VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

APLIKACE ZALOŽENÉ NA GENERÁTORU TESTOVACÍCH PŘÍPADŮ

APPLICATIONS BASED ON GENERATOR OF TEST CASES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VALERIIA LEONTEVA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JOSEF STRNADEL, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání bakalářské práce

Studentka: **Leonteva Valeriia**
Program: Informační technologie
Název: **Aplikace založené na generátoru testovacích případů**
Applications Based on Generator of Test Cases
Kategorie: Modelování a simulace

Zadání:

1. Proveďte detailní rešerši v oblasti nástrojů pro generování testovacích případů a jejich aplikací. Shrňte vlastnosti nástrojů, analyzujte jejich dostupnost a pro další práci si zvolte vhodný a dostupný nástroj.
2. Detailně se seznamte se zvoleným nástrojem; způsob práce s ním ověřte a zdokumentujte pomocí vhodně zvolené sady systémů.
3. Zvolte alespoň dvě různé aplikace založené na generátoru testovacích případů (např. testování založené na pokrytí hran, stavů či dvojic definice-užití) a alespoň tři různé systémy (např. z kategorií software, hardware, algoritmus, protokol), na které se budou testovací případy aplikovat.
4. Diskutujte a navrhněte prostředky a způsob implementace modelů zvolených systémů, generování testovacích případů a aplikací založených na generátoru.
5. Modely zvolených systémů implementujte, vygenerujte pro ně příslušné testovací případy a tyto případy vhodně ověřte ve zvolených aplikacích.
6. Zhodnoťte dosažené výsledky, přínos generátoru pro praxi a navrhněte možné směry dalšího využití generátorů testovacích případů.

Literatura:

- Hessel A., Larsen K.G., Nielsen B., Pettersson P., Skou A. (2004) Time-Optimal Real-Time Test Case Generation Using Uppaal. In: Petrenko A., Ulrich A. (eds) Formal Approaches to Software Testing. FATES 2003. Lecture Notes in Computer Science, vol 2931. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-24617-6_9.

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

Vedoucí práce: **Strnadel Josef, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této bakalářské práce je vytvořit přehled aktuálního stavu nástrojů, které umožňují automaticky generovat testovací případy. Dále na příkladu zvoleného nástroje ukázat způsob práce s ním a jeho schopnosti v generaci spustitelných testu. Pro následující práci byl vybrán nástroj UPPAAL, který umožňuje vytvořit model zvoleného systému pomocí časovaných automatů, ověřit a simulovat jeho běh a následně vytvořit testovací případ pro daný systém. Ve výsledku nástroj vygeneruje cestu průchodu systémem, kterou je možné uložit ve formě spustitelného testovacího případu a to v libovolném programovacím jazyce. Pro testování byly zvoleny tři různé systémy: systém vypínače světla, implementovaný v jazyce Java; 2-bitová násobička, jejíž chování je popsané pomocí jazyka Verilog; a systém zjednodušeného výtahu, který je představen v jazyce C. Ve výsledku byly získány spustitelné testovací případy pro zvolené systémy, spolu s jejich vlastnostmi jako pokrytí systému, počet kroků pro dosažení cílů a kvalita vygenerovaných cest.

Abstract

The aim of this bachelor's thesis is to create an overview of the current state of tools that allow automatic generation of test cases and select one tool to show how it works and its ability to generate executable tests. The UPPAAL program was chosen for the following work. Tool allows to create a model of the selected system using timed automata, verify and simulate its operation and create a test case for the system. In the results, the tool generates a path through the system, which can be saved in the form of executable test cases in any programming language. Three different systems were chosen for testing: a light switch system implemented in Java; 2-bit multiplier, which behavior is described by Verilog language; and a simplified elevator system, which working process is introduced in C language. As a result, executable test cases were obtained for selected systems along with their features such as system coverage, number of steps to achieve goals, and quality of generated paths.

Klíčová slova

Testování, testovací případ, generátor testovacích případů, pokrytí, UPPAAL.

Keywords

Testing, test case, test case generation, coverage, UPPAAL.

Citace

LEONTEVA, Valeriia. *Aplikace založené na generátoru testovacích případů.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Aplikace založené na generátoru testovacích případů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

.....
Valeriia Leonteva
11. května 2021

Poděkování

Ráda bych poděkovala panu Ing. Josefu Stnadelovi, Ph.D. za cenné rady, ochotu, odborné připomínky a trpělivost při vedení mé bakalářské práce. Rovněž děkuji rodině a přátelům za podporu.

Obsah

Kapitola 1

Úvod

Testování je jedním z nejdůležitějších procesů životního cyklu vývoje softwaru. Tento proces zkoumá, zda software splňuje specifikované či implicitní potřeby uživatelů. Výzkum ukazuje, že v komerčních firmách proces ladění, testování a verifikace softwaru se pohybuje mezi 50 a 70 % celkových nákladů na vývoj [19]. Samotný proces testování vynakládá 30-40 % celkového úsilí. Hlavním důvodem velkých nákladů je to, že testování je součástí všech etap vývoje softwaru. Díky tomu je zajištěna rychlá oprava chyb během jednotlivých fází procesu vytváření programu. V případě, že proces testování probíhá pouze v poslední fázi vývoje softwaru nebo až po jeho nasazení do provozu, může nastat situace, že se objeví chyba, kterou bylo možné snadno a rychle opravit ještě během první fáze vývoje programu. Teď je ale potřeba změnit velkou část systému, která navazuje na místo s danou chybou, což způsobí značné náklady pro její opravu.

Obvykle proces testování probíhá ručně, tzn. software je testován člověkem. Takový způsob dovoluje sledovat reakci testovaného programu v reálném čase při různých podmínkách použití. Jedna z hlavních nevýhod tohoto způsobu je situace, kdy je potřeba ověřit správnost nějaké části programu a to opakovaně - např. pokud je potřeba ověřit správnost stisknutí tlačítka a reakce na stisk, člověk je nucen opakovaně zmáčknout dané tlačítko n-krát. V tomto případě je lepší optimalizovat proces testování, a to pomocí zavedení automatického testování. V posledních letech se objevilo velké množství nástrojů, které umožňují generovat a opakovaně aplikovat testovací případy na testovaný systém.

Cílem této bakalářské práce je vytvořit přehled existujících nástrojů pro automatické generování testovacích případů a ukázat způsob prací z jedním z vybraných nástrojů. Při vyhledávání byl kladen velký důraz na aktuálnost nástrojů a možnost generování spustitelných testů pro různé programovací jazyky.

V následující kapitole se zaměříme na vysvětlení pojmu testování a jeho vlastnosti. Zároveň bude čtenář seznámen s pojmem testovací případ a způsobem jeho generování. Na závěru kapitoly bude ukázán přehled aktuálních nástrojů a jejich krátký popis, který zahrnuje princip testování, vstupní a výstupní data a čím se liší od ostatních nástrojů. V kapitole 3 bude popsán způsob práci se zvoleným nástrojem a jeho hlavní součástky. V této bakalářské práci byl zvolen nástroj UPPAAL, který podporuje offline generování testovacích případů a umožňuje výsledný spustitelný testovací případ popsat v jakémkoliv programovacím jazyku. V kapitole 4 budou ukázány zvolené modely, na které budou aplikovany testovací případy a jejich implementace v nástroji UPPAAL. Spolu s tím bude uvedeno, jakým způsobem budou implementovány testovací případy a přehled problémů, které vznikly během implementace. V kapitole 5 budou představeny jednotlivé experimenty a získané výsledky.

Kapitola 2

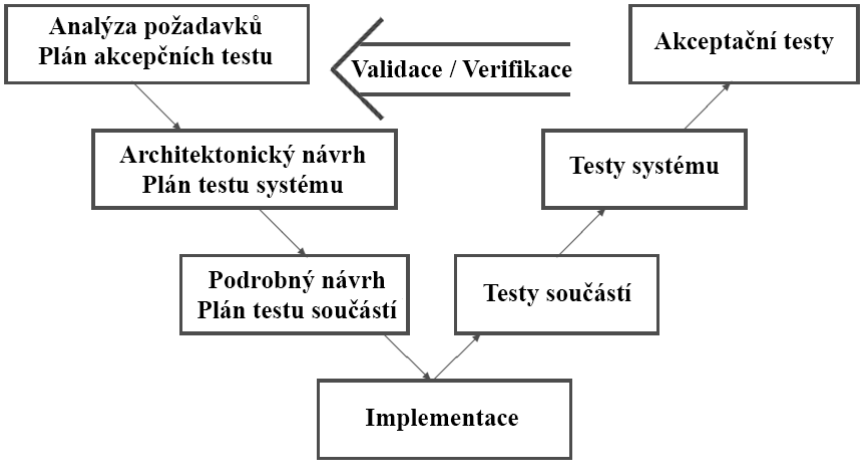
Definice a rešerše k řešené problematice

Cílem této kapitoly je uvedení pojmu testování, jeho role v životním cyklu vývoje softwaru a dělení dle různých kritérií. Poté se čtenář seznámí s pojmem testovací případ, jeho aplikací a způsobem stanovení jeho efektivnosti. Tyto informace jsou potřebné pro lepší pochopení principů a činností nástrojů pro generování testovacích případů, které jsou uvedeny na konci kapitoly.

2.1 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru (angl. Software Development Life Cycle, SDLC) je proces zaměřený na vytváření a udržování funkčnosti, kvality a spolehlivosti softwaru. Tento proces se skládá z několika etap:

- Analýza a specifikace požadavků: získání, analýza a definování požadavků uživatele. Následně jsou tyto požadavky přiváděny z neformálního popisu do strukturovaného popisu požadavků. Součástí této etapy je provedení studia vhodnosti, identifikace a analýza rizika a plán akceptačního testování.
- Návrh softwaru: ujasnění koncepce systému, jeho dekompozice, definování vztahů mezi částmi systému, podrobná specifikace softwarové části (stanovení fyzické struktury dat a způsoby ošetřování neočekávaných a chybových stavů), návrh testování celého systému a jeho jednotlivých součástí včetně testovacích dat.
- Implementace: implementace jednotlivých částí systému a jejich testování.
- Integrace a testování: spojení součástí do podsystémů, testování podsystémů, integrace podsystému do celého systému a následné testování podsystémů a celého systému. V případě nalezení chyb, je potřeba je opravit a vrátit se k etapě implementace.
- Akceptační testování a instalace: testování systému uživatelem, nasazení systému.
- Provoz a údržba: průběžné řešení problémů a chyb, které se objeví až za běhu softwaru.



Obrázek 2.1: V-model.

2.2 Testování

Testování softwaru je kontrola shody mezi skutečným a očekávaným chováním programu. Provádí se na konečné sadě testů, vybraných určitým způsobem [6]. Hlavním účelem testování je hledání chyb, tedy nesrovnalostí, mezi pozorovaným chováním softwaru a požadavkem na software. Zároveň samotný proces testování by měl sledovat celkovou kvalitu programu a zda software splňuje všechny požadavky a potřeby, které byly uvedeny ve specifikaci. Kromě detekce chyb, by měly být poskytnuty informace o tom, kde byly chyby nalezeny, jejich závažnost, a dále také celková spolehlivost a správnost testovaného systému (angl. System Under Test), informace o stabilitě jednotlivých funkcí a komponent. Testování by mělo zajistit, s ohledem na budoucí změny v softwaru, stabilní vývoj systému. Dále by mělo poskytovat nástroje pro sledování změn a predikci možných problémů po zavedení těchto změn.

2.2.1 Základní klasifikace

Automatické a manuální testování

V praxi se testování skládá z provedení určitých akcí v testované aplikaci, získání výsledků z těchto akcí a jejich porovnání s údaji, které jsou definované jako referenční. Tento proces lze provést buď ručně (testy jsou prováděny člověkem) nebo automaticky, za využití různých softwarových nástrojů. Hlavními funkcemi automatických testovacích nástrojů jsou spuštění programu, inicializace, provádění jednotlivých kroků testů, analýza výsledků a jejich zobrazení.

Automatické testování je vhodné zavést v případě testů, které se často opakují, jsou neměnné a pokrývají části programu, které se často nemění. Zároveň je vhodné využít automatizovaný nástroj na té části softwaru, která již prošla manuálním testováním. V případě změn je potřeba zkontrolovat konfigurace nebo upravit testovací skript, jelikož většina nástrojů neumí reagovat na jakékoliv modifikace v testovaném programu. Tento druh testování nevylučuje ruční testování z procesů vývoje softwaru, ale výrazně snižuje jeho podíl v případě opakovaného testování části programu, který není změněn během delší doby. Au-

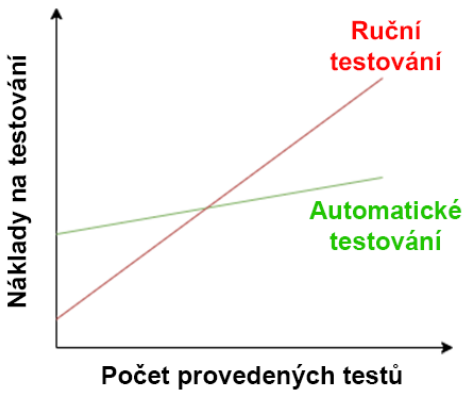
tomatické testování dále snižuje náklady a zlepšuje kvalitu vyvíjených produktů. Využití takových nástrojů má své výhody i nevýhody.

Mezi pozitiva automatického testování patří:

- Rychlost provádění testů.
- Zmenšení nákladů po zavadení automatického testování.
- Schopnost provádět stejné testy vícekrát.
- Možnost provádění složitých testovacích případů.
- Schopnost ukládat a analyzovat velké množství dat.
- Nepřítomnost lidského faktoru (nesoustředěnost, únava), které mohou ovlivnit proces testování.

Mezi negativa automatického testování patří:

- Náklady na začátku zavedení procesu automatického testování.
- Vývoj testovacích sad a jejich údržba během rozvoje softwaru.
- Zastaralost testů v případě změn požadavků nebo revize některých součástek testovaných produktů.
- Určité omezení a neschopnost kontrolovat malé chyby, např. chyby uživatelského rozhraní, gramatické chyby atd.



Obrázek 2.2: Porovnání nákladů při automatickém a ručním testování.
Převzato a upraveno z [42].

Techniky testování z pohledu přístupu k softwaru

Z pohledu jednotlivých technik, které jsou používané při testování, se základní metody testování dělí na černou skříňku a bílou skříňku [35]. Rovněž existuje i metoda šedé skříňky, která kombinuje přístupy bílé a černé skříňky [13]. Metody se od sebe liší mírou přístupu ke zdrojovému kódu testovaného programu.

- Černá skříňka (angl. Black box) je speciální metoda testování výkonu softwaru, při které funkčnost produktu je zkoumána bez znalosti a analýzy zdrojového kódu. Testovací případy jsou vytvořeny na základě požadavků a specifikací softwaru. Používá se při testování provozu celého systému a taky při funkčním testování.

  Mezi výhody této metody patří:

  - Testování se provádí z pohledu koncového uživatele a je schopno odhalit nepřesnosti a nesrovnalosti ve specifikaci.
  - Znalosti programování nejsou potřebné, stejně jako způsob implementace softwaru.
  - Testovací případy lze vyvinout ihned po dokončení práce se specifikací.

  Mezi nevýhody patří:

  - Bez jasné specifikace je poměrně obtížné vytvořit efektivní testovací případy.
  - Testuje se pouze velmi omezený počet způsobů spuštění programu.

- Bílá skříňka (angl. White box), na rozdíl od metody černé skříňky, potřebuje znalosti vnitřní struktury a technické vlastnosti softwaru. Testování bílé skříňky zahrnuje pokrytí kódu, rozvětvení jeho cest, příkazů a interní logiky kódu. Využití této metody je nejvhodnější pro testování na nižší úrovni, jako je testování jednotek nebo integrační testování.

  Metoda bílé skříňky má svoje přínosy:

  - Testování lze provést v rané fázi vývoje softwaru (např. testování jednotlivých funkcí či jednotek).
  - Na rozdíl od černé skříňky, testování může pokrývat velké množství způsobů spuštění programu.

  Mezi nedostatky patří:

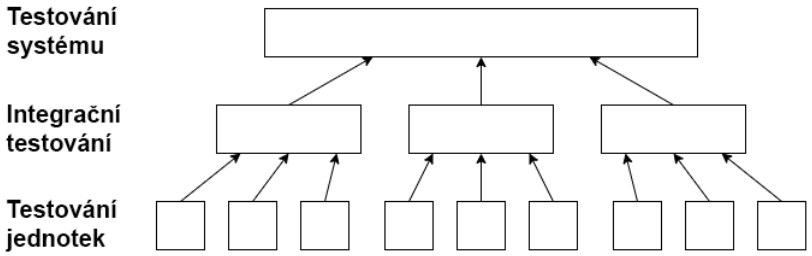
  - Vyžaduje specializované znalosti softwaru a programovacích jazyků.
  - Při použití automatických testů a při časté změně samotného software, je potřeba udržovat a aktualizovat testovací skripty.

- Šedá skříňka (angl. Grey box) je metoda, která se skládá z kombinací přístupů bílé skříňky a černé skříňky. Znalosti vnitřní struktury softwaru nejsou úplné, což dovoluje vytvářet testy na základě jednoduchých znalostí algoritmu a dalších charakteristik chování softwaru, ale samotné testování se provádí technikou černé skříňky, tj. z pohledu uživatele.

Úrovně provádění testů

Testování na různých úrovních abstrakce programu se provádí během celého životního cyklu vývoje a údržby softwaru. Úroveň testů určuje, na kterých dílčích součástkách systému probíhá testování. Můžou to být jednotlivé prvky, konkrétní oblasti systému (skupina spolupracujících prvků) a produkt jako celek.

- Testování jednotek (angl. Unit testing): testování nejmenších částí (např. tříd nebo metod) softwaru izolovaně od zbytku programu. Hlavní myšlenka jednotkového testování spočívá v napsání testu pro každou netriviální funkcí nebo metodu. Takový přístup umožňuje rychle zkontrolovat, zda další změna v kódu nevedla k regresi, tj. k výskytu chyb v již testovaných částech programu. Zároveň jednotkové testování usnadňuje detekci a eliminaci zjištěných chyb.
- Integrační testování (angl. Integration Testing): testování interakce několika jednotek, které pracují společně. Ověřuje bezchybnost komunikace mezi jednotlivými komponentami programu. Pro lepší pokrytí je potřeba vytvořit velké množství testů, jelikož počet testů je exponenciální závislý na počtu interagujících komponent (čím více komponent, tím více kombinací mezi nimi).
- Testování systému (angl. System testing): testování celého systému za účelem ověření, zda systém splňuje původní funkční i nefunkční požadavky. Kromě výše zmíněného testování v této fázi probíhá akceptační testování, zátěžové testování, testování použitelnosti, testování kompatibility, testování uživatelského rozhraní, testování výkonu atd.



Obrázek 2.3: Úrovně provádění testů.

Techniky testování

V současné době lze nástroje klasifikovat podle principů a technik, které využívají pro generování testovacích případů:

- Testování softwaru založené na zdrojovém kódu (angl. Code-Based Testing) je testování, které kontroluje, zda během testu byl spuštěn každý příkazový řádek kódu. Úkolem je zjistit, existují-li v programu nějaké chyby, a to pomocí rozumně vybraných sad testovacích případů, které generují potřebné výsledky. Tyto výsledky by měly zajišťovat splnění kritéria pokrytí. Z ohledem na míru přístupu, tento druh testování používá techniky bílé skříňky. Pro generování testů je nutné pochopení programovacích jazyků a programové části softwaru. Testy se provádí na úrovni jednotek, tj. jednotlivé komponenty systému jsou testované izolované vůči jiným komponentám. Kromě toho na úrovni zdrojového kódu je možné provést integrační testování a ověřit spolupráci několika jednotek.

  Výhody testování založeného na zdrojovém kódu:

  – Pokrývá všechny možné cesty kódu, čím umožňuje provádět důkladné testování.
  – Testovací případy lze snadno automatizovat.

– Umožňuje optimalizovat kód při odhalení skrytých chyb, které se během obyčejného testování nemusí objevit.

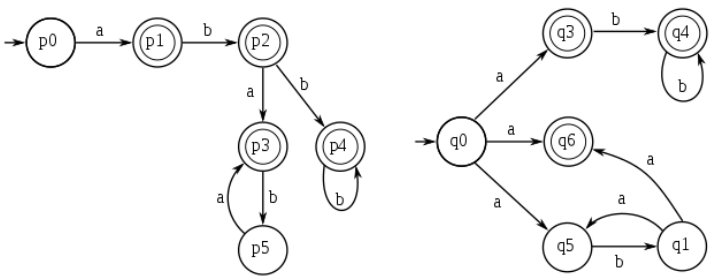
Nevýhody:

– Složitý a nákladný postup testování, který vyžaduje znalosti programování a pochopení testovaného systému.
– Pokud se implementace často mění, je potřeba aktualizovat testovací skripty během každé změny.
– Nutnost vytvořit řadu vstupů, které by pokrývaly všechny cesty a podmínky testovaného systému.
– Pokud je nástroj zaměřen na vysoké pokrytí, může se stát, že některé chyby nebudou nalezeny. [16].

