VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

OVĚŘOVÁNÍ TEMPORÁLNÍCH VLASTNOSTÍ KONEČNÝCH BĚHŮ PROGRAMŮ

CHECKING OF TEMPORAL PROPERTIES OF FINITE TRACES OF PROGRAMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PETRA SEČKAŘOVÁ**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Studentka: **Sečkařová Petra, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Ověřování temporálních vlastností konečných běhů programů**
Checking of Temporal Properties of Finite Traces of Programs
Kategorie: Analýza a testování softwaru

Zadání:

1. Nastudujte temporální logiky lineárního budoucího a minulého času (LTL a ptLTL). Nastudujte principy monitorování běhu programů za účelem ověřování vlastností programů specifikované v těchto logikách.
2. Navrhněte systém pro monitorování potenciálně nekončených běhů programů, u nichž jsou známé zdrojové kódy a způsob překladu.
3. Implementujte program pro ověřování temporálních vlastností. Program by měl podporovat programy v jazyce C. Implementujte program jako subsystém platformy Testos.
4. Správnost funkcionality sledování splnitelnosti formulí podpořte automatickými jednotkovými testy.

Literatura:

- Havelund, K.; Roşu, G.; Efficient monitoring of safety properties. In International Journal on Software Tools for Technology Transfer (2004) 6: 158. doi: 10.1007/s10009-003-0117-6

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Temporální vlastnosti programů jsou používány ke specifikaci korektního průběhu jejich vykonávání. Jedním z nejčastějších způsobů formálního popisu těchto vlastností je *lineární temporální logika – LTL*, případně její varianty. Tato práce se zabývá návrhem a implementací nástroje pro automatizované ověřování temporálních vlastností běhů programů specifikovaných pomocí LTL minulého času (past-time LTL). Výsledný program na základě dané specifikace vygeneruje statickou knihovnu, která dokáže spolehlivě ověřit, zda jsou její formule v každém okamžiku běhu kontrolovaného programu splněny, a případné neočekávané nebo nesprávné chování hlásí společně s podrobnou zprávou o okolnostech tohoto chybového stavu, která má napomáhat k nalezení chyby v konkrétním místě kódu.

Abstract

Correct behavior of programs can be defined by their temporal properties. One of the options for formal specification of such properties is *linear temporal logic – LTL*. This master's thesis describes design and implementation of a tool for automatic checking of temporal properties of programs, that are specified using Past-Time LTL formulae. The trace of a given program is analyzed in run-time and any violation of given formulae is reported in details to help to find the code location with a root cause of the bug.

Klíčová slova

verifikace programů za běhu, temporální vlastnosti programů, lineární temporální logika

Keywords

run-time verification, temporal properties of programs, linear temporal logic

Citace

SEČKAŘOVÁ, Petra. *Ověřování temporálních vlastností konečných běhů programů*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Ověřování temporálních vlastností konečných běhů programů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana doktora Aleše Smrčky. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Petra Sečkařová
21. května 2019

Poděkování

Za ochotnou spolupráci, cenné rady a odborný dohled děkuji vedoucímu práce, doktoru Aleši Smrčkovi. Mé poděkováni patři též Tomáši Kratochvílovi z firmy Honeywell za poskytnutí materiálů pro demonstraci reálného použití implementovaného nástroje a spolupráci při jejich zpracovávání.

