relačními databázemi*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20386/.

[18] Ochodek, M. *Nástroj pro analýzu obsahu databáze pro účely testování softwaru*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19259/.

[19] Oháňka, M. *Automatizovaná detekce datových typů ve strukturách*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21837/.

[20] Olšák, O. *Generování strukturovaných testovacích dat*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21539/.

[21] Rychnovský, J. *Verifikovaná knihovna datových struktur*. Brno, CZ, 2013. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/14713/.

[22] Skupina Testos. *Domovská stránka projektu Testos*. [online]. 2018 [cit. 13.04.2020]. Dostupné z: http://testos.org/.

[23] Sušovský, T. *Generování testovacích vstupů podle stopy programu*. Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22088/.

[24] Červinka, R. *Asistent pro generování testovacích scénářů*. Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/20303/.

[25] Želiar, D. *Automatizovaná syntéza stromových struktur z reálných dat*. Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22217/.

[26] Wagner, M., Kleine, K., Simos, D. E., Kuhn, R. a Kacker, R. CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, s. 191–200. DOI: 10.1109/ICSTW50294.2020.00041.

[27] Wu, H., Nie, C., Petke, J., Jia, Y. a Harman, M. A Survey of Constrained Combinatorial Testing. *ArXiv*. 2019, abs/1908.02480.

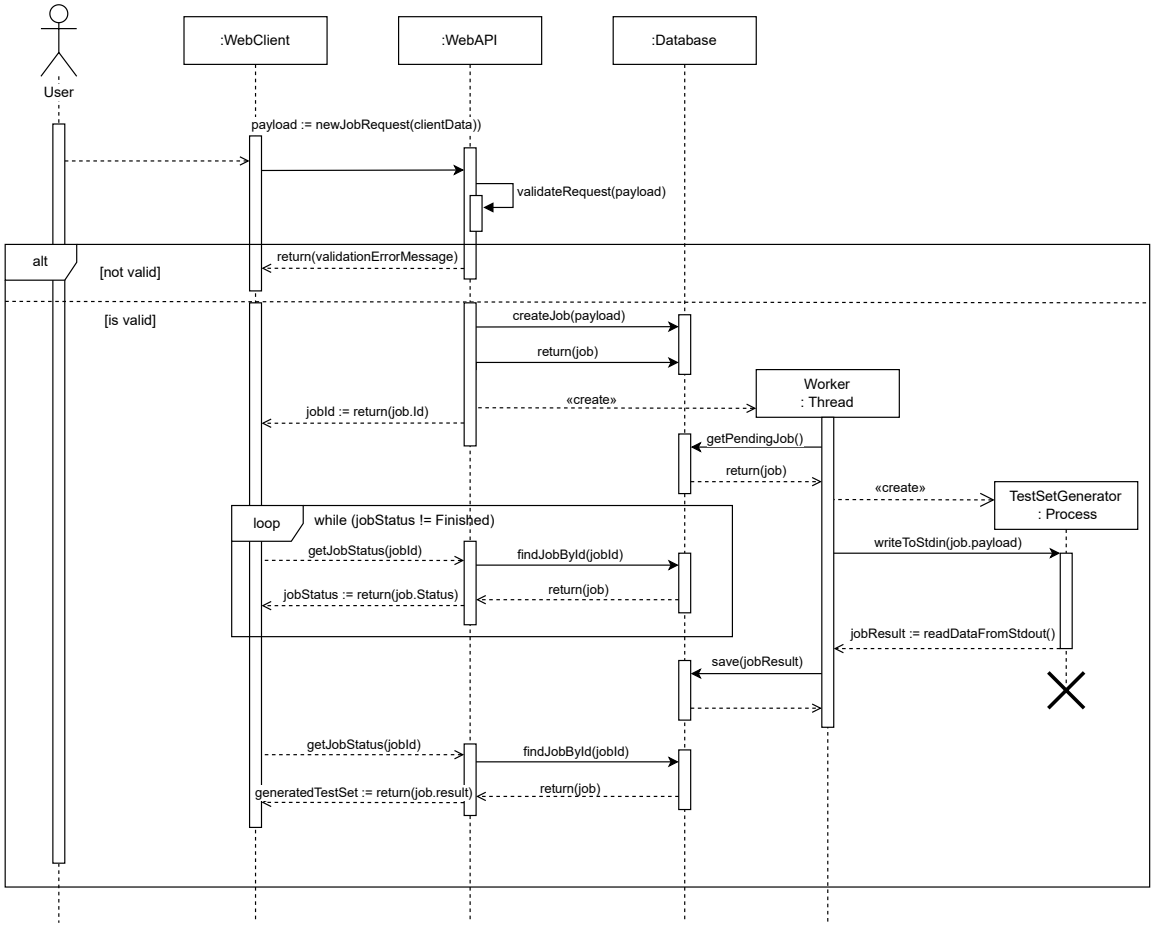
[28] Wu, H., Nie, C., Petke, J., Jia, Y. a Harman, M. Comparative Analysis of Constraint Handling Techniques for Constrained Combinatorial Testing. *IEEE Transactions on Software Engineering*. 2021, sv. 47, č. 11, s. 2549–2562. DOI: 10.1109/TSE.2019.2955687.

[29] Yilmaz, C., Fouché, S., Cohen, M. B., Porter, A., Demiroz, G. et al. Moving Forward with Combinatorial Interaction Testing. *Computer*. 2014, sv. 47, č. 2, s. 37–45. DOI: 10.1109/MC.2013.408.

[30] Yu, L., Lei, Y., Kacker, R. N. a Kuhn, D. R. ACTS: A Combinatorial Test Generation Tool. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, s. 370–375. DOI: 10.1109/ICST.2013.52.

Příloha A

Sekvenční diagram komunikace webové aplikace a generátoru



Obrázek A.1: Sekvenční diagram komunikace frontendu, backendu a generátoru testovací sady.