- Testování založené na modelu (angl. Model-Based Testing) [53] je testování softwaru, při kterém jsou testovací případy částečně nebo zcela získané z modelu, který popisuje některé aspekty (ve většině případů se jedná o funkční aspekty) testovaného systému. Testovací modely jsou popsané pomocí grafů, tabulek, diagramů atd. a měly by být vytvořeny ve fázi návrhu systému (nebo jeho jednotlivých částech) pro demonstraci práce systému a spolupráce jeho součástek. Tímto model poskytuje jasnější pohled na systém všem účastníkům vývoje softwaru a usnadňuje údržbu po nasazení systému do provozu. Testování na základě modelů využívá techniky černé skříňky: systém má určitý "vstup" pro zadávání informací a "výstup" pro zobrazení výsledků práce, zatímco procesy probíhající během provozu systému nejsou známy. Předpokládá se, že stav výstupů je funkčně závislý na stavu vstupů.

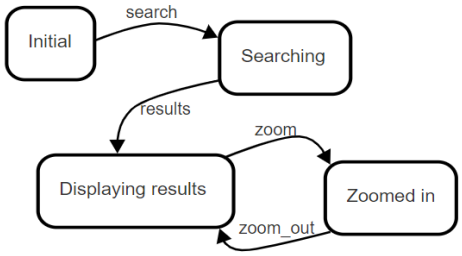
  Pro popis chování softwaru nebo jeho součástí je potřeba vytvořit model, který pak bude použit pro generování testovacích případů. Chování lze popsat z hlediska vstupních a výstupních dat, akcí a podmínek pro jejich provádění, a popisu toku dat. Při testování lze využít modely, které jsou popsány ve formě:

  – Model konečného automatu (angl. Finite State Machine): jedná se o abstraktní model, který zahrnuje konečný počet stavů a přechodů mezi nimi. Dělí se na:
    – Deterministický: automat se nachází právě v jednom ze svých vnitřních stavů, tzn. z libovolného stavu je možný přechod pouze do jednoho cílového stavu.
    – Nedeterministický: v jednom okamžiku se může nacházet v celé množině svých vnitřních stavů, tzn. z jednoho stavu je možný přechod do množiny stavů. Navíc může obsahovat prázdné přechody.



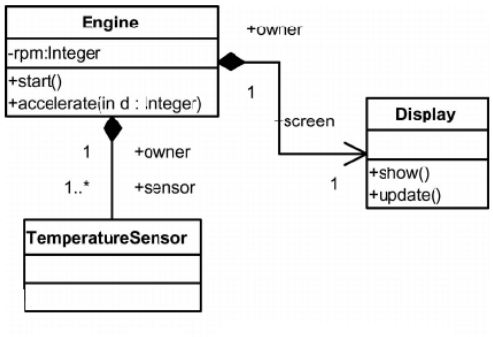
Obrázek 2.4: Deterministický (vlevo) a nedeterministický (vpravo) automat.

– Statecharts [20]: představuje vizuální formalismus pro popis stavů a přechodů, který umožňuje shlukování, ortogonalitu, podporuje možnosti přechodu mezi úrovněmi abstrakce a přidává všesměrovou (angl. broadcast) komunikaci pomocí globálních proměnných. [23]



Obrázek 2.5: Statecharts.

– UML (angl. Unified Modeling Language): standardizovaný grafický jazyk pro specifikaci, vizualizací, konstrukci a dokumentaci při objektově orientované analýze a návrhu. Poskytuje integrovaný modelovací rámec, který zahrnuje softwarovou strukturu a architekturu, stejně jako popisy chování programu [18].



Obrázek 2.6: UML.

Výhody testování založené na modelech:

– Umožňuje se podívat na systém (nebo část systému) jako na jeden celek a zjistit zjevné závislosti.
– Nevyžaduje žádné programovací znalosti.
– Pomocí modelu lze snadno vygenerovat velké množství testovacích případů. Testy mohou být založeny na základě požadavku a specifikace softwaru (např. akceptační testy).
– Testovací případy jsou generovány z modelů za využitím testovacích kritérií. Kvalitní nástroj by měl pracovat s více než jedním kritériem pokrytí, čímž způsobí různé kombinace vstupních dat pro testovaný systém a tedy zvětšují šanci najít chyby.

  - Je to ideální varianta pro dlouhodobé projekty, kde velký počet testovacích případů zhorší pochopení principu činností systému. Jednoduchý a intuitivní diagram, ze kterého se skládá model, ho naopak zjednoduší.

  Nevýhody:

  - Je kladen velký důraz na údržbu a aktualizaci modelu.
  - Vytvoření modelu zabírá více času, než jiné alternativy (např. vytvoření checklistu). .
  - Používání testovacích modelů vyžaduje určité dovednosti abstraktního myšlení spojeného s důrazem na detaily.
  - Zvyšuje celkové náklady na testování projektu.

- Testování založené na chybách (angl. Fault-based testing) [34] je testování, cílem kterého je zjistit nepřítomnost předem specifikovaných chyb v softwaru. Tato technika testování softwaru je založená na změnách zdrojového kódu a kontrole odezvy na tyto změny v sadě automatických testů. Testovací případy jsou aplikovány na původním programu a také na mutantní verzi programu. Po porovnání výsledků, pokud byl vygenerován stejný výstup, pak jsou dvě možnosti: změněný kód nebyl pokryt žádným z testovacích případů, nebo testovací případy nejsou efektivní. Mutantní verzi programu lze vytvořit pomocí změny logických, aritmetických a relačních operátorů nebo změny příkazů.

  Hlavní přínosy testování založeného na chybách:

  - Umožňuje pokrytí celého zdrojového kódu a detekci částí, které nejsou správně testovány.
  - Odkrývá nejednoznačnosti ve zdrojovém kódu.
  - Ve výsledku koncový uživatel získává bezpečný a spolehlivý systém.

  Hlavní nedostatky jsou:

  - Tento proces je extrémně nákladný a časově náročný.
  - Složité mutace je obtížné implementovat.
  - Je potřeba automatizovat proces testování.

2.2.2 Testovací případ

Jednotlivé kroky, konkrétní podmínky a parametry potřebné k testování systému, jeho funkce nebo jeho části, dohromady tvoří tzv. testovací případ (angl. Test case). Testovací případ je přesně popsaný algoritmus pro testování, vytvořený ke kontrole výskytu určitých situací v programu a očekávaných výstupních dat. V praxi jsou případy odvozeny ze softwarových artefaktů (např. specifikace nebo navrhu programu). Množina testovacích případů vytváří testovací sadu (angl. Test suite).

Testovací případy se rozdělují podle očekávaného výsledku na pozitivní a negativní. Pozitivní testovací případ používá pouze platná data a ověřuje, zda aplikace správně provedla požadovanou funkci. Tyto případy jsou zaměřeny na kontrolu provozu softwaru a pracují s datovými typy, které jsou validní pro daný program. Negativní testovací případ využívá nesprávná data za účelem kontroly odolnosti systému vůči různým vlivům, ověření nesprávných údajů a řešení výjimečných situací.

Pro generování testovacích případů existuje několik technik [1]:

- Symbolická exekuce (angl. Symbolic execution): program dostává symbolický vstup, který neobsahuje konkrétní data, ale sadu všech možných dat, kterou je schopen přijmout program. Jedná se o techniku hledání chyb, která vytvoří konkrétní vstup (testovací případ), u kterého program nesplní specifikací. Symbolická exekuce nemůže prokázat absenci chyb v programu. [26]
- Náhodné generování (angl. Random generation): program získává na vstup náhodná nezávislá data. Výsledky testování jsou pak porovnány se specifikacemi softwaru. [5] Hlavní výhoda je možnost rychle vygenerovat velký počet testů. Zároveň proces hledání chyb je stejně náhodný jako samotný vstup programu.
- Testování softwaru založené na vyhledávání (angl. Search-based software testing) používá meta-heuristické vyhledávací techniky (např. Genetický algoritmus, Simulované žíhání nebo Gradientní algoritmus) k částečné nebo úplné automatizace procesů testování. [29]

Postupy generaci testových případů se dělí na:

- Offline: testovací sada je předem vypočítána ze specifikace, a poté se automaticky provádí na testované implementaci (angl. Implementation Under Test). Hlavní výhodou offline generování testů je to, že při změně požadavků nebo modelu testovacího případu, jsou testy automaticky znovu vygenerovány bez ruční aktualizace každého testovacího případu či skriptu.
- Online (on-the-fly): kombinuje generování testů a jejich spuštění. Testovací generátor vygeneruje z modelů pouze jeden vstup (typicky je vybrán náhodně), který se poté okamžitě provede na testované implementaci. Následně produkovaný výstup z implementací (pokud existuje) a čas jeho výskytu jsou zkontrolovány dle specifikace, a poté se vygeneruje nový vstup atd. Tento proces se opakuje do té doby, pokud není detekována chyba nebo pokud proces testování není ukončen zvenku. Z toho vyplývá hlavní výhoda online generování testů: testování může pokračovat po dlouhou dobu (hodiny nebo dny) a tím umožní provádění zátěžového testování.

2.2.3 Kritéria pokrytí

Součástí testování je stanovení efektivity testů. Toho se dá dosáhnout pomocí sledování pokrytí (angl. Coverage). Pokrytí je míra, která udává, jak moc daná testovací sada zkoumá testovaný systém. Hlavním cílem testovacích sad na základě předem zvoleného kritéria pokrytí je dosáhnout 100% pokrytí.

- Kritéria pokrytí zdrojového kódu (angl. Code Coverage)
  - Pokrytí příkazu (angl. Statement coverage): kontrola, zda byl spuštěn každý řádek programu alespoň jednou během provádění testu. Používá se pro výpočet počtu příkazů ve zdrojovém kódu, které byly provedeny. Hlavním účelem je pokrytí všech možných cest, řádků a příkazů ve zdrojovém kódu.

    Například, máme definovanou funkci `func` (viz. obrázek 2.7), která dostává na vstup proměnnou `a`, poté funkce porovnává ji s číslem `5`. V závislosti na získaném výsledku podmíněného příkazu vypíše různý text. V případě, že proměnná `a = 6`,

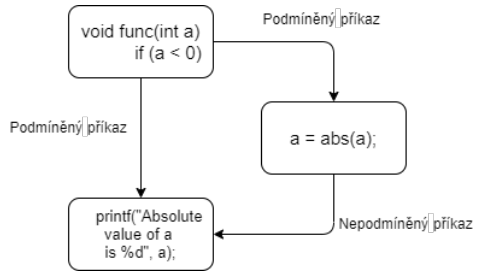
budou provedeny 4 příkazy (1, 2, 3 a 6 řádek). Celkový počet řádků je 6, tz. pokrytí příkazu bude $4/6 = 67\%$.

```
1 void func(int a){
2     if (a > 5)
3         printf("A is greater than 5.");
4     else
5         printf("A is smaller than 5.");
6 }
```

Obrázek 2.7: Příklad č.1.

– Pokrytí větvení (angl. Branch coverage): během testu by program měl projít všemi větveními podmíněného příkazu (např. `if-else`, `switch-case`) a zkontrolovat jejich úplnost. Příklad testovacího kódu je znázorněn na obrázku 2.8. Máme celkem tří větví: dvě z podmíněného příkazu `if` a jedna větev nepodmíněného příkazu. Pokud je proměnná `a` definovaná hodnotou `-7`, pak budou pokryty dvě ze tří větví kódu. V tomto případě pokrytí větvení je $2/3 = 67\%$.

```
1 void func(int a){
2     if (a < 0)
3         a = abs(a);
4     printf("Absolute value of a is %d", a);
5 }
```



Obrázek 2.8: Příklad č.2.

– Pokrytí podmínek (angl. Condition coverage): používá se k testování a vyhodnocení proměnných nebo dílčích výrazů v podmíněném příkazu, které mohou nabývat hodnot `true` nebo `false`. Cílem pokrytí je zkontrolovat jednotlivé výsledky pro každou logickou podmínku. Například, máme podmínku `(a > 0) && (a < b)` (viz. obrázek 2.9). Celkem jsou 4 možnosti:

  `(a > 0) = True` a `(a < b) = True`,

  `(a > 0) = True` a `(a < b) = False`,

  `(a > 0) = False` a `(a < b) = True`,

  `(a > 0) = False a (a < b) = False`.

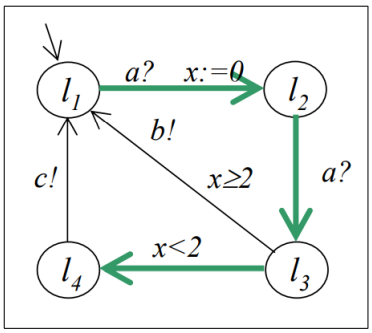
  Pokud se proměnná `a` rovná `5` a proměnná `b` se rovná `3`, pak nastává první možnost, a to `True && False`. Pokrytí podmínek je v tomto případě $1/4 = 25\%$.

```
1 void func(int a, int b){
2     if ((a > 0) && (a < b))
3         printf("A is positive number, but smaller than B");
4
5 }
```

Obrázek 2.9: Příklad č.3.

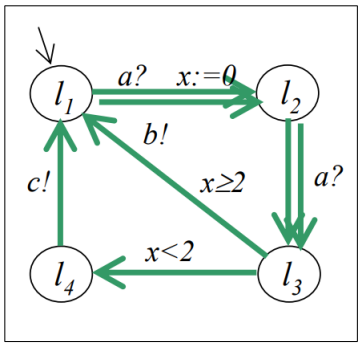
  - Pokrytí cest (angl. Path coverage): testuje splnění kritéria pokrytí každé logické cesty programu. Pokrytí cest sleduje všechny možné průchody kódem.

- Kritéria pokrytí modelu (angl. Model Coverage)
  - Pokrytí stavu (angl. Location coverage): testovací případ, během spuštění na modelu, by měl navštívit každé místo z vybraných komponent. Máme definovaný časovaný automat (viz. obrázek 2.10) který se skládá ze čtyř míst l_1, l_2, l_3, l_4 a z pěti hran mezi nimi. Chceme-li dosáhnout 100% pokrytí míst, potřebujeme, aby testovací případ navštívil každé z těch míst. Jedna z možností cest je $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4$.



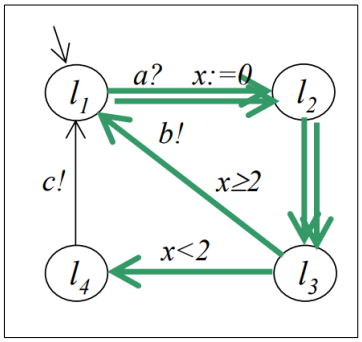
Obrázek 2.10: Příklad č.4.

  - Pokrytí hran (angl. Edge coverage): testovací sekvence prochází každou hranou vybraných komponent během spuštění na modelu. Pro 100% pokrytí všech hran je třeba projít každou hranou automatu. Jedna z možných cest je zobrazena na obrázku 2.11, a vypadá jako $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_1$.



Obrázek 2.11: Příklad č.5.

  - Dvojice definice-užití (angl. Definition/use pair coverage): jedná se o techniku pokrytí toku dat. Hlavním cílem je pokrytí cesty od definování proměnné a do pozdějšího použití této proměnné. V uvedeném příkladě je pokryta cesta hodinové proměnné x(viz. obrázek 2.12). Cesta zahrnuje všechny hrany, které využívají tuto proměnnou. V tomto případě by měla cesta vypadat takhle $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4$.



Obrázek 2.12: Příklad č.6.

2.3 Přehled existujících nástrojů

V posledních letech se objevilo velké množství nástrojů, které umožňují automatické generování testů, ale pouze některé z nich po jejich realizaci byly dále rozvíjeny a jsou v dnešní době aktuální. Existuje několik výzkumů [17] [10] [45] [30] [31], které dávají přehled nad současným stavem nástrojů pro generování testovacích případů. K zpracování byly vybrány programy podle následujících kritérií: dostupnost popisu programu ve vědecké literatuře, program byl aktualizován během posledních pěti let, volný přístup k programu přes oficiální stránku nástrojů. Kromě toho byly přidány komerční programy pro zobrazení situací i v tomto směru.

CATG

CATG [44] je nástroj pro concolic (kontaminace angl. slov concrete (konkrétní) a symbolic (symbolický)) testování jednotek zapsaných v programovacím jazyce Java. Na vstup nástroj přijímá java byte kód. Concolic testování je hybridní technika ověřování softwaru, která provádí dynamicky symbolickou exekuci, zatímco software je spuštěn s konkrétními vstupními hodnotami. Concolic testování udržuje konkrétní stav a symbolický stav: konkrétní stav mapuje všechny proměnné na jejich konkrétní hodnoty a symbolický stav mapuje pouze proměnné, které nemají konkrétní hodnoty. [48] Nástroj je volně dostupný a poslední aktualizace proběhla v roce 2018.

Conformiq

Conformiq Holding [9] je firma, která nabízí několik komerčních nástrojů pro automatizaci procesů testování. Hlavní z nich jsou Conformiq Creator, který pracuje spolu s Conformiq Transformer, a Conformiq Designer.

- Conformiq Creator automaticky generuje testy a aplikuje je na software, webové aplikace a webové služby. Pro vstupní data využívá vlastní modelovací jazyk, který je založen na diagramu aktivit a jazyce pro popis akcí. Modely lze importovat z již existujících aktiv, pokud jsou popsány, např. pomocí vývojového diagramu, BPM nebo jsou to ručně psané testy a testy napsané v jazyce Gherkin [46]. Požadavky lze stáhnout z RMT a výsledky ve formě vygenerovaných testů lze exportovat do nástrojů, jakož jsou ALM, Excel, různé skriptovací jazyky, nebo lze spustit provádění testů pomocí nástrojů Conformiq Transformer.

- Conformiq Designer (dříve Conformiq Qtronic) [24] se používá při vývoji vestavěného softwaru k automatizaci procesů testování a zrychlování vývoje v různých průmyslových oblastech, včetně oblastí komunikace, automobilového průmyslu a poskytovatelům síťových zařízení. Modely jsou implementované pomocí stavového modelu UML a modelového jazyku Qtronic. Vygenerované testy lze exportovat do nástrojů pro správu testů nebo TTCN-3. V poslední aktualizaci, která proběhla v roce 2019 (v5.0) byla přidána funkce testování založená na chybách, což umožňuje zvýšit kvalitu generovaných testovacích případů.

EvoSuite

EvoSuite [14] je volně dostupný nástroj pro automatické generování Junit testovacích případů pro vybrané třídy v jazyce Java. EvoSuite pracuje na úrovni bitového kódu, to znamená, že nástroj nevyžaduje zdrojový kód testovaného softwaru a je v zásadě použitelný i v jiných programovacích jazycích, které se kompilují do bitového kódu Java (napr. Scala nebo Groovy). Využívá techniku testování založenou na vyhledávání - EvoSuite používá hybridní přístup, který generuje a optimalizuje testovací případy, aby splňoval kritérium pokrytí. Umožňuje optimalizace různých kritérií pokrytí, např. pokrytí příkazů a větvení [15]. Nástroj minimalizuje počet testů tím, že zachovává pouze ty, které přispívají k dosažení pokrytí. Poslední verze programu je v1.1.0 z roku 2020.

GraphWalker

GraphWalker [49] je volně dostupný nástroj. Generuje testy pro systémy, které jsou reprezentovány ve tvaru směrovaných grafů. Pro prohledávání využívá A* algoritmu nebo náhodné algoritmy s omezením pro různá kritéria pokrytí (pokrytí stavu, hran a požadavků). [54] Nástroj se neustále vyvíjí, poslední aktualizace proběhla v roce 2021 (v.4.3.1).

Jtest

Parasoft [37] je firma, která nabízí několik komerčních nástrojů pro automatické testování programů napsaných v programovacím jazyce C, C++, Java atd. Poslední aktualizace těchto programů proběhla v roce 2021. Jtest [38] je jeden z nejvýznamnějších nástrojů firmy Parasfot. Nabízí automatické generování testovacích případů pro software v jazyce Java, jejich provádění, statickou analýzu, regresní testování, detekci chyb za běhu programu a anotace kódu.

KLEE

KLEE [50] je open-source nástroj který je založen na symbolické exekuci pro automatické generování testovacích případů. KLEE je navržen jako upravený LLVM (Low Level Virtual Machine). Kromě symbolické deklarace proměnných v programu, nástroj poskytuje symbolickou knihovnu POSIX, která umožňuje analyzovat software [7]. Z poslední verzí nástrojů (v2.2 od roku 2020) byla přidána podpora výjimek jazyka C++ a podpora poslední verzi LLVM.

Modbat

Modbat [8] je modelový testovací nástroj, který je založen na konečných automatech. Nástroj se specializuje na testování aplikačního programového rozhraní (api) softwaru. Model, se kterým pracuje nástroj, má strukturu nedeterministického konečného automatu. K testování je vybrán jeden ze všech možných přechodů systému. Anotace přechodu určují, který úsek kódu má být proveden. Poté co je test ukončen, model se resetují do původního stavu. Podporuje online a offline běh testování. [2] Poslední verze nástroje byla vydána v roce 2020 (v 3.5). Byly přidány určité knihovny a byla vylepšena kompatibilita prostředí Java 11.