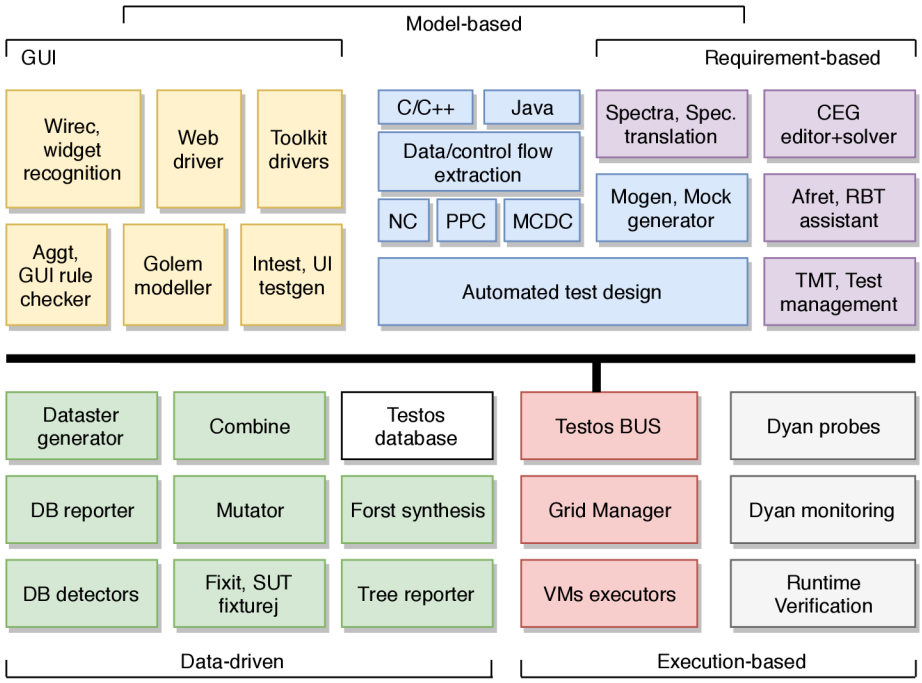
Obsah

Kapitola 1

Úvod

Jedním ze základních úkolů testování programu je kontrola, jestli jeho chování odpovídá dané specifikaci. V té však někdy nestačí pouze definovat očekávané reakce programu na určité vstupy, ale je navíc potřeba popsat i korektní způsob, jakým by měl probíhat jejich výpočet. K takovému popisu se používají temporální vlastnosti běhů programů, jejichž automatizovaným ověřováním se tato práce zabývá.

Tato diplomová práce vznikla jako součást projektu Testos (Test Tool Set) [15], jehož hlavním cílem je vytvoření sady nástrojů podporující automatizované testování softwaru. Nástroje v platformě Testos (viz obrázek 1.1) kombinují různé úrovně testování a lze je řadit do několika kategorií: testování grafického uživatelského rozhraní (GUI), dynamická analýza (Execution-based) nebo testování založené (i) na modelech (Model-based), (ii) na datech (Data-driven), nebo (iii) na požadavcích (Requirement-based).



Obrázek 1.1: Schéma typů zaměření nástrojů platformy Testos a jejich propojení.

Nástroje spadající do kategorie testování založené na požadavcích mají za úkol ulehčit práci testera z hlediska vytváření a správy testů. Ty však ne nutně závisí na konkrétní implementaci, ale zaměřují se na externě specifikované správné chování testovaného subjektu. Nástroje se zaměřují na požadavky definované buďto (i) neformálně (Afret), (ii) programově pomocí BDD scénářů (testBuDDy), nebo (iii) formálně pomocí CFG grafů (CEG solver), nebo temporálních logik.

Cílem této práce je vytvořit podsystémem nástroje Spectra (Specification translation), jehož účelem je transformace formální specifikace softwaru do vhodných forem strukturovaných dat. V případě této práce je zvolenou formou kód v jazyce C/C++, který zkoumá splnění specifikovaných temporálních vlastností pro potenciálně nekonečné běhy kontrolovaného programu. Ve specifikaci programu se k popisu požadovaného chování využívá lineární temporální logika vztažená k historii sledovaného běhu programu – ptLTL. Vytvořený nástroj má být snadno použitelný jak samostatně, tak i společně s ostatními nástroji platformy Testos, a měl by sloužit jako podpora pro hledání a opravu sémantických chyb v implementaci.