MoMut

MoMut [51] je sada nástrojů pro generování testovacích případů pro systémy založené na modelu. Umožňuje generování a spuštění testu v offline režimu. Vstupní data může být ve formě grafu UML, akčních systému, časovaných automatu. Na rozdíl od ostatních nástrojů pro generování testovacích případů MoMuT je zaměřen na testování založené na chybách. Kromě standardních technik, nástroj obsahuje strategii generování testovacích případů založené na chybách (pomocí operátoru mutace). [27] Poslední aktualizace nástroje proběhla v roce 2020 (v 4.1.1).

PragmaDev

PragmaDev Studio [39] je sada nástrojů pro modelování a testování, která umožňuje spravovat složitost vývoje různých systémů. Skládá se z několika modulů:

- PragmaDev Specifier umožňuje jednoznačně specifikovat a ověřovat funkce systému a definovat architekturu systému pro dosažení nejlepšího výkonu nebo energetické účinnosti. Výsledkem je grafický a spustitelný model systému.
- PragmaDev Developer dovoluje psát udržovatelný a dokumentovaný kód.
- PragmaDev Tester poskytuje prostředky pro psaní integračních testů. Podporuje mezinárodní testovací jazyk TTCN-3 včetně syntaktického a sémantického ověřování, simulace, generování kódu, ladění a grafických stop.
- PragmaDev Tracer používá se v počáteční fázi k popisu očekávaného chování systému, jeho vlastnosti a různých scénářů. V pozdější fázi se používá ke sledování běhu programů a jeho ověřování, zda výsledný systém je v souladu s očekávanými vlastnostmi popsanými v první fázi.

S poslední verzí, která byla vydána v roce 2021 (v5.6), byly změněné některé části GUI a přidána možnost ukládání implementovaných modelů do formátu PDF.

Randoop

Randoop [43] je volně dostupný nástroj pro testování jednotek v jazyce Java, který používá náhodné generování testů se zpětnou vazbou. Tato technika pseudonáhodně generuje sekvenci metod/konstruktérů pro testování tříd v programu. Randoop slouží k vyhledávání chyb a vytváření regresních testů. Nástroj provádí vytvořenou sekvenci za použití výsledků provádění k vytvoření tvrzení, které zachycuje chování programu nebo umožňuje zachycení chyb [36]. Nástroj se neustále vyvíjí a poslední změny proběhly v roce 2021 (v 4.2.6).

T3

T3 [41] je automatizovaný nástroj pro testování tříd v jazyce Java, které probíhá na úrovni jednotek. Vzhledem k cílové třídě, která se má otestovat, nástroj náhodně generuje sekvenci volání metod třídy. Vygenerované sekvence lze uložit jako soubor testovací sady pro opakované použití. Podobně Randoop využívá generování testů se zpětnou vazbou, což umožní v případě neúspěchu rozšíření testovací sekvenci se vrátit na předchozí krok a zkusit přidat jiný krok [40]. Poslední verze nástroje byla vydána v roce 2019.

UPPAAL

UPPAAL [52] je softwarový nástroj pro modelování, validaci a verifikaci systémů reálného času. Systémy jsou reprezentované ve formě sítě časovaných automatů, které jsou rozšířené o datové typy. Pomocí simulátoru Uppaal umožňuje generovat a vizualizovat různé cesty běhu systému. Verifikátor dovoluje ověřit systém pomocí dotazů, které používají temporální logiku. Generátor testovacích případů přebírá systém vytvořeny v Uppaal a vytváří sadu testovacích případů, jejichž cílem je pokrýt všechny syntaktické přechody (hraný) modelu (angl. edge coverage). Nástroj má dlouhou historii a byl použit v řadě průmyslových případových studií. Poslední stabilní verzí byla představena v roce 2019. [21].

Kapitola 3

Realizační prostředky

Pro následující práci byl zvolen nástroj UPPAAL v4.1.24, který nabízí uživatelské rozhraní pro modelování, simulaci a verifikaci modelu, umožňuje generovat spustitelné testovací případy na základě testovacího kódu zadaného do modelu. Díky tomu, že se testovací kód zadává v textové podobě, pro výstupní testovací kód lze použít jakýkoliv programovací jazyk. Cílem této kapitoly je seznámit čtenáře s nástrojem, jeho součástmi, vysvětlit proseč modelování systémů, ověření jeho běhu a poté způsob vytvoření testovacích případů. Při psaní kapitoly byla použita oficiální dokumentace nástroje [4] a vědecké články [28] [22].

3.1 UPPAAL

UPPAAL je softwarový nástroj pro modelování, validaci a verifikaci systému v reálném času. Systémy jsou reprezentované ve formě sítě časovaných automatů, rozšířené o datové typy. Nástroj UPPAAL byl vytvořen ve spolupráci dvou univerzit: Katedry Informačních Technologií (Department of Information Technologies) na Univerzitě Uppsala ve Švédsku a Katedry Informatiky (Department of Computer Science) na Univerzitě Aalborg v Dánsku. První verze byla vydána v roce 1995 a od té doby se neustále vyvíjí. Poslední aktualizace programu proběhla v roce 2019, kdy byla vydána stabilní verze 4.0.15 a verze 4.1.24, která se nachází ve fázi vývoje. UPPAAL obsahuje uživatelské rozhraní, které je napsané v jazyce Java, a nástroje pro verifikaci v jazyce C++. Tento softwarový nástroj je volně dostupný pro akademické účely a lze ho získat z oficiální stránky [52], navíc UPPAAL nabízí komerční verzi svého nastroje [25].

Rozšíření nastroje UPPAAL

Kromě základní verze programu, UPPAAL nabízí různá rozšíření.

- Cora [3] poskytuje efektivní nástroj pro analýzu nákladově optimální dosažitelnosti (angl. Cost Optimal Reachability Analysis). Používá rozšíření časovaných automatů pod názvem LPTA. LPTA umožňuje anotovat model za využitím pojmu “cena”. Může to být cena zpoždění v určitých situacích nebo cena konkrétních akcí. Poté nástroj vyhledá optimální cestu odpovídající cílovým podmínkám.
- SMC [11] nabízí vysoce rozšiřitelný nástroj pro kontrolu statického modelu, který podporuje analýzu výkonu stochastických hybridních automatu. Odkazuje na řadu

technik, které monitorují několik běhů systému s ohledem na žádanou vlastnost a poté používá výsledky k získání odhadu správností návrhu.

- Stratego [32] podporuje syntézu (za využitím strojového učení) a hodnocení blízko optimálních (near-optimal) strategií pro stochastické časované hry. Umožňuje efektivní a flexibilní průzkum "strategických prostorů" před adaptací do finální implementace.
- Tiga [12] umožňuje automatickou syntézu strategií s důrazem na zadané cíle bezpečnosti a živosti. Implementuje první efektivní "on-the-fly" algoritmus pro řešení her založených na časovaných herních automatech s ohledem na dosažitelnost a bezpečnosti vlastnosti.
- Tron [33] využívá se pro testování shody časovaných systém pomocí strategie černé skříňky, zaměřený hlavně na vestavěný software používaný v různých řádcích.

3.2 Časovaný automat v nástroje UPPAAL

Nástroj je založen na teorii časovaných automatů. Časovaný automat je konečný automat rozšířený o hodinové proměnné, které nabývají reálné hodnoty. V Uppaalu je systém modelován jako síť několika takových časovaných automatů, které běží paralelně. Při inicializaci systému se hodiny nastaví na nulu, a poté všechny hodiny běží spojitě a se stejným tempem pro celý systém. V případě potřeby lze jednotlivé hodiny vynulovat.

Formálně lze časované automaty zapsat jako šestici $A = (\Sigma, L, l_0, C, I, E)$

- Σ je konečná množina akcí,
- L je konečná množina míst,
- $l_0 \in L$ je množina počátečních míst,
- C je konečná množina hodin,
- I přiřazuje každému místu nějaké omezení hodin z $\Phi(C)$. $I : L \to \Phi(C)$,
- $E \subseteq L \times \Sigma \times 2^C \times \Phi(C) \times L$ je množina přechodů

Ohodnocení hodin je funkce $u : C \to R_{\geq 0}$ z množiny hodin do nezáporných reálných čísel. Nechť R^C je množina všech ohodnocení hodin. Nechť $u_0(x) = 0$ pro všechna $x \in C$. Zápis $u \in I(l)$ bude znamenat, že u splňuje $I(l)$.

Sémantika časovaného automatu A je definována přechodový systémem $\langle S, s_0, \to \rangle$, kde $S \subseteq L \times R^C$ je množina stavů, $s_0 = (l_0, u_0)$ je počáteční stav, $\to \subseteq S \times \{R_{\geq 0} \cup A\} \times S$ je průchodová relace taková, že

- $(l, u) \xrightarrow{\mathrm{d}} (l, u + d)$ pokud $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$, a
- $(l, u) \xrightarrow{\mathrm{a}} (l', u')$ pokud $\exists e = (l, a, g, r, l') \in E$ *s.t.* $u \in g$, $u' = [r \mapsto 0]u$, a $u' \in I(l)$,

kde pro $d \in R_0$, $u + d$ zobrazuje každé hodiny $x \in C$ na hodnotu $u(x) + d$,

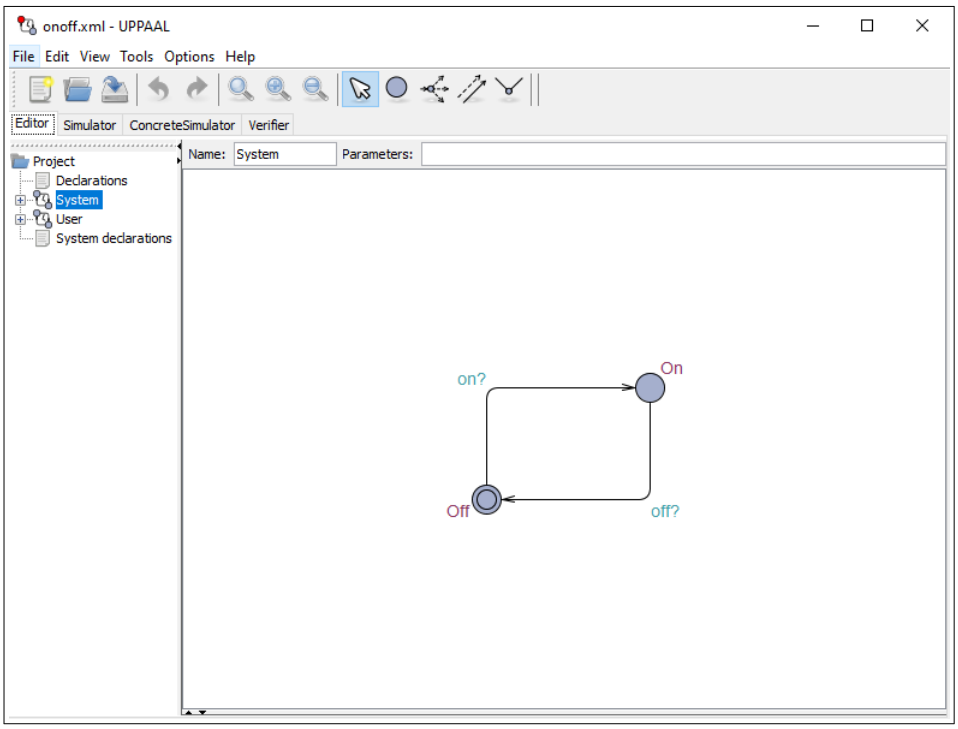
$[r \mapsto 0]u$ označuje ohodnocení hodin, která mapuje každé hodiny v r na 0 a souhlasí s u nad $C \backslash r$.

3.3 Součásti nástroje

Nástroj se skládá ze čtyř hlavních částí:

- Editor slouží k vytvoření modelu. Poskytuje prostředky pro definice automatu, deklarace proměnných a funkcí.
- Simulátor umožňuje zkoumání dynamických spuštění systému a to buď ručně (uživatel může vybrat, které přechody se mají provést), náhodně (systém běží sám) nebo uživatel může projít trasou, která byla uložena nebo importovaná z verifikátorů.
- Konkrétní Simulátor (angl. Concrete Simulator) umožňuje zkoumání systémů v raném stádiu vývoje nebo modelování. Hlavní rozdíl on Simulátorů spočívá v tom, že simulace se skládá z konkrétních stop (trace), pomocí kterých lze zvolit konkrétní čas pro aktivaci transakce/přechodu (transition). Rovněž z toho lze zjistit čas aktivace přechodu.
- Verifikátor (angl. Verifier) se používá k specifikaci a ověřování různých dotazů, a k získání výsledků z nich. Využívá se on-the-fly zkoumání systému.

Kromě verifikátorů existuje nástroj pro generování testovacích případů, které se zapíná přes tlačítko *Tools*. V raných verzích se tento nástroj jmenoval Yggdrasil. Nástroj se používá pro generování testovacích případů a případně zvýšení pokrytí hran vygenerované cesty. Yggdrasil generuje cesty (angl. traces) z modelu a jejich následující převedení do testovacích případů. Pracuje s deterministickými systémy, ve kterých nedoje k vzájemnému čekání (angl. deadlock free), které lze ověřit pomocí verifikátoru.



Obrázek 3.1: Editor.

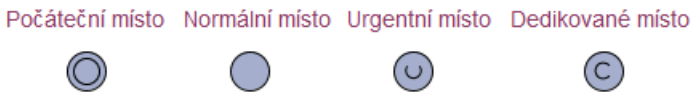
Editor

Editor umožňuje definovat globální deklarace (angl. Declaration) a vytvořit šablonu (angl. Template), která slouží pro definování automatu. Pro spojení jednotlivých šablon je potřeba deklarovat systém (angl. Systém declaration). Nástroj UPPAAL používá pro deklaraci syntaxi, která je podobná syntaxi programovacího jazyku C. Umožňuje používat datový typ `integer`, `boolean`, `double`, konstantní proměnné, pole, vícerozměrné pole s nastavenou velikostí, kanály (`chan`), hodiny (`clock`), vlastní typy dat a metody.

Pro popis chování součástek systémů se používají šablony, které se skládají z automatu a lokální deklarace. Časovaný automat je prezentován jako graf, který má místa (angl. location) a hrany (angl. edge) mezi nimi.

Místa lze pojmenovat podle potřeby, je to užitečné při verifikaci modelu nebo jeho dokumentaci. Kromě názvu u míst lze definovat podmínky (angl. invariant), které je potřeba splnit pro opuštění místa. Existuje několik typů míst:

- Počáteční místo (angl. initial location) se používá pro označení počátečního stavu modelu. Každá šablona musí mít přesně jedno umístění označené jako počáteční. Je označené pomocí kroužků uvnitř místa.
- Urgentní místa (angl. urgent location) jsou místa, ve kterých je čas pozastaven. Místo musí být opuštěné dříve než bude pokračovat běh času. Během pozastavení, lze provést jiné přechody, které nevyžadují čas pro svůj běh. Jsou označené pomocí `u` uvnitř místa.
- Dedikovaná místa (angl. committed location), stejně jako urgent místa, zastavují čas. Pro další přechod jsou však povoleny pouze přechody z jednoho `committed` místa. Jsou označené pomocí `c` uvnitř místa.
- Normální jsou ostatní místa.

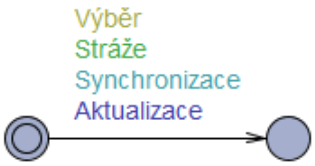


Obrázek 3.2: Různé typy míst.

Pro propojení jednotlivých míst a následujícího provedení přechodů se používají hrany. Hrany mají čtyři vlastnosti:

- Výběr (angl. selection), zde je možné vytvořit lokální (pouze pro tuto hranu) proměnnou. Podporované typy proměnné jsou ohraničená celá čísla a skalární sady.
- Stráže (angl. guard), výraz, který musí být splněn, aby bylo možné provést přechod. Stráž musí obsahovat jenom jednoduché podmínky na hodiny, rozdíly mezi hodinami a booleovské výrazy na proměnné.
- Synchronizace (angl. synchronization) slouží k synchronizaci přechodu za využitím kanálů.

- Aktualizace (angl. update) se používá pro přiřazení hodnot proměnné, vynulování hodin nebo pro volací funkci. Při využití několika výrazů je potřeba je oddělit pomocí čárky.

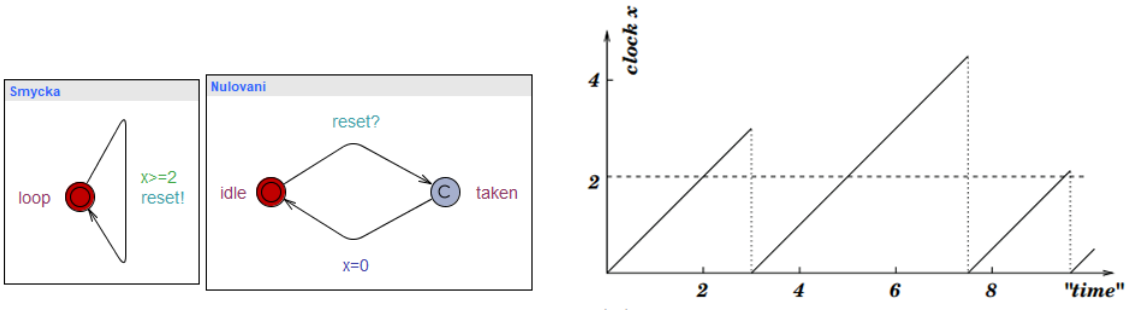


Obrázek 3.3: Vlastnosti hrany.

Jednou z hlavních součástí systému je kanál, který slouží pro komunikaci mezi modely. Pro odesílání signálu se používá konstrukce s vykřičníkem `signal!`, zatímco přijímací signál je označen otazníkem `signal?`. Když se dva procesy synchronizují, obě hrany se vypálí současně, tj. aktuální umístění obou procesů se změní. Existují tři typy kanálů:

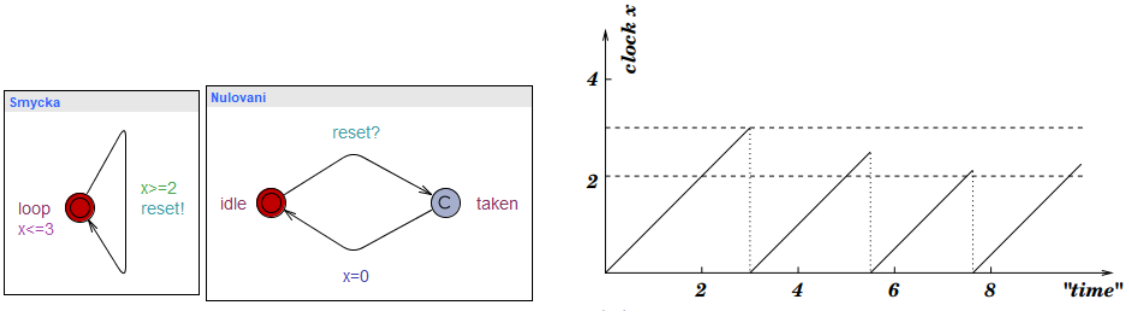
- Regulární (`chan c`): při vysílání signálů se může provést pouze jeden přechod se stejným přijímacím signálem. V případě několika přechodů se náhodně provede jeden z nich. Pokud však neexistuje žádný přijímací signál, přechod nemůže být proveden.
- Urgentní (`urgent chan c`): při vysílání nedojde ke zpoždění. Při popisu chování hrany je zakázané využívat hodiny ve stráži, ale využití výrazu, který obsahuje proměnné, je povoleno.
- Broadcast (`broadcast chan c`): vysílací kanál má jednoho odesílatele a více přijímačů. Hrana s vysílacím kanálem vždy vystřelí bez ohledu na to, zda jsou v systému nějaké hrany, které přijímají signál. Stejně jako u urgentních kanálů, i zde je zakázané mít výraz ve stráži, který obsahuje hodiny. V případě více přijímačů, přechody se provádějí zleva doprava podle pořadí, v jakém jsou procesy uvedeny v definici systému.

UPPAAL používá model nepřetržitého času, proto je velmi důležité pochopit princip použití času v podmínkách stavu a hran. Na obrázku 3.4 je znázorněn systém, který se skládá z modelu `Smyčka` a modelu `Pozorovatele` (v systému je pojmenován jako `Nulování`). Zároveň je v systému definována hodinová proměnná `x` a synchronizační kanál `reset`. Hrana `Smyčky` může vystřelit pouze v tom případě, pokud uběhne 2 nebo více časových jednotek a spolu s tím pošle signál `Pozorovateli`, který poté vynuluje hodinovou proměnnou.

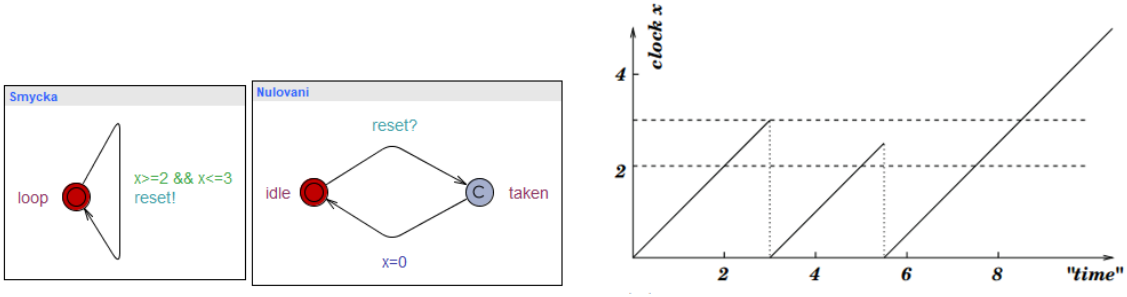


Obrázek 3.4: Příklad použití podmínky na hraně (vlevo) a graf chování času (vpravo).

V případě potřeby, lze zajistit horní hranice hodinové proměnné, a to pomocí přidání podmínky do stavu (viz. obrázek 3.5). Tímto vznikne omezení, které říká kolik časových jednotek se může v systému nacházet v tomto stavu. Před uplynutím tohoto času systém musí opustit tuto hranu. Stejné omezení je možné přidat do podmínek hrany, např. `x>=2 && x<=3` (viz. obrázek 3.6), ale v této situaci může nastat uváznutí pokud se systém někde zdrží a to tak, že podmínky pro výstřel hrany už nebudou splněny, např. `x` bude na 4 časové jednotce.



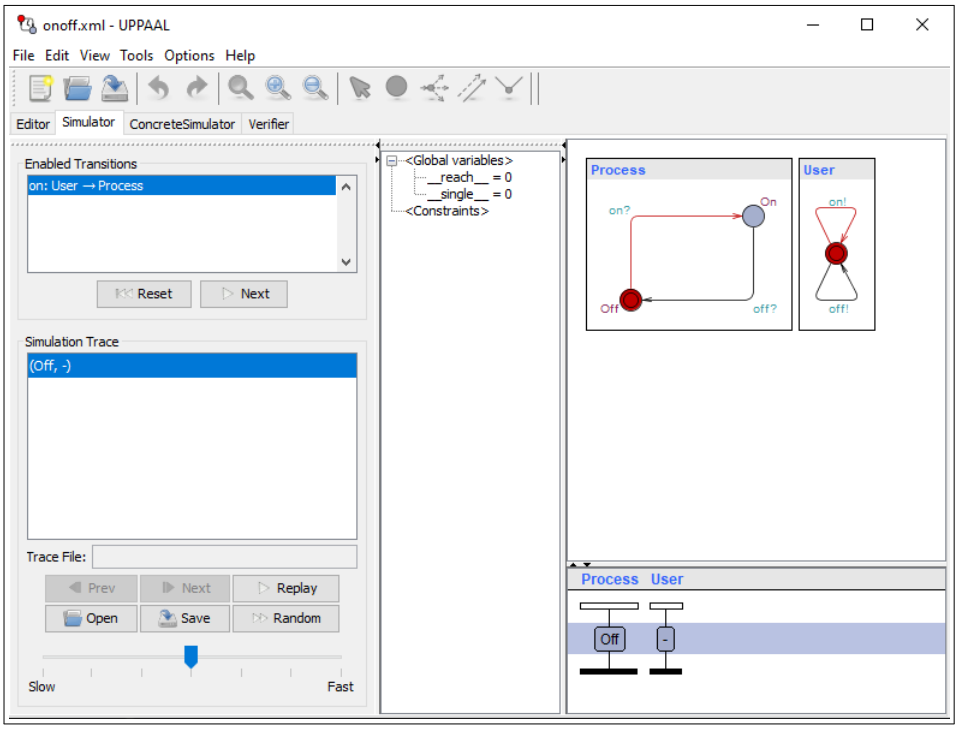
Obrázek 3.5: Příklad použití podmínky na hraně a na stavu (vlevo) a graf chování času (vpravo).



Obrázek 3.6: Příklad použití dvojité podmínky na hrane(vlevo) a graf chování času (vpravo).

Simulátor a konkrétní simulátor

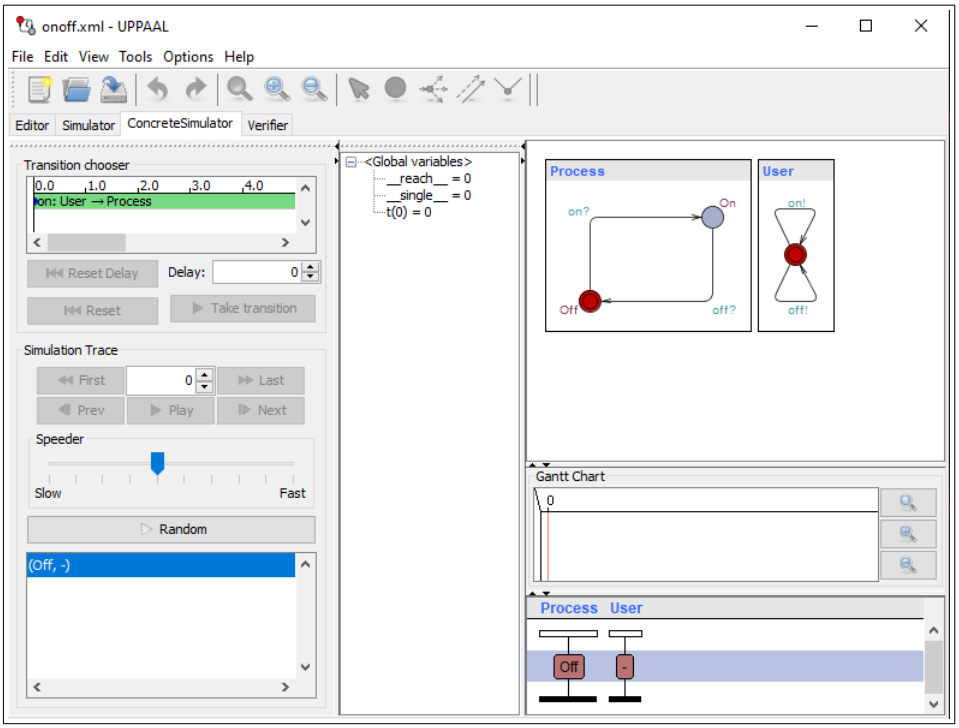
UPPAAL poskytuje dva simulátory: obyčejný a konkrétní. Hlavní rozdíl je v tom, že konkrétní simulátor zahrnuje čas v simulaci. Na obrázku 3.7 je ukázán obyčejný simulátor. Sekce *Enabled Transition* zobrazuje všechny možné přechody, zatímco *Simulation Trace* dokumentuje provedené přechody. Simulátor poskytuje informaci o hodnotách všech lokálních a globálních proměnných včetně hodin. Všechny automaty, které jsou definovány v systému, jsou zobrazeny v pravé části. Aktuální stavy jsou označeny červeně. V pravé dolní části jsou znázorněny stavy, ve kterých se automaty nacházely, na základě provedených přechodů.



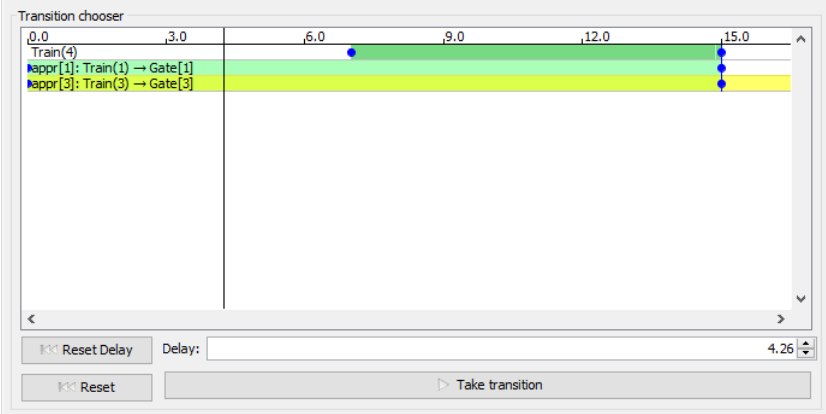
Obrázek 3.7: Simulátor.

Konkrétní simulátor umožňuje sledovat běh systému s náhledem na čas. Při výběru přechodů v konkrétním simulátoru je časový okamžik, ve kterém se přechod aktivuje definován tím, kam uživatel klikne při aktivaci přechodu (viz. obrázek 3.8).

Přechody jsou označené různou barvou podle situací (viz. obrázek 3.9): červená znamená, že v tuto chvíli žádný přechod nemůže být proveden z důvodu omezení; zelená umožňuje provést přechod bez porušení jakýchkoli stráží; bílá barva znamená, že v tento časový okamžik přechod nemůže být proveden, ale je možné jeho provedení před nebo po určitém čase; žlutá barva znamená, že tento přechod je vybrán pro provedení (pomocí jednoho kliknutí myší, pro samotné provádění je potřeba kliknout dvakrát po sobě).



Obrázek 3.8: Konkrétní simulátor.

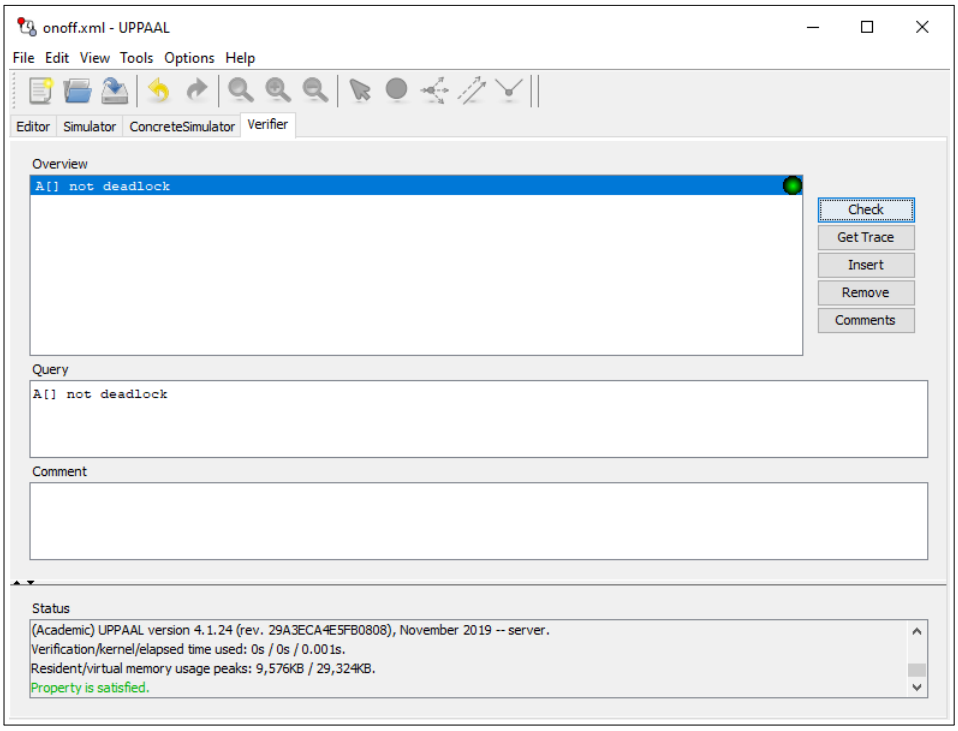


Obrázek 3.9: Pole pro výběr přechodu v Konkrétním simulátoru.

Verifikátor

Verifikátor se používá k ověření systému pomocí logických výrazů. Nástroj je navržena ke kontrole invariantních a dosažitelných vlastností, zejména zda jsou určité kombinace řídících uzlů a omezení na hodinách a celočíselných proměnných dosažitelné z počáteční konfigurace. Pro tento účel nástroj UPPAAL používá zjednodušenou verzi Logiky Výpočetního Stromu (angl. Computational Tree Logic, CTL). Stejně jako CTL, dotazovací jazyk se skládá ze stavových formulí a formulí hran mezi stavy. Stavové formule popisují jednotlivé stavy, ve

kterých se model může nacházet. Pomocí formule cest lze sledovat vlastnosti dosažitelnosti (angl. reachability), bezpečnosti (angl. safety) a živostí (angl. liveness).



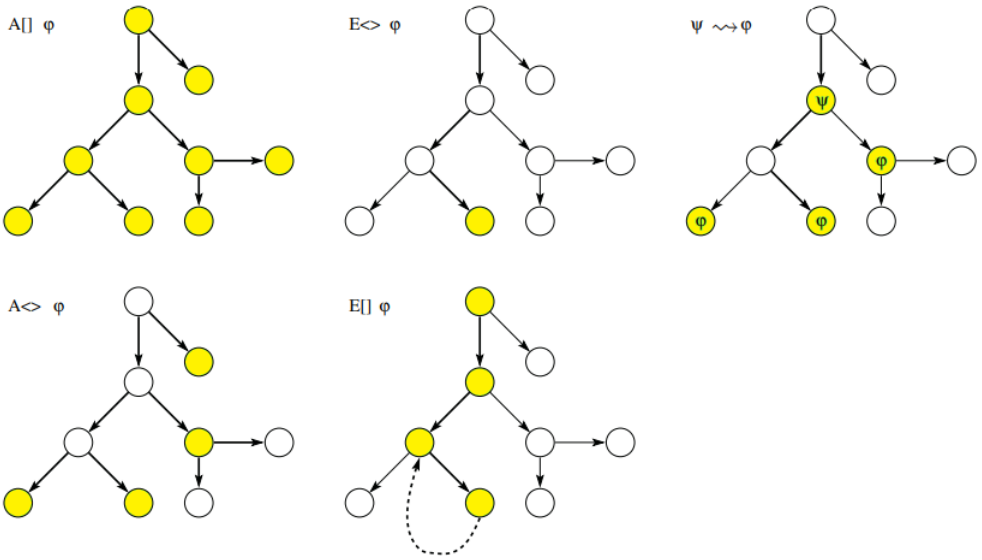
Obrázek 3.10: Verifikator.

Na obrázky 3.11 jsou zobrazené možné dotazy, které podporuje nástroj UPPAAL. Žluté místa jsou místa, které splňují ϕ.

Dostupné pravidla dotazovacího jazyka:

- `A<>` `p`: Pro všechny cesty podmínka p někdy platí.
- `A[]` `p`: Pro všechny cesty podmínka p vždy platí.
- `E<>` `p`: Existuje cesta, kde podmínka p někdy platí.
- `E[]` `p`: Existuje cesta, kde podmínka p vždy platí.
- `p` $\rightarrow$ `q`: pokud je někdy splněna podmínka p, poté bude q také splněna.

Nejdůležitější dotaz v tomto nástroji je `A[] not deadlock`, který ověřuje, zda systém je deadlock free, tz, situace, kde neexistují žádné přechody z aktuálních stavů, nebo z žádného stavu po nějakém zpoždění. Pokud po spuštění dotazu výsledek bude vyhodnocen jako pravdivý, zvýrazní se to zelenou barvou, jinak červenou. UPPAAL zjistí, která cesta způsobila negativní výsledek dotazu.



Obrázek 3.11: Dotazovací jazyk. Převzato z [4].

Generátor testovacích případů (YGGDRASIL)

Nástroj UPPAAL nabízí možnost offline generování testovacích případů s cílem zvýšit pokrytí hran. Yggdrasil generuje cesty z modelu a převádí je do testovacích případů na základě testovacího kódu, zadaného do modelu na konkrétní místa a hrany. Pro využití generátoru testovacích případů je potřeba pracovat s deterministickým modelem a ověřit, zda systém je deadlock free (dotaz `A [] not deadlock`). Poskytuje tři režimy pro generování testovacích případů:

- Na základě dotazu: generuje testovací případ určený dotazem `E<>`. Lze vybrat všechny nebo konkrétní dotaz, které jsou zde představeny. Někdy se může stát, že po přidání dotazu do Verifikátoru a přepnutí do okna generátoru testovacích případů, nebude nový dotaz zobrazen. V tomto případě je potřeba aktualizovat Simulátor, a to pomocí tlačítka F5. Tento režim je určen pro výpočet optimální délky testovacích případů nebo jeho doby trvání.

- Depth-first: používá heuristiku náhodného vyhledávání na zadaný počet kroků. Pro využití tohoto režimu je potřeba deklarovat globální proměnné `int __reach__ = 0;`. Počet kroků závisí na velikosti a hloubce modelu. Pokud je hloubka nastavena na malé číslo, existuje riziko, že některé hrany nebudou pokryty testem. Pokud je hloubka nastavena na příliš velké číslo, vygenerují se zbytečně dlouhé testovací případy. Generuje několik testovacích případů, dokud se pokrytí nezlepší. Pro malé systémy obvykle generuje dva nebo tři testovací případy.

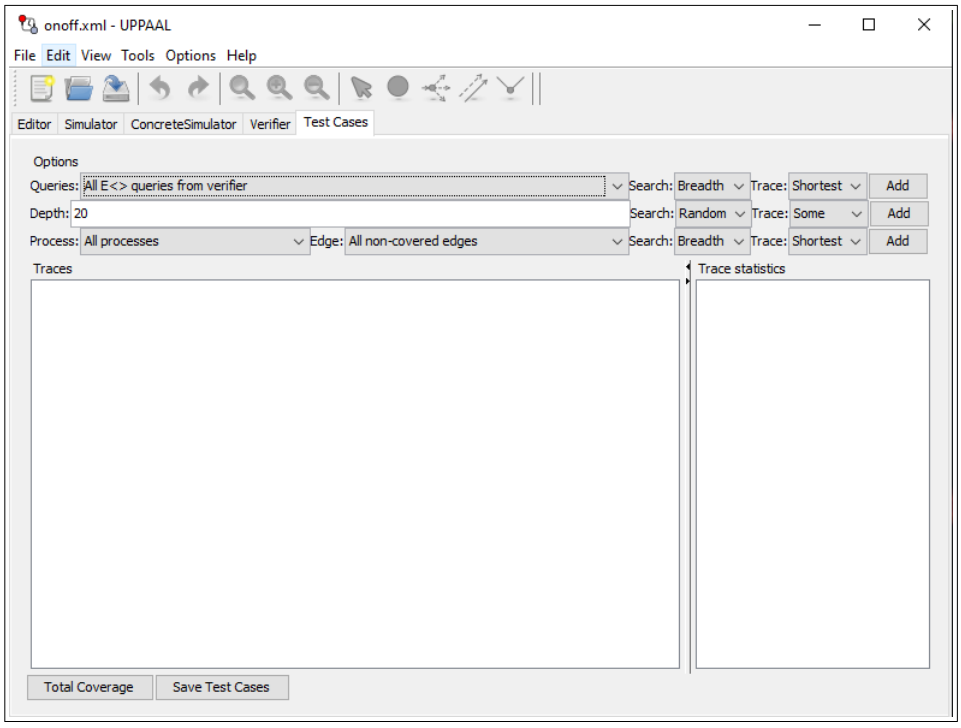
- Single-step: generuje individuální testovací případy na každou hranu, která nebyla pokryta dříve vygenerovanými cestami. Podobné předchozímu režimu, single-step potřebuje deklarovat globální proměnnou `int __single__ = 0;`. Je možné nastavit, které procesy je potřeba projít a pokrýt jejich nekryté hrany.

Každý variant generátorů má dva parametry: *Search* a *Trace*.

- *Search* určuje pořadí hledání ve stavovém prostoru. *Breadth* je určen pro systémy, které se skládají z velkého počtu paralelních procesů, *Depth* se používá při dlouhé posloupnosti přechodů v systému. *Random* je podobné režimu *Depth*, s tím rozdílem, že náhodně určuje nedeterministické možnosti přechodu.
- *Trace* určuje kritérium optimalizací. *Shortest* vyhledá cestu, která se skládá z nejmenšího počtu kroků či provedených přechodů, *Fastest* je určen pro nejrychlejší cesty, které zabírají co nejméně času, a *Some* je první nalezená cesta.

YGGDRASIL je navržen tak, aby generování testovacích případů probíhalo dle níže uvedeného postupu:

- Vygenerovat testovací případy na základě dotazu.
- V případě nedostatečného pokrytí, lze použít *depth-first* pro generování testovacích případů na zadanou hloubku. Tento proces se opakuje, pokud nová vygenerovaná cesta nepřidá nové pokrytí oproti předchozím cestám.
- Pro pokrytí zbývajících hran, které nebyly pokryty v předchozích krocích, použít *single-step* režim, který přidá pro každou z těchto nekrytých hran samostatný testovací případ.

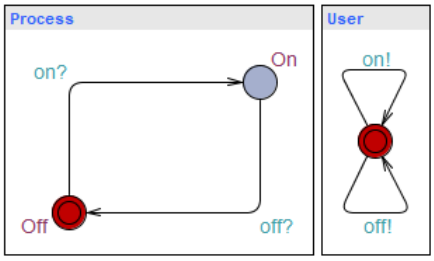


Obrázek 3.12: YGGDRASIL.

Hlavní výhoda nástroje je generování testovacích případů, které se zapisují do spustitelného kódu (na základě kódu, který je zadán u míst a hran v modelu). Testovací kód se zadává jako doslovný text, což dovoluje použít tento nástroj pro jakýkoliv programovací jazyk. Zároveň je potřeba definovat `Prefix` a `Postfix` kódů, které definuje začátek a konec testovacího souboru. V `Prefix` lze také přidat informace o názvu a příponě vygenerovaného souboru, a to pomocí `TEST_FILENAME` a `TEST_FILEEXT`. U míst a hran je možné zadat hodnotu proměnné, která bude předána do testovacího kódu, a to pomocí symbolu `$`, např. `$(var)` pro převod globální proměnné nebo `$(Process.var)` pro převod lokální proměnné z procesu.

3.4 Příklad generování testovacích případů

Dále bude popsané využití generátoru testovacích případů na příkladě systému On/Off, který je dostupný z oficiálních demo příkladů nástrojů UPPAAL.