Následující kapitola se věnuje základním principům ověřování vlastností běhů programů. Poté budou v kapitole 3 představeny vybrané formální systémy pro popis temporálních vlastností programů a možné způsoby jejich vyhodnocování. Kapitola 4 vysvětluje principy monitorovacího algoritmu vybraného pro implementaci nástroje, jehož návrh je poté v rámci kapitoly 5 podrobně popsán. Kapitola 6 se věnuje implementačním detailům a je následována experimentálním vyhodnocením a validací implementace. Nakonec je v kapitole 8 popsán způsob napojení implementovaného nástroje na platformu Testos.

Kapitola 2

Ověřování vlastností běhu programu

Testování kódu je nesporně velmi důležitou fází vývoje softwaru. Lze k němu přistupovat mnoha různými způsoby, z nichž nejběžnější je založen na definici očekávaných odpovědí systému na dané vstupy a jejich porovnávání se skutečnými odezvami programu. Existují však případy, kdy pouhá kontrola výstupů nestačí, a je potřeba zkoumat i samotný průběh prováděného výpočtu. V takových případech je možné použít monitorování běhu programu, které zkoumá jeho chování v čase výpočtu.

Monitorování programových běhů je podle Bartocciho [4] odlehčenou, komplementární metodou ke klasické vyčerpávající formální verifikaci programů. Tato technika se zaměřuje na jednotlivé běhy daného programu, spíše než na program jako celek, jak je tomu např. u formální verifikace programu proti modelu. To sice limituje spolehlivost výsledků pouze na ověřené běhy programu, avšak ověřování programu pomocí monitorování jeho běhů dokáže poměrně snadno poskytnout cenné a precizní informace o tom, jak se komplexní program chová v čase. Typickým cílem sledování běhu programu je potom právě kontrola správnosti jeho dynamického chování.

Postup, jak ověřovat, že chováni programu za běhu odpovídá nějaké specifikaci, se skládá ze tří kroků [4].

1. Je nutné specifikovat požadované chování systému v podobě formálního popisu jeho vlastností.

2. Na základě této specifikace je vytvořen monitor, který je schopen požadované vlastnosti systému zkoumat a vyhodnocovat jejich splnění.

3. Monitor je napojen na testovaný systém – System Under Test (SUT), který je poté spouštěn a vytvořený monitor ověřuje jeho vlastnosti.

Tento přístup se nijak nevylučuje s ostatními způsoby testování. Naopak je často jednoduše aplikován v rámci ostatních testů, které mohou představovat spouštění zmiňované v bodě 3. Monitorování vlastností běhu programů má dnes široké využití jak v akademickém prostředí, tak v průmyslu, kde je součástí jak testování a verifikace softwaru, tak i krokování kódu už v rámci jeho vývoje, a případně zůstává zabudováno také po nasazení softwaru, pro zajištění jeho bezpečnosti a robustnosti [4].

2.1 Specifikace vlastností programu

Prvním krokem ověřování vlastností běhu programu je vytvoření specifikace očekávaného chování systému, který tento program představuje. Chováním systému se rozumí způsob, jakým se tento systém vyvíjí v čase, tedy jak se mění jeho interní stavy, jaké provádí vnější a vnitřní akce, nebo jak jakýmkoli dalším způsobem ovlivňuje své běhové prostředí [4]. Všechny tyto jevy, jejichž posloupnosti udávají chování pozorovaného systému, se souhrnně nazývají událostmi.

Při vytváření specifikace požadovaného chování systému nemusí být zdaleka použity všechny existující události daného systému. Zpravidla je k popisu chování systému použita pouze relativně malá, konečná podmnožina těchto událostí, v závislosti na účelu specifikace nebo důležitosti korektnosti určitých vzorců chování. Jejich výběr je ovšem klíčový pro rozsah informací o systému, které bude možné jeho monitorováním získat.

Chování programu během jeho jednoho běhu lze abstrahovat jako sled *pozorovaných událostí*. Pro rozbor tohoto sledu je důležitá jak potenciální nekonečnost tohoto sledu, tak jeho struktura, protože aktuální stav systému nemusí být definován pouze jednou předcházející událostí, ale celou jejich množinou. Ačkoli je pohled uvažující každou událost jako změnu stavu přirozenější a přehlednější, často příliš nereflektuje realitu a špatně se nad ním definuje pořadí událostí, které mohou vznikat asynchronně.