Obrázek 3.13: Systém On/Off.

Celý systém se skládá ze dvou modelů 3.13: model systému (na obrázku vlevo), který mění svůj stav na `On` nebo `Off` a model uživatele (na obrázku vpravo), který nedeterministicky mění stav systému na `On` nebo `Off`. Pro generování testovacích případů je potřeba deklarovat globální proměnné, které umožní generátoru používat techniky *depth-first* a *single-step*. Tyto proměnné se v modelu nesmí měnit.

```
int __reach__ = 0;
int __single__ = 0;
```

Zároveň v deklaraci systému je nutné přidat `TEST_PREFIX` a `TEST_POSTFIX`, které určují začátek a konec výstupního kódu. V tomto příkladě každý testovací případ představuje třídu `Test` s metodou `main`, což dovoluje přímo spustit vygenerovaný kód. Kód prefixu a postfixu představuje obálku pro nastavení této třídy.

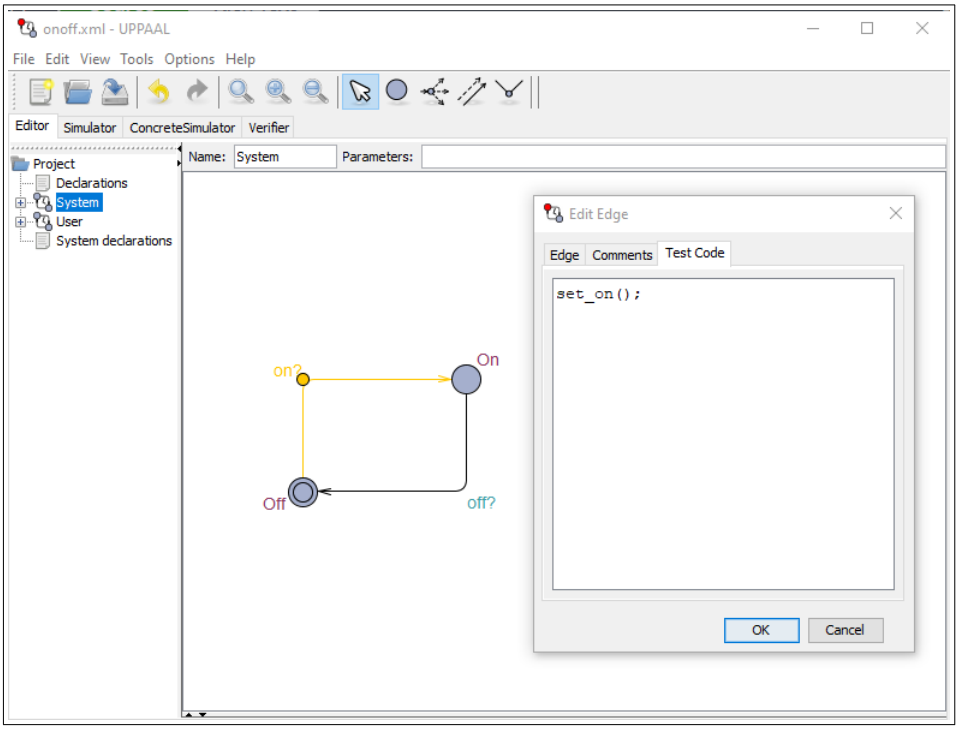
```
/** TEST_PREFIX
    package app;

    import app.App;

    class Test extends App {
    public static void main(String[] args) {
*/
```

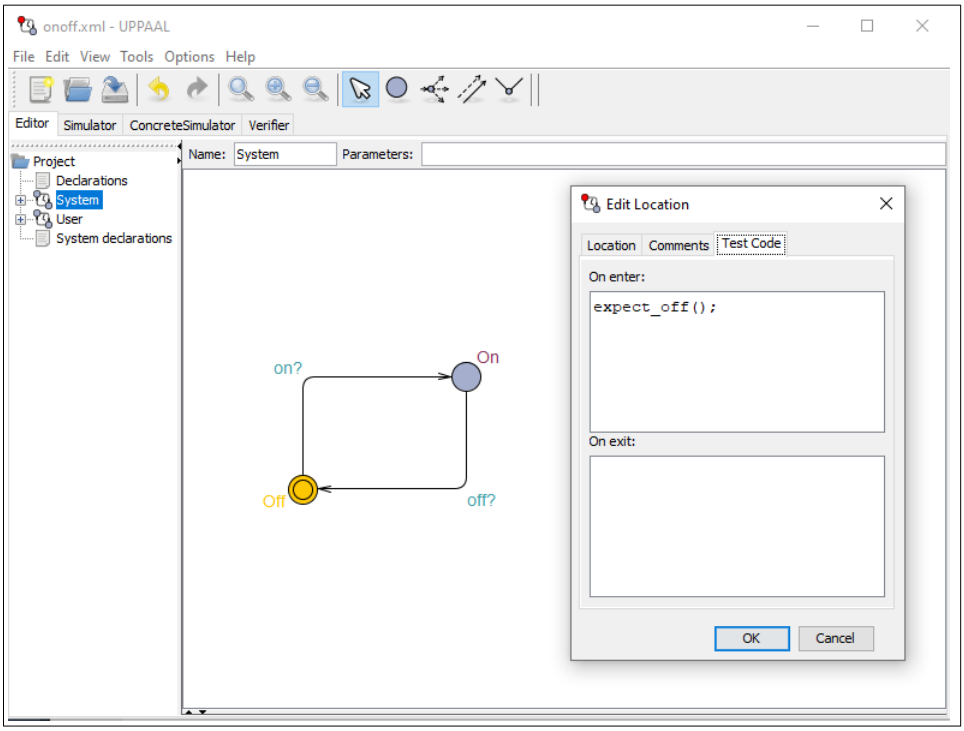
```
/** TEST_POSTFIX
    }
}
*/
```

Pro vytvoření hlavní části testovacího kódu musí být k modelu systému přidán zdrojový kód. Tohle lze zajistit ve vlastnosti *Test code* každého stavu nebo hrany. V daném příkladě je potřeba doplnit model systému. Hrana, která je označena `on?` je doplněna kódem `set_on();`. Tím se provede metoda `set_on()` v testovaném programu, kdykoli se provede přechod této hrany v testovacím případu. Podobně hrana `off?` je doplněna kódem `set_off();`. V případě míst lze definovat kód ve dvou fázích: na vstupu do stavu (angl. Enter) nebo na výstupu ze stavu (angl. Exit). Zde na vstupu stavu `OFF` je přidán kód `expect_off();` a na vstupu stavu `ON` je přidán kód `expect_on();`.



Obrázek 3.14: Přidání testovacího kódu do hrany.

Před generováním testovacích případů, je nutné ověřit systém na přítomnost deadlocku a to pomocí verifikátoru. Navíc je přidán dotaz `E<> Process.On`, který ověřuje, zda v systému existuje cesta k stavu `On`. Tento dotaz bude poté využit generátorem.

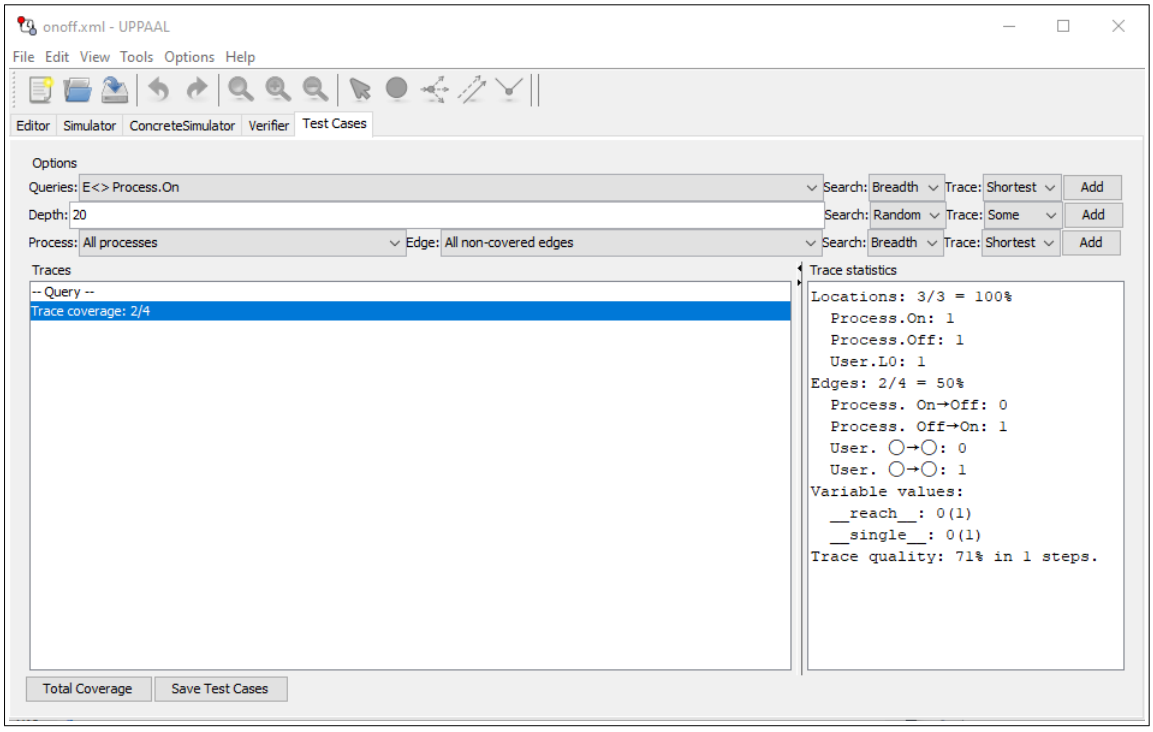


Obrázek 3.15: Přidání testovacího kódu do stavu.

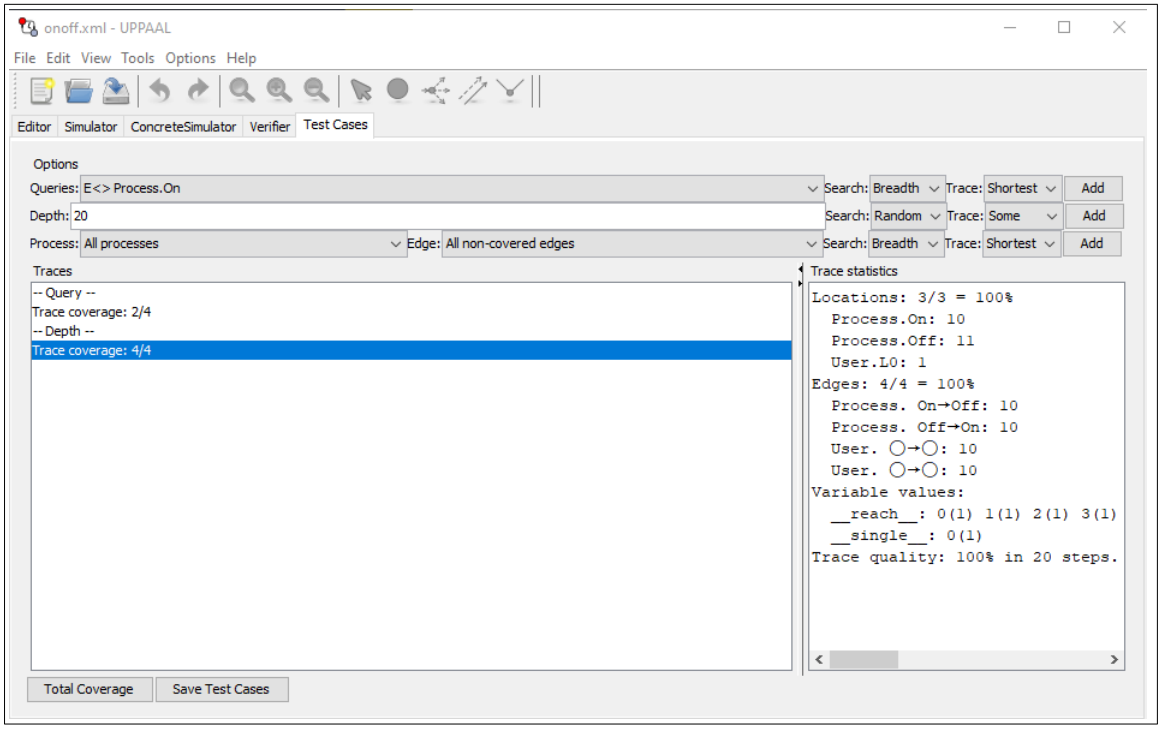
Pro spuštění gui generátoru je nutné ho zapnout pomocí tlačítka *Tools*, které se nachází na horní části nástrojů UPPAAL, zatímco je potřeba vybrat *Test cases*. Dále bude využit doporučený postup pro generování testovacích případů:

- Generování cesty pomocí zadaného dotazu. Zde je použít dotaz `E<> Process.On`. Vygenerovanou cestu lze sledovat na obrázku 3.16. Pomocí dotazů byl vygenerován testovací případ, který pokrývá 2/4 cesty systému. Vpravo je zobrazena statistika cesty. Zde lze zjistit, že byla pokryta všechna místa (`Locations: 3/3 = 100 %`), ale cesta pokryla jenom část hran (`Edges: 2/4 = 50%`), z toho vyplývá celková kvalita vygenerované cesty (`Trace quality: 71 %`).
- Pro vylepšení pokrytí je potřeba vygenerovat cestu na zadanou hloubku, v našem případě byla vybrána hloubka `20`. Díky vygenerované cestě bylo dosaženo 100% pokrytí systému, tz. 100% pokrytí hran a stavu (viz obrázek 3.17).
- V tomto příkladě 100% pokrytí bylo dosaženo již v druhém kroku, proto dále není možné generovat novou cestu pomocí techniky *single-step*.

Pro získání testovacích případů ve formě spustitelného testovacího souboru je potřeba uložit testovací případy do složky s testovaným programem, a to pomocí tlačítka *Save Test Cases*. Na výstupu jsou vygenerovány dva soubory se zadaným názvem a sekvenčním číslováním. V tomto příkladě každý soubor obsahuje uvnitř třídu Test, která zahrnuje posloupnost metod vyvolaných vygenerovanými cestami. Tyto soubory je možné spustit jednotlivě a podívat se na chování testovaného programu. Při spuštění testovacího souboru by se měla zobrazit chybová hláška, pokud najde jakoukoliv chybu v programu.



Obrázek 3.16: Generování cesty pomocí dotazů.



Obrázek 3.17: Generování cesty pomocí vybrané hloubky.

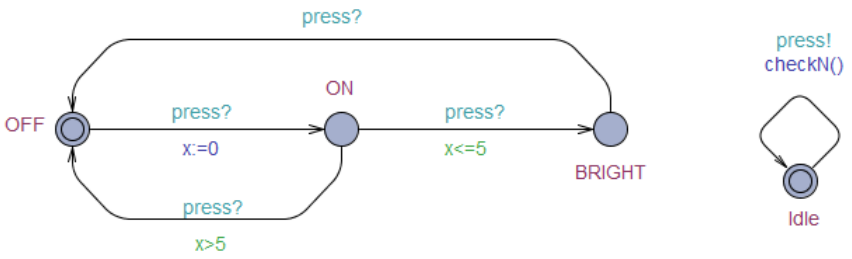
Kapitola 4

Návrh a implementace řešení

V této kapitole jsou popsané vybrané systémy, pro které budou vytvořeny a následně aplikovány testovací případy z nástrojů UPPAAL. Pro naše účely byly zvoleny tři systémy: vypínač světla, 2-bitová násobička (přesná a přibližná) a výtah. Cílem této kapitoly je ukázat vytvořené modely a popsat jejich základní algoritmus práce.

4.1 Vypínač světla

První systém, který byl modelován, je systém vypínače světla, který byl převzat z [4]. Na rozdíl od obyčejného vypínače, vybraný model poskytuje možnost regulace intenzity osvětlení. Pokud je světlo vypnuté a uživatel stiskne tlačítko, pak se světlo rozsvítí. V případě, že uživatel stiskne tlačítko ještě jednou, pak se světlo vypne. Avšak, pokud je světlo vypnuté a uživatel rychle stiskne tlačítko dvakrát, světlo se rozsvítí do jasna.



Obrázek 4.1: Systém vypínače světla. Šablona Light (vlevo) a Button (vpravo).

Systém se skládá ze dvou modelů: model světla `Light` a model uživatele `Button`. Světlo má tři stavy: vypnuté `OFF`, zapnuté `ON` a jasné `BRIGHT`. Model je rozšířen o časovou proměnnou `x`, která sleduje čas stisknutí tlačítka. Pokud uživatel stiskne jednou tlačítko, jsou tři možnosti: uživatel stiskne tlačítko ještě jednou a to do 5 časových jednotek, světlo se rozsvítí do jasna; uživatel stiskne tlačítko po 5 časových jednotkách, pak světlo se vypne; uživatel nestiskne tlačítko a systém zůstává ve stavu zapnuto. Model uživatele umožňuje stisknout tlačítko. Pro naše účely, byl model rozšířen o proměnnou `n`, která sleduje počet stisků tlačítek. S ohledem na omezení typu `int` byla přidána funkce, která po 100 stisknutí vynuluje proměnnou `n`. Systém byl také rozšířen, a to o časovou proměnnou `y`, která sleduje celkový čas běhu systému.

Testovaný systém je napsán v jazyce Java. Skládá se z 4 souborů: soubor `Main.java`, který obsahuje hlavní metodu pro spuštění programu; soubor `Light.java`, který obsahuje třídu `Light`, na kterou budou aplikovány testovací případy; pro mutační testování byl vytvořen soubor `LightM.java`, který obsahuje mutaci v metodě `isOff()`. Při správném chování metoda ověřuje, zda aktuální stav je `OFF` a v případě jiného stavu se zobrazí chyba. Mutace spočívá v změně kontroly, kde namísto stavu `OFF` metoda čeká na stav `ON`; `LightC.java` je soubor, který obsahuje správnou verzi zdrojového kódu. Podle spuštěného skriptu do souboru `Light.java` bude kopírován obsah buď `LightC.java` nebo `LightM.java`.

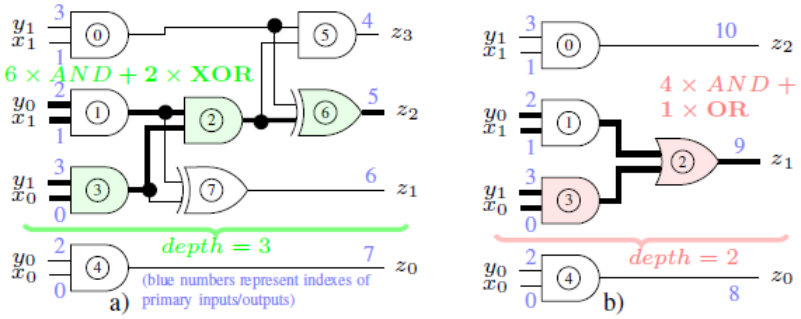
Pro generování testovacích případů v nástrojích UPPAAL byl k systému přidán testovací kód: systémová deklarace obsahuje `TEST_FILEEXT` pro generaci testovacího souboru ve formátu `.java`, `TEST_PREFIX` s třídou `Test` a metodou `main`, což dovoluje spuštění jednotlivých testů samostatně, a `TEST_POSTFIX`. Model světla je také doplněn kódem. Každý stav obsahuje metodu, která kontroluje, zda aktuální stav systému odpovídá stavu třídy. Hrany jsou doplněny vyvoláním metody, která změní aktuální stav na stav, do kterého tato hrana vede. Např. přechodu ze stavu `OFF` do stavu `ON` odpovídá posloupnosti vyvolaných metod:

```
isOff();
turnOn();
isOn();
```

Při návrhu systémů a testovacích případů bylo zjištěno omezení v přidávání hodnot do testovacího kódu. Nástroj UPPAAL dovoluje zapsat do testovacího kódu jenom globální a lokální proměnné typu `int` nebo `bool`. V daném systému by bylo nejlepší odesílat čas stisknutím tlačítka, aby optimalizoval práci testovaného systému. Kvůli tomu, že je čas v nástrojích reprezentován ve formě omezení, není možné získat jeho přesnou hodnotu. Proto bylo rozhodnuto udělat pro každou hranu systému v UPPAAL odpovídající metodu v testované třídě `Light`.

4.2 2-bitová násobička

Následující systém, pomocí kterého budou vygenerovány testovací případy je systém 2-bitové násobičky (model, spolu se zdrojovým kódem pro nástroj UPPAAL, jsou převzaty z [47]). Na obrázku 4.2 jsou ukázány logické obvody těchto násobiček.



Obrázek 4.2: Přesná (vlevo), přibližná (vpravo) násobička. Převzato z [47].

Pro vstupní a výstupní bity jsou používány globální proměnné. Konstantní proměnné typu `int`: `NPI` počet vstupních bitů, `NPO` počet výstupních bitů, `PIxy[NPI]={0, 1, 2, 3}`

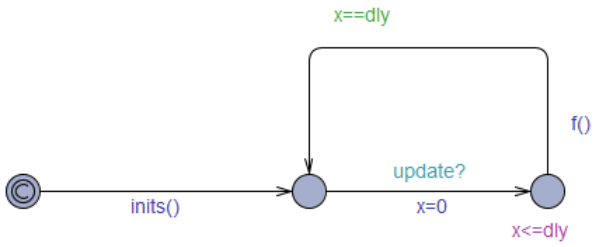
je pole sdílených vstupních indexů násobičky, `POx[NPO]={4, 5, 6, 7}` je pole výstupních indexů přesné násobičky a `POy[NPO] = {8, 9, 10, -1}` je pole výstupních indexu přibližné násobičky. Pro aktuální stav všech vstupních, výstupních a pomocných proměnných je definované bitové pole `bits[MAX_BITS]`, které využívá výše zmíněné konstantní proměnné pro uchování jednotlivých dat.

x_1x_0 \ y_1y_0	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001 0111
	$z_3z_2z_1z_0$			

Obrázek 4.3: Pravděpodobnostní tabulka násobiček. Převzato z [47].

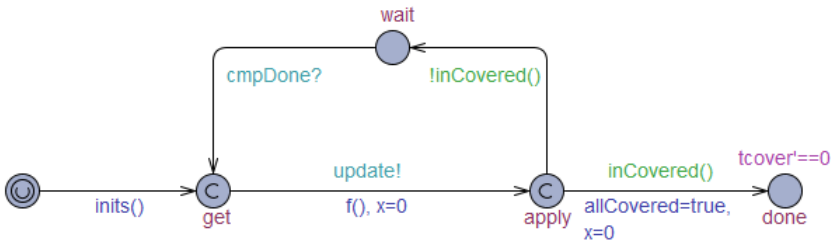
Systém se skládá z 6 šablon:

`tmul2any` popisuje model přesné násobičky. Po inicializaci čeká na signál `udpate?`, který označuje přidání nových stimulů pro zpracování. Po obdržení potřebného signálu, model čeká maximálně `DLY_MUL2` časových jednotek, poté vyvolává funkci `f()`, která zpracuje stimuly a zapíše výstupní hodnoty podle pravdivostní tabulky.



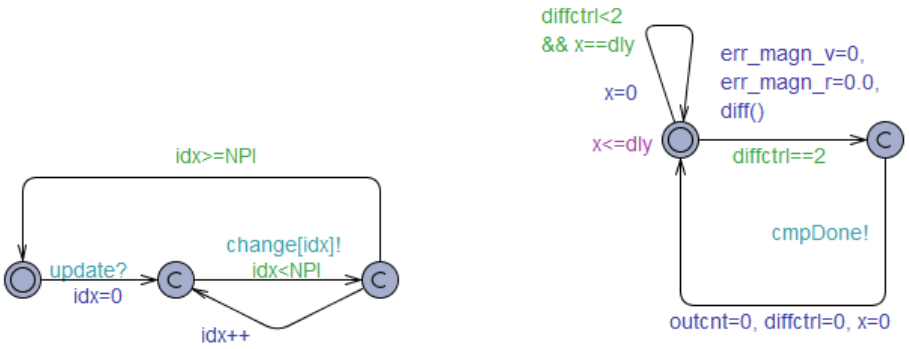
Obrázek 4.4: Šablona tmul2any.

`tmul2_tb_exhaust` je model generátoru stimulů pro `x` a `y`. Po inicializaci, model vyvolá funkci `f()`, která vyprodukuje stimuly podle uvedených specifikací, v našem případě se jedná o postupné přidávání `1`, tz. vstupní stimuly budou vypadat následně: `0000`, `1000`, `0100`, `1100`, atd. Zároveň model posílá signál `update!`, který sděluje ostatním prvkům systému přidání nových stimulů pro zpracování. Stimuly jsou generované, dokud nebudou pokryty všechny možné varianty stimulů nebo pokud nenastane chyba. V případě, že nebylo dosáhnuto 100% pokrytí pomocí `inCovered()`, se model převede do stavu `wait` a čeká na signál `cmpDone?`, který udává ukončení zpracování stimulů násobičkami a ukončení porovnávání výstupů z nich. Poté model pokračuje v generování nových stimulů.



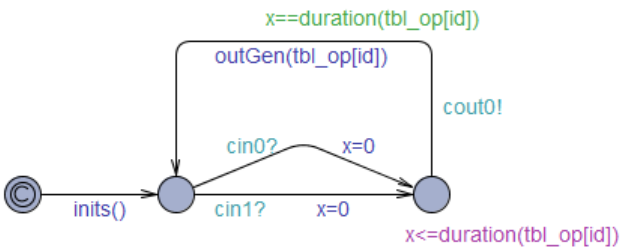
Obrázek 4.5: Šablona tmul2_tb_exhaust.

`syncPrimary` slouží pro oznámení jednotlivým hradlům o přidání nových stimulů. Model čeká na aktualizaci stimulů, poté pomocí kanálu `change[idx]!` oznámí všem definovaným hradlům v systému o této události, čímž zahájí zpracování získaných vstupu.



Obrázek 4.6: Šablona syncPrimary (vlevo) a eval_diff (vpravo).

`eval_diff` kontroluje shodu získaných výstupních hodnot z přesné a přibližně násobičky. Model čeká, dokud nejsou k dispozici výstupy ze dvou násobiček. Po jejich obdržení, model porovnává výstupy pomocí funkcí `diff()` a následně odesílá signál `cmpDone!`, který dovoluje generátoru stimulů pokračovat svůj běh.



Obrázek 4.7: Šablona gate2.

`gate2` je šablona pro jednotlivé hradla se dvěma vstupy a jedním výstupem. Po inicializaci, model čeká na signál `cin0?` nebo `cin1?`, který označí změnu stavu jednoho ze vstupů. Dále je model zpožděn a po uplynutí času vypočítá svůj výstup a označí událost pomocí zaslání signálu `cout0!`. V našem případě pomocí sady modelů z této šablony, je vytvořena přibližná násobička, která je představena na obrázku 4.2.

Pro implementaci 2-bitovych násobiček bylo rozhodnuto použití programovacího jazyku Verilog. Byly vytvořeny dva soubory: `accurate.v`, který obsahuje modul přesné násobičky a `approximate.v`, který obsahuje modul přibližné násobičky. Každý z těchto modulů přijímá na vstup dvě 2-bitové proměnné `x` a `y` a proměnnou pro ukládání výsledků. V případě přesné násobičky se výsledek ukládá do 4-bitové proměnné, pro přibližnou násobičku stačí pouze 3-bitová proměnná pro uložení výstupu.

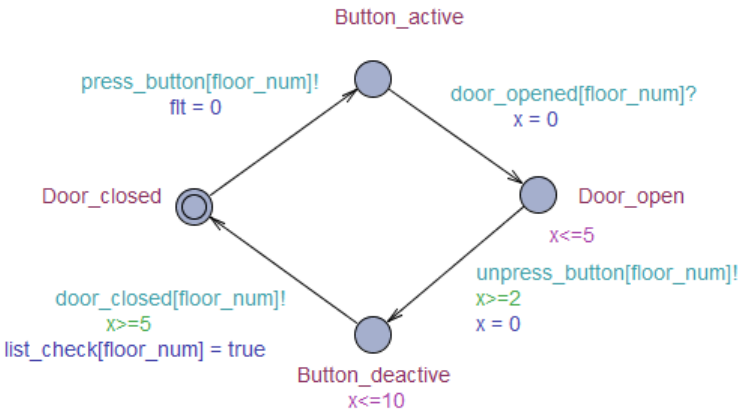
Stejně jako u systému vypínače světla, bylo zjištěno omezení přidávání různých parametrů do testovacího kódu. Nástroj UPPAAL neumožňuje přidat jednotlivá data z pole, i když tato data jsou typu `int` a `bool`. Proto bylo pro všechny vstupní a výstupní bity z násobičky nutné udělat jednotlivé proměnné, které jsou poté odeslány do testovacího souboru. Model `tmul2_tb_exhaust`, po změně stimulů pomocí funkcí `f()`, ukládá každý z nich do pomocných lokálních proměnných `hin0`, `hin1`, `hin2`, `hin3`, poté co systém vstoupí do stavu `apply`, přidá jednotlivé vstupní bity testovacímu kódu `hin0` odpovídá `X[0]`, `hin1` - verb|X[1]|, `hin2` - `Y[0]` a `hin3` - `Y[1]`. Pro kontrolu, zda implementované modely ve Verilog odpovídají správnému chování násobiček, je potřeba přidat do testovacího kódu výstupní signály ze dvou násobiček, které byly získané systémem v nástroji UPPAAL. Toto lze zajistit pomocí modelu `ediff`, který čeká na výsledek z obou násobiček, a pak porovnává jejich výstup. Pokud model obdrží signál, že přesná a přibližná násobička zpracovali signály a je možné porovnat jejich výstup, zároveň s tím proběhne ukládání jednotlivých bitů z výstupu do lokálních proměnných pro přibližnou násobičku `test_app_z0` (odpovídá výstupu `Z[0]`), `test_app_z1` (`Z[1]`), `test_app_z2` (`Z[2]`), a přesnou násobičku `test_acc_z0` (`Z[0]`), `test_acc_z1` (`Z[1]`), `test_acc_z2` (`Z[2]`), `test_acc_z3` (`Z[3]`). Po provedení přechodů v modelu, proběhne odeslání vygenerovaných výstupních bitů do testovacího souboru, a poté proběhne kontrola, zda výsledky z nástrojů UPPAAL odpovídají výsledkům z programu ve Verilog.

4.3 Výtah

Poslední systém, který byl implementován, je systém zjednodušeného výtahu. Zahrnuje samotný výtah a jednotlivá patra, mezi kterými se výtah může pohybovat. Ačkoli systém představuje zjednodušenou verzi výtahu, podporuje zmáčknutí tlačítka kdykoli během pohybu výtahu. Byly vytvořeny 4 šablony modelu: `Floor`, `Controller`, `Elevator` a `Engine`. Globální deklarace obsahuje proměnnou `N`, která udává počet pater v budově. V našem případě se jedná o budovu, která obsahuje 4 patra. Číslo patra začíná od 0, tz. poslední patro je 3.patro. Navíc globální deklarace zahrnuje pole `bool list_ckeck[N]` pro záznam stisknutých tlačítek (např. na 0.patře bylo stisknuté tlačítko → `list_check[0] = true`), proměnnou `current_floor` pro označení aktuálního patra, na kterém se nachází výtah a proměnná `moving_direction`, která ukazuje, jakým směrem se pohybuje výtah. `moving_direction` může nabývat 3 hodnot: `0` - výtah jede nahoru, `1` - výtah jede dolů a `2` - výtah stojí na místě.

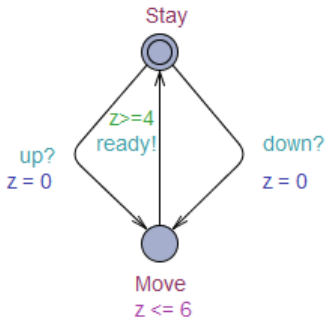
Šablona `Floor` slouží pro popis chování tlačítka a dveří na jednotlivých patrech. Při vytvoření systému dostává model konstantní parametr `floor_num`, který odpovídá číslu pátra. Chceme-li zmáčknout tlačítko a tím zavolat výtah na potřebné patro, je nutné provést

přechod `press_button[floor_num]!`. Zároveň s tím je poslán signál modelu `Controller`, který přidá požadavek do fronty. Poté model čeká, dokud nedostane signál, že se výtah dostavil na toto patro. V případě, že je výtah tady, proběhne simulace otevření dveří a odemknutí tlačítka. Spolu s tímto odešle se signál `unpress_button[floor_bum]!`, který říká, že požadavek o přivolání výtahu byl splněn. Pak model výtahu čeká na hodinovou proměnnou `x` (tím je simulovaná doba, během které jsou dveře výtahu otevřené) a prozatím se dveře zavřou. Zároveň je model doplněn o hodinovou proměnnou `flt` pro sledování celkové doby čekání na výtah, tz. při stisknutí tlačítka, se hodinová proměnná `flt` vynuluje, a pak běží synchronně spolu s jinými hodinami.



Obrázek 4.8: Šablona Floor.

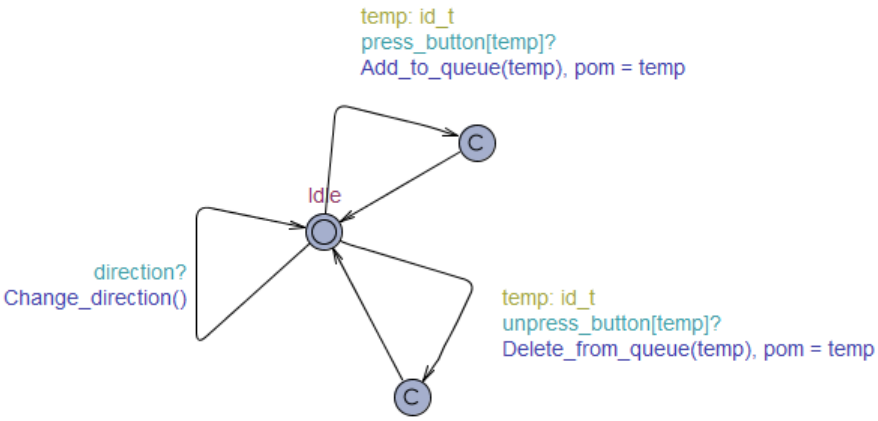
`Engine` slouží pro simulaci pohybu výtahu za použití časově proměnné `z`.



Obrázek 4.9: Šablona Engine.

`Controller` přijímá informace o stisknutí tlačítka nebo jejich odemknutí. V případě stisku model vyvolá funkci `Add_to_queue()`, která zapíše požadavek do globálního pole `list_check[N]` a do lokálního pole `temp_list`, které slouží pro ukládání požadavků v pořadí jejich výskytu. Pro ověření správnosti ukládání dat, funkce používá lokální proměn-

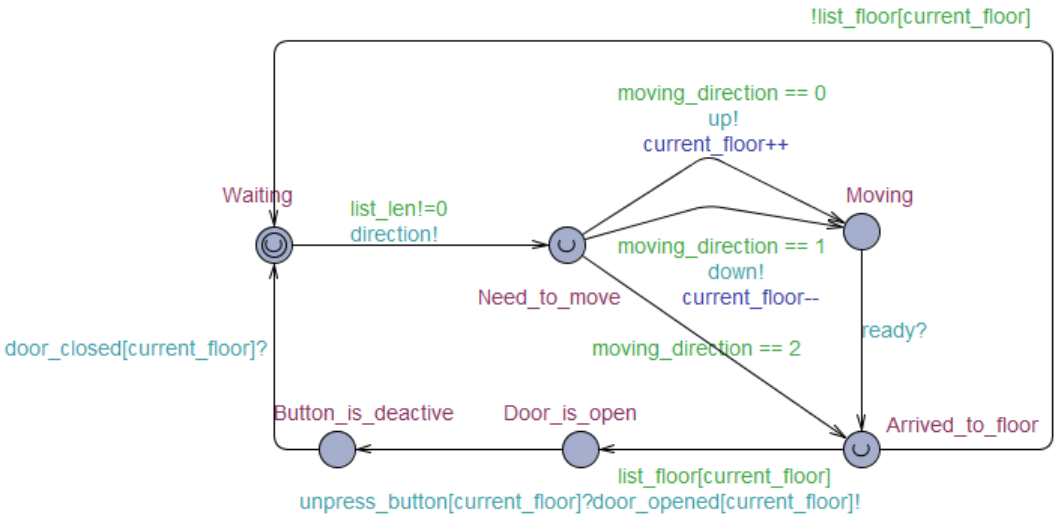
nou `temp_len` pro ukládání počtu požadavků, které je porovnatelné s globální proměnnou `list_len`. Pokud model získá signál o zpracování požadavku, poté je vyvolána funkce `Delete_from_queue`, která vymaže zpracované patro z výše zmíněných polí a aktualizuje počet pater, které čekají na zpracování. Zároveň model může dostat signál `direction?` pro aktualizaci směru pohybu výtahu.



Obrázek 4.10: Šablona Controller.

`Elevator` reprezentuje algoritmus práce výtahu, který vypadá následně: pokud fronta požadavků není prázdná, signál `direction!` se odešle pro získání směru pohybu. Podle toho, kde se nachází samotný výtah a první patro ve frontě čekajících na zpracování (zde je použita fronta `temp_list`, která řídí pořadím stisknutých tlačítek), je rozhodnuto, zda výtah jede nahoru, dolů či stojí na místě. V případě pohybu, model změní číslo aktuálního patra a čeká na simulaci pohybu. Poté se model nachází ve stavu `Arrive_to_floor` a je nutné zkontrolovat, je-li na tomto patře stisknuté tlačítko. Pokud není stisknuté tlačítko, model se vrátí do inicializačního stavu a opakuje celý tento algoritmus, dokud fronta není prázdná. V případě, že na aktuálním patře bylo stisknuté tlačítko, je odeslán signál modelu `Floor` pro simulaci akce otevření a zavření dveří. Pokud jsou dveře zavřené, model se vrátí do inicializačního stavu a pokračuje svůj běh v případě čekajících požadavků.

Např. může nastat situace, kde se výtah nachází na 1.patře. Poté někdo stiskne tlačítko na 3.patře, čímž přidá nový požadavek do odpovídajících front a výtah začne svůj pohyb směrem nahoru. Zároveň někdo stiskne tlačítko na 2.patře ještě v době pohybu výtahu z 1.patra na 2.patro. V tomto případě, při dorážení na 2.patro, by výtah měl zastavit z důvodu optimalizace běhu, a jenom poté pokračovat svou cestu do 3.patra. Daný model právě zahrnuje tuto optimalizaci, díky čemuž doba čekání je poměrně snížena na rozdíl od situací, kdy by výtah na začátku jel do 3.patra, a poté by se vrátil na 2.patro.



Obrázek 4.11: Šablona Elevator.

Testovaný systém výtahů je implementován v programovacím jazyce C. Program se skládá ze dvou souborů: `elevator.h`, který obsahuje všechny potřebné funkce pro pohyb výtahu, a `main.c` s tělem programu. Systém obsahuje dvě struktury: struktura výtahů, která se skládá z aktuálního patra, na kterém se nachází výtah (`current_floor`) a směr pohybu výtahu (`moving_direction`); struktura fronty, která sestává z front požadavků podle pořadí stisknutí tlačítka (`temp_list`) a bitového pole (`list_floor`), stejně jako je to implementované u systému v nástrojů UPPAAL. Proces výtahu je implementován pomocí vlákna. Funkce `Elevator_controller` definuje práci výtahu a to pomocí smyčky. Výtah čeká na příchozí požadavky, a v případě, že fronta není prázdná, začíná svůj pohyb. Algoritmus zpracování požadavků je identický k algoritmu, který se používá v UPPAAL. Pokud výtah obdrží signál `stop_signal`, skončí svou práci a tím se skončí běh celého systému. Signál se posílá na konci `main`, pokud jsou zpracované všechny požadavky, které byly odeslány předtím. Pro simulaci pohybu výtahu při výskytu nějakého děje je použita funkce `sleep()`. Např. pokud výtah jede na horu, proces běhu se pozastaví na 5 časových jednotek a to pomocí `sleep(5);`.

Pro vytvoření spustitelných testovacích případů do systémové deklarace, podobně jako u vypínače světla, byly přidány `TEST_FILEEXT .c`, aby výstupní spustitelný soubor byl v jazyce C a prefix a postfix programu. Při implementaci testovacího kódu, který bude předán testovanému systému, se vyskytuje omezení. Oba systémy (systém v jazyce C vytvořený v UPPAAL) pracují s časem. Již při popisu vypínače světla bylo zmíněno omezení o přidávání dat z nástrojů UPPAAL do spustitelného testovacího případu, konkrétně neschopnost nástrojů předat data o běhu hodin. Proto nastává situace, kde je obtížné spustit testovaný systém tak, aby simulace běžela synchronně s tím, co je navrženo v UPPAAL. To znamená, že není možné porovnat správný hodinový běh dvou vytvořených systémů. Jedna z možností je přidat testovací kód `sleep()` na každou hranu, která má časové omezení, ale s tím se velká část kódu na výstupu bude skládat z volání výše zmíněné funkce. Proto byla vybrána jiná možnost, kde systém výtahu, konkrétně model `Controller`, byl doplněn

o testovací kód. Bylo rozhodnuto doplnit systém tak, aby vygenerovaný testovací případ poslal požadavky při stisknutí tlačítka, a následně čekal na jejich zpracování. Tímto by mělo být zajištěno stejné pořadí zpracování signálů stisknutí tlačítka v obou navržených systémech. Příklad vygenerovaného kusu kódu, který odpovídá pořadí stisknutí tlačítka, vypadá následně:

```
Queue1 = Add_to_queue(Queue1, 0);
Expected_button_unpressed(Queue1, 0);
Queue1 = Add_to_queue(Queue1, 1);
Expected_button_unpressed(Queue1, 1);
Queue1 = Add_to_queue(Queue1, 2);
Queue1 = Add_to_queue(Queue1, 3);
Expected_button_unpressed(Queue1, 2);
Expected_button_unpressed(Queue1, 3);
```

Kapitola 5

Zhodnocení dosažených výsledků

Cílem této kapitoly je popsat jednotlivé testovací případy, které byly vygenerované a pokrytí systému, které bylo dosaženo po generování těchto případů. Bylo zvoleno použití dvou pokrytí, které budeme sledovat během testování, a to pokrytí hran a stavu vytvořeného systému v UPPAAL. Zároveň ze statistik, které generuje nástroj, je převzato celkové pokrytí systému. Pro určení, zda testovací případ prošel (angl. pass) nebo selhal (angl. fail), je generovaný testovací soubor spuštěn na testovacím systému. Před začátkem prací v generátoru, byly všechny implementované modely ověřeny ve Verifikátoru pomocí dotazů `A[] not deadlock` pro kontrolu, zda neexistuje žádný deadlock.