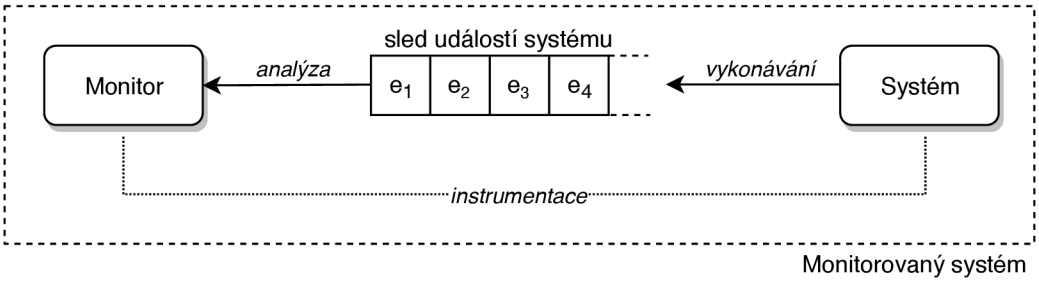
Konkrétní vlastnost systému je definována jako potenciálně nekonečná množina sledů událostí, tedy abstrahovaných správných vzorců chování. Přesné vymezení množiny sledu událostí se pak nazývá specifikací vlastnosti systému. Z hlediska tohoto rozlišení vlastností a specifikace vždy platí, že pro jednu vlastnost může existovat mnoho specifikací, ale specifikovaná vlastnost musí být vždy unikátní a nezávislá na zvoleném zápisu specifikace. Vzhledem k tomu, že je potřeba vlastnosti systému pro monitorování definovat naprosto jednoznačně, nepoužívá se typicky pro specifikaci vlastností programu jazyk přirozený, ale formální.

Jazyků, které jsou použitelné pro specifikaci vlastností systémů, je mnoho a jejich výběr závisí na dané aplikaci. Přímo vykonavatelné jazyky, jako např. stavové automaty, usnadňují monitorování, avšak zpravidla nejsou schopné přehledně vyjádřit vlastnosti na vysoké úrovni abstrakce. Oproti tomu jazyky deklarativní, jakým je např. lineární temporální logika, sice z hlediska monitorování navíc vyžadují konstrukci spustitelného monitoru, avšak dokáží jednoduše a přehledně vyjádřit širší škálu vlastností. Důležitou roli pro volbu specifikačního jazyka může hrát také jeho schopnost popisovat nekonečné sledy událostí, jejich časové souvislosti, nebo datové obsahy.

2.2 Monitorování programových běhů

Po sestavení specifikace kontrolovaného systému je možné přistoupit k ověřování definovaných vlastností, jehož základní schéma ukazuje obrázek 2.1. Sestava pro monitorování se skládá ze tří hlavních součástí – (i) testovaného systému, (ii) monitoru a (iii) mechanismu instrumentace [4].

Monitor je programová entita určená ke sledování průběhu chování daného SUT. Měl by být schopen určit, jestli pozorované chování odpovídá specifikovaným vlastnostem. Jeho verdikt se obecně považuje za definitivní a správný, na což je nutné pamatovat při jeho konstrukci. Monitor je tvořen buďto přímým využitím vykonavatelného specifikačního jazyka, nebo sestaven (typicky automaticky generován) na základě deklarativní formy specifikace požadovaného chování.



Obrázek 2.1: Základní sestava pro monitorování systému podle [4]

Analýza prováděná monitorem je napojena na vykonávání sledovaného programu pomocí *instrumentace*, která může probíhat buď na úrovni spustitelných souborů, nebo zdrojových kódů, kde ji lze provádět i manuálně. Jejím úkolem je monitoru zpřístupňovat vybrané aspekty SUT ve formě událostí, jejichž sledem je definováno skutečné chování sledovaného programu. Instrumentace také definuje vztah mezi vykonáváním systému a monitoru, které mohou být buďto prováděny současně, nebo je nejdříve dokončen běh SUT a až poté spuštěn kód monitoru nad záznamem z jeho běhu.