Je nutné si uvědomit, že nástroj UPPAAL generuje tzv. abstraktní testovací případ, který je platný pro implementovaný systém ve formě časovaného automatů. Po ukládání vygenerované cesty dostáváme na výstupu již konkrétní testovací případ. Pokud mluvíme o pokrytí systému, tak se jedná o pokrytí časovaného automatu v nástroji UPPAAL. Konkrétní testovací případ vynechává informace o dosažených místech v testovaném programu, protože je vytvořen tak, že umožňuje provádět testování metodou černé skříňky, kde vnitřní struktura testovaného systému není známa.

5.1 Vypínač světla

Pro generování testovacích případů byly zvolené 4 scénáře, které jsou zapsané ve formě dotazu `E<>`. Zároveň bylo rozhodnuto přidat mutaci do testovacího systému pro ukázku chování vygenerovaných testovacích souborů po jejich spuštění (soubor `LightM`).

Experiment č.1

První scénář je zaměřen na ověření, zda implementovány vypínač světla umožňuje rozsvítit žárovku do jasna. Tohle lze zjistit pomocí dotazu `E<> L.BRIGHT`, který je použit pro generování testovacího kódu. Na začátku generování bylo otestováno, zda změna vlastností generátoru mění vygenerovanou cestu. V našem případě, kde generujeme cestu pomocí dotazů s ohledem na velikost testovaného systému, byly ve výsledku získány stejné cesty. Proto bylo rozhodnuto neměnit tyto vlastnosti, tz. hledání cesty probíhá na šířku systému a snaží se najít nejkratší cestu k ověření vlastností, která je uvedena pomocí dotazů. Statistika vygenerované cesty je zobrazena v tabulce 5.1. Bylo dosaženo 100% pokrytí stavu, ale jenom 60% pokrytí hran. Důvodem je, že pro dosažení stavu `BRIGHT` stačí provést jenom tři přechody. Pro vylepšení pokrytí je použito generování testovacích případů na zadanou hloubku. Hloubka je nastavena na 10, je zapnuto náhodné hledání a hledání náhodné cesty.

Podle zmíněných vlastností, generátor vytvořil dva testovací případy, díky kterým bylo dosaženo 100% pokrytí stavu a hran. Všechny tři testovacích případy byly uložené a následně vyzkoušeny na testovacím modelu. Spuštění jednotlivých testů proběhlo úspěšně.

Tabulka 5.1: Experiment č.1

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	4/4 (100%)	3/5 (60%)	77%	2	Pass
2	3/4 (75%)	3/5 (60%)	66%	10	Pass
3	4/4 (100%)	5/5 (100%)	100%	10	Pass
Celkem	4/4 (100%)	5/5 (100%)	100%	22	

Experiment č.2

Pro další experiment byl zvolen scénář, který je možné popsat dotazem `E<> (L.ON and L.x>5)`. Dotaz představuje situaci, že vypínač světla po prvním stisknutí nebyl stisknut ještě jednou, a to v průběhu 5 časových jednotek. Výsledek generované cesty je znázorněn v tabulce 5.2. Pro ověření tohoto dotazu stačilo vystřelit jednou, tím bylo provedeno 2 přechody, a poté systém čeká na hodiny. Tento krát pro vylepšení pokrytí byly změněné vlastnosti generování cesty na zadanou hloubku, a to změnou hledání na šířku a vyhledání nejrychlejší cesty. Na výstupu generátoru bylo přidané dvě cesty, ale kvalita cesty odpovídá 88%. Při prohledávání celkového pokrytí bylo zjištěno, že generátor nepokryje jednu hranu `L.ON -> OFF`. Je možné tedy spustit generátor metodou single-step pro provádění přechodů nepokryté hrany. Ve výsledku bylo dosažené 100% pokrytí po 23 krocích.

Tabulka 5.2: Experiment č.2

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	4/4 (100%)	4/5 (80%)	88%	5	Pass
2	3/4 (75%)	3/5 (60%)	66%	2	Pass
Celkem	4/4 (100%)	5/5 (100%)	100%	7	

Experiment č.3

Pro následující scénář byla použita proměnná, která ukládá počet stisknutí tlačítka. Dotaz `E<> (B.n == 5 and L.BRIGHT)` ověřuje, zda po pěti stisknutích tlačítka se světlo rozsvítí do jasna. Generátor nástrojů UPPAAL vyprodukoval cestu, která pokrývá všechny stavy a skoro všechny hrany kromě `L.ON->OFF`. Zde je možné vynechat náhodné generování cesty na zadanou hloubku a již teď vygenerovat cestu pro pokrytí hrany, která zůstala.

Tabulka 5.3: Experiment č.3

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	4/4 (100%)	3/5 (60%)	77%	2	Pass
2	3/4 (75%)	3/5 (60%)	66%	10	Pass
3	4/4 (100%)	5/5 (100%)	100%	10	Pass
Celkem	4/4 (100%)	5/5 (100%)	100%	22	

Experiment č.4

Poslední experiment, který používá pro generaci testovacích případů dotaz, je ověření, zda je systém schopen za méně než 10 časových jednotek přijat více než 10 stisknutí tlačítka. Nástroj UPPAAL vygeneroval cestu se 100% hran a stavu, a to pomocí dotazu `E<> (y <10 and B.n > 10)`.

Tabulka 5.4: Experiment č.4

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	4/4 (100%)	5/5 (100%)	100%	11	Pass
Celkem	4/4 (100%)	5/5 (100%)	100%	11	

Experiment č.5

Poslední experiment je možné provést pomocí jakéhokoliv z vygenerovaných testovacích kódů nebo je možné vygenerovat nový testovací soubor. Pro testování modelu, které obsahuje mutaci, je potřeba využít skript *testM.sh*, který bude používat pro testování soubor s upravenou třídou. Při spuštění vygenerovaného testovacího případu by se měla zobrazit chybová hláška. V našem případě nastala chyba:

```
testcases/testcase-000.java
Exception in thread "main" java.lang.AssertionError
        at app.Light.isOff(LightM.java:29)
        at app.Test.main(Test.java:9)
```

5.2 2-bitová násobička

Experiment slouží pro ověření, zda implementovaný model v nástrojích UPPAAL a moduly napsané v jazyce Verilog pracují správně a na výstupu vyprodukují správně data podle pravdivostní tabulky.

Experiment č.1

Prvni experiment generuje testovací případ pro model přesně násobičky. Pro vstup byly zvoleny dvojice `x1x0 = 01` a `y1y0 = 11` nebo `x1x0 = 11` a `y1y0 = 01`. Pokud se podíváme na pravdivostní tabulku, na výstupu by měl být výsledek `z3z2z1z0 = 0011`. Generátor získá na vstup dotaz `E<> (bits[4] == 1 && bits[5] == 1 && bits[6] == 0 && bits[7] == 0)`, který ověřuje, zda existuje cesta, že na výstupních bitech přesné násobičky se bude nacházet potřebný výsledek. Po vygenerování cesty je nutné se podívat na vygenerovanou cestu v simulátoru a ověřit, zda na vstupu byly generovány bity podle pravdivostní tabulky. Nástroj UPPAAL vygeneroval cestu, která se skládá z 172 kroků a pokrývá 88 % systému (z nich je pokryto 96 % stavu a 82 % hran). Na vstupu násobičky byly `x1x0 = 11` a `y1y0 = 01`, což odpovídá pravdivostní tabulce. Poslední nepokrytý stav je stav, do kterého je potřeba vygenerovat všechny možné kombinace vstupních dat systému. Teoreticky je možné využít metodu depth search. Z ohledem na to, že systém vyprodukoval první cestu za 172 kroku a tím pokryl 8 kombinací možných vstupních bitů, s velkou pravděpodobností při pokusu o hledání na hloubku 360 by bylo možné pokrýt cestu, která nám zůstala. Ještě

jedna možnost se zaměřit na generování cesty pomocí metody *single step*. Proto bylo rozhodnuto, zaprvé zkusit vygenerovat pro lepší pokrytí cestu pomocí prohledávání hloubky a zatím vygenerovat jednotlivé cesty pro každou nepokrytou hranu. Při pokusu o generaci cesty na hloubku 360, nástroj ukázal zajímavé chování: na místo generace cesty byla vypsána hláška `Not verified... MAYBE_OK`. Při pokusu o zmenšení čísla hloubky, se hláška stále zobrazovala, dokud číslo nebylo zmenšené na 340. Ve výsledku byly získány dvě nové cesty a podařilo se zvýšit pokrytí o 4 %. Dále byly generovány cesty pomocí metody single step, které také nebyly schopny pokrýt chybějící hrany.

Při podrobnějším prohledávání statistik bylo zjištěno, že dosud nepokryté hrany patří k modelu hradel, které jsou používány u systému přibližné násobičky. Konkrétně se jedná o hrany, které obsahuje signál `cin0?` nebo `cin1?`. Jediný model, u kterého byly pokryty oba přechody je model hradla, který implementuje hradlo `OR` (na obrázky 4.2 je označeno číslem 2) a pro svoje provádění potřebuje výstupy 1 a 3 hradla, proto systém může generovat cestu tak, že každá z těchto hran je schopna se vystřelit. Signál `change[idx]!` je odeslán modelem `syncPrimary`, a generuje číslo `idx`, které se začíná od `0` a postupně roste. V deklaraci hradel prvního řádu je uděláno tak, aby každé z nich bylo synchronizované s ostatními, což dovoluje provést přechod zároveň u dvou ze čtyř hradel. Daná synchronizace se probíhá pomocí čísla, které je získané od modelu `syncPrimaru`, tz.pomocí `idx`. Právě kvůli tomu nastává situace, že se vždycky vystřelí hrany, u kterých přechody jsou označené 0 a 1, a přechody, které jsou synchronizovány pomocí čísel 2 a 3 se nikdy neprovedou. Pro řešení daného problému je nutné změnit model hradel nebo jejich vlastnosti pro dosažení 100 % pokrytí. Následně se budeme snažit vygenerovat maximální dosažitelné pokrytí, bez zahrnutí těchto hran.

Tabulka 5.5: Experiment č.1

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	27/28 (96%)	29/35 (82%)	88%	172	Pass
2	28/28 (100%)	30/35 (85%)	92%	340	Pass
3	28/28 (100%)	30/35 (85%)	92%	340	Pass
Celkem	28/28 (100%)	31/35 (88%)	93%	852	
4	28/28 (100%)	21/35 (60%)	74%	19	Pass
5	28/28 (100%)	21/35 (60%)	74%	19	Pass
6	28/28 (100%)	21/35 (60%)	74%	19	Pass
7	28/28 (100%)	21/35 (60%)	74%	19	Pass
Celkem	28/28 (100%)	31/35 (88%)	93%	928	

Experiment č.2

Nasledne je testována přibližná násobička se stejnými vstupními daty. Na výstupu je očekáván výsledek `011`, proto je zde používán dotaz pro generaci cesty `E<>(bits[8] == 0 && bits[9] == 1 && bits[10] == 1)`. Jelikož se v systému nejdříve generují výstupy z přesně násobičky, a poté z přibližné násobičky, systém potřeboval 229 kroků pro dosažení potřebného výstupu. Kvalita pokrytí hran a stavu je stejná s výsledkem předchozího experimentu. Tentokrát zkusíme již od začátku vygenerovat pro vylepšení pokrytí cestu pomocí metodu *single step*. Byly vygenerováno 5 nových cest, pomocí kterých bylo dosaženo 93 % celkového pokrytí systému za 629 kroků.

Tabulka 5.6: Experiment č.2

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	27/28 (96%)	29/35 (82%)	88%	229	Pass
2	28/28 (100%)	30/35 (85%)	92%	324	Pass
3	26/28 (92%)	21/35 (60%)	74%	19	Pass
4	26/28 (92%)	21/35 (60%)	74%	19	Pass
5	26/28 (92)	21/35 (60%)	74%	19	Pass
6	26/28 (100%)	21/35 (60%)	74%	19	Pass
Celkem	28/28 (100%)	31/35 (88%)	93%	629	

Experiment č.3

Třetí experiment pracuje již s dvěmi násobičkami a úkolem je pokrýt všechny možné kombinace vstupu násobiček. To je možné zajistit pomocí dotazu `E<> allCovered`. Na výstupu generátoru je testovací případ, který se skládá z 324 kroků a má 100 % pokrytí stavu a 30 % pokrytí hran. Při porovnání s dosaženými výsledky předtím je vidět, že nebyl pokryt jeden přechod, proto je možné vylepšit pokrytí a v tomto případě stačí použít metodu *single step*.

Tabulka 5.7: Experiment č.3

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	28/28 (100%)	30/35 (85%)	92%	324	Pass
2	26/28 (92%)	21/35 (60%)	74%	19	Pass
3	26/28 (92%)	21/35 (60%)	74%	19	Pass
4	26/28 (92%)	21/35 (60%)	74%	19	Pass
5	26/28 (92%)	21/35 (60%)	74%	19	Pass
Celkem	28/28 (100%)	31/35 (88%)	93%	400	

Experiment č.4

Poslední experiment spočívá v zavedení náhodného generování vstupních bitů. Během verifikace a generování testovacích skriptů UPPAAL nebyl schopen splnit naše potřeby. Ve všech možných dotazech nástroj hlásil chybu *IOException: Server connection lost*. Generace testů byla vyzkoušena na všech možných platformách a na různých verzích nástrojů UPPAAL, které umožňují generovat testovací případy pomocí dotazů (včetně poslední, která byla vydána na konci dubna 2021), ale výsledek byl stejný. Při podrobnějším sledování chování bylo zjištěno, že nástroj může generovat jenom dva různé vstupy. Při pokusu o generaci třetího vstupu proces generování trvá více než 30 minut, a poté program skončí svůj běh.

5.3 Výtah

V případě testování systému výtahu, je kladen velký důraz na generaci vyvolání jednotlivých stisknutí tlačítka a jejich zpracování. Pro generování testovacích případů byly vytvořeny tři dotazy a zároveň je systém rozšířen o model `Observer`, který je navržen tak, aby byla generována posloupnost náhodného stisknutí tlačítka pro ověření správnosti pohybu výtahu. Ve většině situací generátor produkuje jenom posloupnost pohybů směrem nahoru, jelikož výtah se při inicializaci systému nachází na 0.patře a je schopen zachytit všechny patra

po cestě na 3.patro. Poslední experiment sleduje pokrytí systému a generace testovacích případů různé hloubky, v kombinaci se všemi možnými vlastnostmi této metody.

Experiment č.1

První testovací případ ověřuje, zda existuje situace, kde bylo stisknuté tlačítko na 2. patře a zpracování tohoto signálu (včetně otevírání dveří na patře) trvalo míň než 5 časových jednotek. Pro ověření je možné využít dotaz ve formě `E<> Floor(2).Button_deactive and Floor(2).flt <5` .Z důvodu velikosti systému bylo pro lepší pokrytí zvoleno nastavení náhodného vyhledávání cesty. Nástroj vygeneroval cestu, která pokrývá všechny stavy, ale pokrytí hran odpovídá 96%. Při podrobnějším prohledávání statistiky vygenerované cesty bylo zjištěno, že nebyl pokryt jenom jeden stav. Teoreticky je možné vygenerovat cestu s žádanou hloubkou, ale je dost obtížné vybrat potřebnou hloubku pro pokrytí zbývající hrany. Proto je lepší použít poslední možnost a vygenerovat cestu pro konkrétní nepokrytou hranu.

Tabulka 5.8: Experiment č.1

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	27/27 (100%)	32/33 (96%)	98%	56	Pass
2	13/27 (48%)	10/33 (30%)	38%	6	Pass
Celkem	27/27 (100%)	33/33 (100%)	100%	62	

Experiment č.2

Stejně jako v prvním experimentu, zde je používána proměnná pro sledování doby čekání na zpracování požadavků. Tentokrát je dotaz zaměřen na výskyt situace, kde doba čekání na příjezd výtahu trvala dýl, než 80 časových jednotek. Pro daný účel je použít dotaz `E<> Floor(3).Button_active and Floor(3).flt > 80`. Ve výsledku byla vygenerována cesta, jejíž celková kvalita je 81% (podrobnější statistika je zobrazena v tabulce 5.9). Pro vylepšení pokrytí je použita metoda *depth-search* s žádanou hloubkou 30 a náhodným prohledáváním. Nástroj vygeneroval dvě nové cesty pro průchod systémem, ale celková kvalita se změnila jenom o 7 %, tzn., že je nutné použít větší hloubku pro generování cesty nebo nastavit jiné vlastností generátoru. Nejdříve je zvýšeno číslo hloubky na 40. Tím na výstupu byly vygenerovány dvě nové cesty, pomocí kterých bylo získáno 98

Tabulka 5.9: Experiment č.2

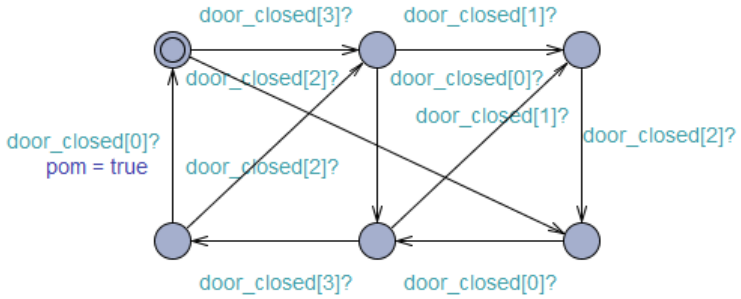
	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	23/27 (85%)	26/33 (78%)	81%	48	Pass
2	25/27 (92%)	26/33 (78%)	85%	30	Pass
3	19/27 (70%)	17/33 (51%)	60%	30	Pass
Celkem	25/27 (92%)	28/33 (100%)	88%	108	
4	25/27 (92%)	27/33 (81%)	86%	40	Pass
5	23/27 (85%)	26/33 (78%)	81%	40	Pass
Celkem	27/27 (100%)	32/33 (96%)	98%	188	
6	13/27 (48)	10/33 (30%)	38%	6	Pass
Celkem	27/27 (100%)	33/33 (100%)	100%	194	

Experiment č.3

Cílem třetího experimentu je ověřit, zda je výtah schopný zastavit na každém patře budovy alespoň jednou. Pro tento účel byla zavedena globální proměnná typu bitového pole `list_check[floor_num]`, do které se ukládá `true` v případě ukončení simulace provozu dveří. Dotaz, pomocí kterého bude generována cesta vypadá následně: `E<> (forall(i: id_t) list_check[i] == true)`. Na výstup byla získána cesta, při kterém všechny tlačítka byla stisknuta již před pohybem výtahu, proto ve výsledku nebyly pokryty hrany, které odpovídají procesu sestupu výtahů a situací, kde výtah dorazil na patro, na kterém není stisknuté tlačítko. Pro vylepšení pokrytí je zde vhodné použít metodu single step, čímž se výrazně sníží celkový počet kroků pro plné pokrytí systémů. Nástroj vygeneroval dvě cesty o 6 a 13 krocích. Celkový výsledek je ukázán v tabulce 5.10.

Tabulka 5.10: Experiment č.3

	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	27/27 (100%)	30/33 (90%)	95%	35	Pass
2	13/27 (48%)	10/33 (30%)	38%	6	Pass
3	19/27 (70%)	20/33 (60%)	65%	13	Pass
Celkem	27/27 (100%)	33/33 (100%)	100%	54	



Obrázek 5.1: Šablona *Observer*.

Experiment č.4

Pro práci s posledním dotazem je potřeba v deklaraci systému přidat proces pro model `Observer`. Model `Observer` je implementován tak, aby na výstupu byla vygenerovaná náhodná cesta, která zahrnuje pohyb výtahu nahoru a dolů. Model obsahuje lokální proměnnou pom, která se používá v dotazu `E<> (Observer.pom)` a označí konec generace cesty. Kvůli tomu, že systém je rozšířen o nový model, se zvýšil počet hran a stavů, které generátor musí projít. Nástroj vygeneroval cestu, která se skládá z 60 kroků a pokrývá jenom 78%, jelikož systém prošel jeden průchod modelem `Observer`, co stačí pro splnění podmínky dotazu. Následně je vhodné vygenerovat náhodnou cestu pomocí metody *depth-serch*. Hloubka prohledávání je změněna na 50. Stejně jako u předchozích experimentů jsou vygenerovány dvě nové cesty průchodu systému s celkovým pokrytím 89%. Celkem bylo pokryto 31 z 33 stavů a 37 z 43 hran systému. Následně bylo zvoleno změnit vlastnost prohledávání z náhodného na hloubku, a poté na šířku, čímž nebylo dosáhnuto zvýšení celkového pokrytí

systému. To znamená, že vybraná hloubka nestačí pro naše účely a je potřeba ji zvýšit. Postupně vybraná hloubka byla změněna na 60, 70 a 80. Zajímavé je, že nebyly pokryty žádné nové hrany či stavy. Důvodem může být špatně navržený model `Observer`, který se skládá ze stisknutí tlačítka na konkrétních patrach a nezahrnuje ostatní, např. inicializační stav vystřelí jenom v situaci, pokud bude stisknuté buď tlačítko na 2. patře nebo na 3. patře a v případě stisknutí tlačítek na 0. a 1. patře nic se nestane. Následně bylo rozhodnuto vygenerovat cesty pro všechny nepokryté části systému, a to pomocí metody *single-step*, která na výstup přidala 4 nové cesty. Ve výsledku byly vygenerováno 12 testovacích souborů.

Tabulka 5.11: Experiment č.4

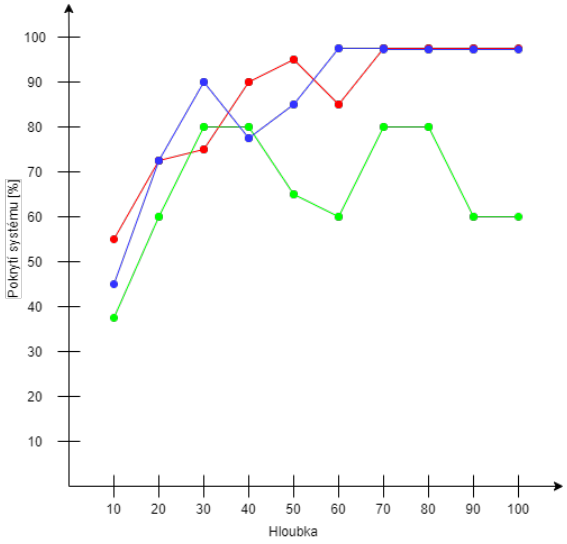
	Pokrytí stavu	Pokrytí hran	Kvalita cesty	Počet kroků	Spuštění na SUT
1	28/33 (84%)	32/43 (74%)	78%	60	Pass
2	27/33 (81%)	28/43 (65%)	72%	50	Pass
3	29/33 (87%)	31/43 (72%)	78%	50	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	160	
4	27/33 (81%)	30/43 (69%)	75%	50	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	210	
5	20/33 (60%)	17/43 (39%)	48%	50	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	260	
6	29/33 (87%)	31/43 (72%)	78%	60	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	320	
7	27/33 (81%)	28/43 (65%)	72%	70	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	390	
8	27/33 (81%)	28/43 (65%)	72%	80	Pass
Celkem	31/33 (93%)	37/43 (86%)	89%	470	
9	28/33 (84%)	31/43 (72%)	77%	35	Pass
10	29/33 (87%)	32/43 (74%)	80%	52	Pass
11	24/33 (72%)	26/43 (60%)	65%	34	Pass
12	28/33 (84%)	31/43 (72%)	77%	39	Pass
Celkem	33/33 (100%)	43/43 (100%)	100%	630	

Experiment č.5

Po získání výsledků z předchozích experimentů bylo rozhodnuto otestovat možnosti generátoru testovacích případů na zadanou hloubku, jelikož při testování různých systémů je obtížné vybrat přesnou hloubku prohledávání a vlastnosti generátorů. Pro generování cesty byl použit systém výtahů, bez rozšíření o model `Observer`. Experiment je zaměřen na sledování celkové kvality vygenerované cesty při různých parametrech, a to hloubka (začíná od 10 a postupně se zvyšuje o 10, tedy 10, 20, 30 atd.) a vlastnosti generátoru, konkrétně způsob hledání a kritéria vygenerované cesty.

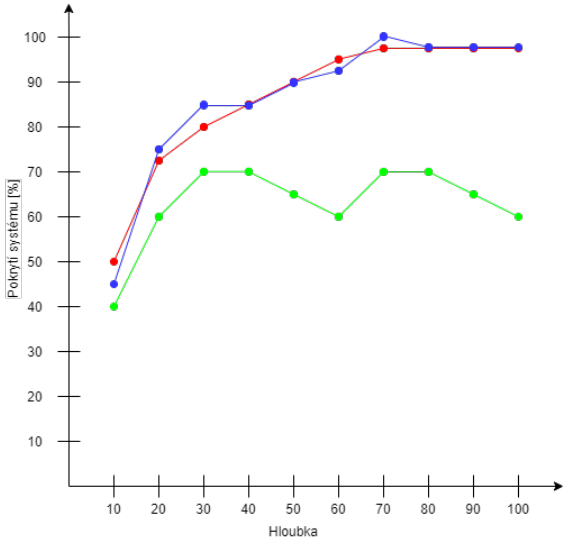
- Náhodné hledání generuje náhodné nedeterministické možnosti průchodu systémem. Na obrázku 5.2 je ukázané pokrytí všech součástek systémů podle nastavené hloubky. Zelenou barvou je označena nejrychlejší cesta. Je vidět, že kvalita cesty je ve formě křivky. Nástroj UPPAAL umožňuje podívat se na vygenerovanou cestu přes *Simulátor*. Při podrobnějším prohledávání byla zjištěna tendence, že všechny vygenerované cesty zahrnovaly buď nekonečný cyklus stisknutí tlačítka na 0 nebo 1. patře, a to tak,

že výtah získal signál od každého patra, ale předtím než pokračoval v pohybu o patro výš, získal signál z aktuálního patra. Červená barva slouží pro zobrazení chování při generování náhodné cesty. Při nejmenším počtu kroků tento způsob má největší procento pokrytí systému při porovnání s ostatníma cestami. Po zvýšení čísla hloubky procento pokrytí rostlo, ale najednou začalo klesat. V simulátoru je vidět, že generovaná cesta se znovu pozastavila na jednom patře a vytvořila smyčku volání signálu na tomto patře, ale tentokrát výtah byl schopen se dostat na poslední patro budovy. Maximální pokrytí, kterého se podařilo dosáhnout, je 98 %. Celkem systém obsahuje 76 součástek, z nich 33 stavu a 43 hran. Je vidět, že pro nejlepší pokrytí stačí použít hloubku 70. Modrou barvou je označena nejkratší cesta, která je generována stejným způsobem jako náhodná cesta, zatímco se snažila pokrýt systém co nejmíň kroků. Na rozdíl od náhodné cesty, systém byl pokryt na 98 % již za 60 kroků.



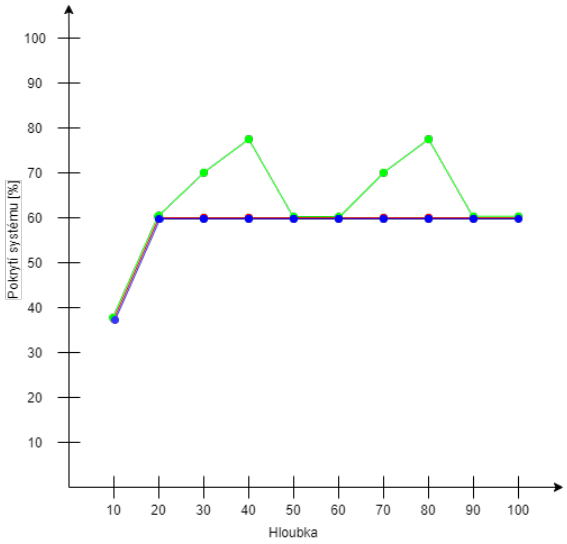
Obrázek 5.2: Kvalita cest při náhodném hledání. Červená barva - první nalezená cesta, zelená - nejrychlejší cest, modrá - nejkratší cesta.

- Hledání na hloubku slouží pro hledání cest v systému, které obsahuje dlouhou posloupnost přechodů. Je možné říct, že daný způsob generování pokrývá mnohem větší počet prvků systému. Chování při hledání nejrychlejší cesty je stále zobrazeno ve formě křivky, ale celkový procent pokrytí je horší, než při náhodném hledání cesty. Maximálně se podařilo dosáhnout jenom 71 % pokrytí systému. Na rozdíl od nejrychlejší cesty, generování nejkratší a náhodné cesty postupně roste. V případě nejkratší cesty se při hledání cesty na hloubku 70 podařilo získat 100 % pokrytí.



Obrázek 5.3: Kvalita cest při hledání na hloubku. Červená barva - první nalezená cesta, zelená - nejrychlejší cest, modrá - nejkratší cesta.

- Hledání na šířku se hodí pro generování cest v systému, které se skládá z několika paralelně běžících procesů. Na obrázky 5.4 je vidět, že náhodná cesta a nejkratší cesta generuje stejný průchod systémem a maximálně pokrývá 60 % celkového systému. Nejrychlejší cesta se chová stejným způsobem jako v předchozích metodách hledání cesty. Z pohledu pokrytí hledání na šířku je nejhorší možnost generování cesty pro náš systém. Důvodem je, že systém neobsahuje velký počet paralelně běžících procesů. Proto by bylo vhodné vyzkoušet a podívat se na chování daného hledání za použitím jiného systému.



Obrázek 5.4: Kvalita cest při hledání na šířku. Červená barva - první nalezená cesta, zelená - nejrychlejší cest, modrá - nejkratší cesta.

Kapitola 6

Závěr

Cílem této bakalářské práce bylo vytvořit přehled o problematice testování, různých druhů přístupu k danému procesu a ukázat existující nástroje, které obsahují generátor testovacích případů. Pro ukázku byl vybrán nástroj UPPAAL, způsoby práce s ním a jeho schopnosti, které jsou výše popsány. Pomocí nástrojů byly vytvořeny tři systémy z různých oblastí a implementovány testovací případy pro ověření jeho schopností. Zároveň s tím byly vytvořeny stejné systémy v různých programovacích jazycích pro ukázku, že vybraný nástroj umožňuje pracovat s jakýmkoliv testovacím systémem. Jediné omezení nastává, pokud je testovaný systém napsán v jazyce, který potřebuje pro správné fungování odsazování jednotlivých částí (např. Python), jelikož nástroj nepodporuje odsazování různých částí kódu, kterými je systém doplněn.

Během implementací jednotlivých modelů v UPPAAL byly odhaleny omezení tohoto nástroje. Hlavní z nich je neschopnost nástroje předat testovacímu kódu různé parametry, např. pole, jeho prvky, náhodně vygenerované číslo, které bylo generováno na hraně pomocí Select a hodiny (jelikož jsou představeny pomocí omezení, ne konkrétního čísla). V případě generování testovacích případů na základě dotazu, se málokdy dá dosáhnout 100% pokrytí systému, proto pro zvýšení kvality je možné vygenerovat nové cesty. Ve výsledku to vede ke generování několika spustitelných testovacích případů a při špatně zvolené metodě, může nastat situace, že na výstupu získáme 5-10 souborů.

Ačkoli nástroj obsahuje výše zmíněné omezení, UPPAAL umožňuje snadno vytvořit model testovaného systému a vygenerovat příslušné testovací případy, pomocí kterých je možné otestovat funkčnost konečného programu. I když nástroj neobsahuje prostředky pro sledování stráveného času při generaci jednotlivých cest, je vidět, že proces hledání cesty trvá jenom pár vteřin. V případě velkého systému nebo generování cesty, která obsahuje přes 200 kroků, doba čekání trvá trochu déle, ale nikdy čas nepřekračuje 10 sekund. Zároveň při použití dotazu pro generování testovacích případů lze snadno ověřit dosažitelnost různých částí systému.

Ve výsledku byly vytvořeny modely z různých oblastí, zároveň byly vymyšleny a implementovány testovací scénáře, které byly použity pro generování testovacích případů. V budoucnosti lze modelování systému rozšířit, např. v systému výtahu je možné přidat šablony uživatele, modelovat situaci poruchy, prioritu páteře atd. Zároveň v této práci byla představena ukázka jednoduché implementace mutačního testování, které lze dál rozvíjet, např. pomocí zavedení mutací do modelu, který je implementován v nástroji UPPAAL, a následně pozorovat chování testovaného systému.

Literatura

[1] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B. et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*. 1. vyd. 2013, sv. 86, č. 8, s. 1978–2001. DOI: 10.1016/j.jss.2013.02.061. ISSN 0164-1212.

[2] Artho, C., Seidl, M., Gros, Q., Choi, E.-H., Kitamura, T. et al. Model-Based Testing of Stateful APIs with Modbat. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, s. 858–863. DOI: 10.1109/ASE.2015.95.

[3] Behrmann, G. *UPPAAL CORA* [online]. [cit. 2021-05-09]. Dostupné z: http://people.cs.aau.dk/~adavid/cora/.

[4] Behrmann, G., David, A. a Larsen, K. G. *A Tutorial on Uppaal*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. 200–236 s. ISBN 978-3-540-30080-9.

[5] Bird, D. L. a Munoz, C. U. Automatic Generation of Random Self-Checking Test Cases. *IBM Syst. J.* 1983, sv. 22, s. 229–245.

[6] Bourque, P., Fairley, R. E. a Society, I. C. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Washington, DC, USA: IEEE Computer Society Press, 2014. ISBN 0769551661.

[7] Cadar, C., Dunbar, D. a Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2008, s. 209–224. OSDI'08.

[8] Computer Science, K. R. I. of Technology School of a Communication. *Modbat* [online]. [cit. 2021-05-09]. Dostupné z: https://people.kth.se/~artho/modbat/.

[9] Conformiq. *Conformiq Products* [online]. [cit. 2021-05-09]. Dostupné z: https://www.conformiq.com/products/.

[10] Cseppentő, L. a Micskei, Z. Evaluating code-based test input generator tools. *Software Testing, Verification and Reliability*. Únor 2017, sv. 27. DOI: 10.1002/stvr.1627.

[11] David, A. *Statistical Model-Checker* [online]. [cit. 2021-05-09]. Dostupné z: http://people.cs.aau.dk/~adavid/smc/.

[12] David, A. *UPPAAL TIGA* [online]. [cit. 2021-05-09]. Dostupné z: http://people.cs.aau.dk/~adavid/tiga/.

[13] Ehmer, M. a Khan, F. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications*. Červen 2012, sv. 3. DOI: 10.14569/IJACSA.2012.030603.

[14] Fraser, G. *EvoSuite / Automatic Test Suite Generation for Java* [online]. [cit. 2021-05-09]. Dostupné z: https://www.evosuite.org/.

[15] Fraser, G. a Arcuri, A. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*. 2013, sv. 39, č. 2, s. 276–291. DOI: 10.1109/TSE.2012.14.

[16] Fraser, G., Staats, M., McMinn, P., Arcuri, A. a Padberg, F. Does automated white-box test generation really help software testers? *2013 International Symposium on Software Testing and Analysis, ISSTA 2013 - Proceedings*. Červenec 2013, s. 291–301. DOI: 10.1145/2483760.2483774.

[17] Galler, S. a Aichernig, B. Survey on test data generation tools - An evaluation of white- and gray-box testing tools for C, C++, Eiffel, and Java. *International journal on software tools for technology transfer*. Springer Verlag. 2014, sv. 16, č. 6, s. 753–773. ISSN 1433-2779.

[18] Graf, S., Ober, I. a Ober, I. A real-time profile for UML. *STTT*. Duben 2006, sv. 8, s. 113–127. DOI: 10.1007/s10009-005-0213-x.

[19] Hailpern, B. a Santhanam, P. Software debugging, testing, and verification. *IBM Systems Journal*. Prosinec 2001, sv. 41, s. 4–12. DOI: 10.1147/sj.411.0004.

[20] Harel, D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*. 1987, sv. 8, č. 3, s. 231–274. DOI: 10.1016/0167-6423(87)90035-9. ISSN 0167-6423.

[21] Hessel, A. et al. *Testing Real-Time Systems Using UPPAAL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. 77–117 s. ISBN 978-3-540-78917-8.

[22] Hessel, A., Larsen, K., Nielsen, B., Pettersson, P. a Skou, A. Time-Optimal Real-Time Test Case Generation Using Uppaal. In:. Leden 2003, s. 114–130.

[23] https://statecharts.dev/. *How to use statecharts - Statecharts* [online]. [cit. 2021-05-09]. Dostupné z: https://statecharts.dev/how-to-use-statecharts.html.

[24] Huima, A. Implementing Conformiq Qtronic. In: Petrenko, A., Veanes, M., Tretmans, J. a Grieskamp, W., ed. *Testing of Software and Communicating Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 1–12. ISBN 978-3-540-73066-8.

[25] Inc, U. *UP4ALL* [online]. 2012 [cit. 2021-05-09]. Dostupné z: https://www.uppaal.com/.

[26] King, J. C. Symbolic Execution and Program Testing. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. 1976, sv. 19, č. 7, s. 385–394. DOI: 10.1145/360248.360252. ISSN 0001-0782.

[27] Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jöbstl, E. et al. Momut::UML model-based mutation testing for UML. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*. Květen 2015. DOI: 10.1109/ICST.2015.7102627.

[28] Larsen, K., Pettersson, P. a yi, W. Uppaal in a Nutshell. *STTT*. Prosinec 1997, sv. 1, s. 134–152. DOI: 10.1007/s100090050010.

[29] McMinn, P. Search-Based Software Testing: Past, Present and Future. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, s. 153–163. DOI: 10.1109/ICSTW.2011.100.

[30] Micskei, Z. *Code-based test generation* [online]. [cit. 2021-05-09]. Dostupné z: http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html.

[31] Micskei, Z. *Model-based testing (MBT)* [online]. [cit. 2021-05-09]. Dostupné z: http://mit.bme.hu/~micskeiz/pages/mbt.html.

[32] Mikučionis, M. *Uppaal Stratego* [online]. [cit. 2021-05-09]. Dostupné z: https://people.cs.aau.dk/~marius/stratego/.

[33] Mikučionis, M. *UPPAAL TRON* [online]. [cit. 2021-05-09]. Dostupné z: https://people.cs.aau.dk/~marius/tron/.

[34] Morell, L. J. A Theory of Fault-Based Testing. *IEEE Trans. Softw. Eng.* IEEE Press. srpen 1990, sv. 16, č. 8, s. 844–857. DOI: 10.1109/32.57623. ISSN 0098-5589.

[35] Myers, G. J., Sandler, C. a Badgett, T. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN 1118031962.

[36] Pacheco, C., Lahiri, S. K., Ernst, M. D. a Ball, T. Feedback-Directed Random Test Generation. In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, s. 75–84. DOI: 10.1109/ICSE.2007.37.

[37] Parasoft. *Automated Testing ti Deliver Superio Quality Software | Parasoft* [online]. [cit. 2021-05-09]. Dostupné z: https://www.parasoft.com/.

[38] Parasoft. *Integrares Java Testing Tool For App Software Development* [online]. [cit. 2021-05-09]. Dostupné z: https://www.parasoft.com/products/parasoft-jtest/.

[39] PragmaDev. *PragmaDev - Modeling and Testing tools* [online]. [cit. 2021-05-09]. Dostupné z: https://www.pragmadev.com/index.html.

[40] Prasetya, I. T3: Benchmarking at Third Unit Testing Tool Contest. In: *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. 2015, s. 44–47. DOI: 10.1109/SBST.2015.18.

[41] Prasetya, S. W. *Home Wiki Prasetya, S.W.B. (Wishnu) / t3 GitLab* [online]. [cit. 2021-05-09]. Dostupné z: https://git.science.uu.nl/prase101/t3/-/wikis/home.

[42] Ramler, R. a Wolfmaier, K. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. New York, NY,

USA: Association for Computing Machinery, 2006, s. 85–91. AST '06. DOI: 10.1145/1138929.1138946. ISBN 1595934081. Dostupné z: https://doi.org/10.1145/1138929.1138946.

[43] Randoop. *Randoop: Automatic unit test generation for Java* [online]. [cit. 2021-05-09]. Dostupné z: https://randoop.github.io/randoop/.

[44] Sen, K. *Ksen007/janala2: a concolic testing engine for Java* [online]. [cit. 2021-05-09]. Dostupné z: https://github.com/ksen007/janala2.

[45] Singh, R., Singhrova, A. a Bhatia, R. Test Case Generation Tools - A Review. *International Journal of Electronics Engineering*. 2018, sv. 10, s. 586–596. ISSN 0973-7383.

[46] Software, S. *Gherkin Syntax - Cucumber Documentation* [online]. [cit. 2021-05-09]. Dostupné z: https://cucumber.io/docs/gherkin/.

[47] Strnadel, J. Statistical Model Checking of Approximate Circuits: Challenges and Opportunities. In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020, s. 1574–1577. DOI: 10.23919/DATE48585.2020.9116207.

[48] Tanno, H., Zhang, X., Hoshino, T. a Sen, K. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015, sv. 2, s. 717–720. DOI: 10.1109/ICSE.2015.231.

[49] team, G. *GraphWalker* [online]. [cit. 2021-05-09]. Dostupné z: https://graphwalker.github.io/.

[50] team, T. K. *KLEE* [online]. [cit. 2021-05-09]. Dostupné z: http://klee.github.io/.

[51] Technology, A. A. I. of. *MoMuT - Model-based Mutation Testing* [online]. [cit. 2021-05-09]. Dostupné z: https://momut.org/.

[52] Uppaal. *About | UPPAAL* [online]. [cit. 2021-05-09]. Dostupné z: https://uppaal.org/.

[53] Utting, M. a Legeard, B. *Practical Model-Based Testing: A Tools Approach*. 2007. ISBN 978-0-12-372501-1.

[54] Wang, R., Li, Y., Xie, H., Xu, Y. a Lui, J. C. S. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, červenec 2020, s. 559–571. ISBN 978-1-939133-14-4. Dostupné z: https://www.usenix.org/conference/atc20/presentation/wang-rui.