Zkoumání záznamů z běhů programu, neboli také analýza *post-mortem*, může časově méně zatěžovat původní výpočet[1] a je zpravidla jednodušší ho aplikovat, neboť zpravidla spočívá ve zpracovávání textových záznamů. To ovšem z hlediska monitorování vyžaduje nejen analýzu velkého množství často redundantních textových dat, ale jeho zásadní nevýhodou také je, že vyžaduje, aby vykonávání ověřovaného programu skončilo dříve, než bude provedena jeho analýza. Tím jsou vyloučeny veškeré stále běžící programy, jako např. webové servery, operační systémy atp. Pokud má být použití monitorování co nejobecnější, je tedy vhodnější monitor spouštět souběžně. Tento přístup má ještě jednu nezanedbatelnou výhodu, a tou je možnost průběžného vyhodnocování a hlášení aktuálního stavu, poskytující možnost např. restartovat systém v případě, že ověřování odhalí chybový stav.

[1] Zde však záleží na relativní ceně analýzy prováděné monitorem za běhu, proti častým I/O operacím, které jsou typicky pro analýzu post-mortem zapotřebí.

Kapitola 3

Specifikace temporálních vlastností běhu programu a jejich monitorování

Časové souvislosti mezi výskytem jednotlivých událostí při provádění systému se nazývají temporálními vlastnostmi jeho běhu. Podle Pnueliho [13] je možné správné chování programu úplně definovat pomocí dvou základních temporálních konceptů.

- *Neměnnost*, která je někdy také nazývána vlastností bezpečnosti (angl. safety property)[1], vyjadřuje předpoklad, že je nějaká vlastnost programu splněna po celou dobu jeho běhu. Typicky se vztahuje např. k předpokladu, že se program nikdy nedostane do nežádoucího stavu.
- Zásadnější pro úplnou specifikaci korektního chování programů je však *temporální implikace*, nebo také obecněji vlastnost živosti (angl. liveness property). Ta popisuje časovou následnost dvou událostí, a to tak, že nastane-li první z nich, musí být zaručeno, že od té chvíle někdy v budoucnu bude dosaženo také té druhé.

Tyto temporální vlastnosti programů lze je popisovat více specifikačními jazyky, z nichž nejpoužívanější jsou různé formy temporální logiky. Nejběžnější variantou temporální logiky je *LTL – lineární temporální logika*. Následující sekce se budou věnovat popisu syntaxe a sémantiky tří základních variant LTL vztahujících se k popisu (i) neomezeného množství budoucích stavů, (ii) omezené budoucnosti a (iii) minulosti systému. Zároveň budou vyhodnoceny jejich základní výhody a nevýhody z hlediska použití jako specifikace vlastností programů pro monitorování jejich běhů a budou také představeny základy monitorování takto specifikovaných vlastností.

3.1 Lineární temporální logika

Základ pro vysvětlování matematických pojmů této sekce byl převzat z článku Kestena et al. [10]. V předchozí kapitole bylo vysvětleno, že na jeden běh programu lze nahlížet jako na sled událostí systému, který tento program reprezentuje. Tento sled lze poté strukturovat do stavů, které se typicky skládají z více než jedné události. Z hlediska temporální logiky je

[1] viz Moller a Struth [11]

tedy běh programu lineární posloupností stavů, v nichž se implementovaný systém během svého vykonávání nachází. Tyto stavy je v temporální logice možné popisovat pomocí *stavových formulí*, které se skládají z výroků o stavových proměnných systému a booleovských operátorů $\neg$, $\vee$ nebo $\wedge$. Pro specifikaci vzájemných časových vztahů mezi stavy běhu jsou pak zavedeny základní *temporální operátory*: $\bigcirc$ – *Next* a $\mathcal{U}$ – *Until*, které spolu s booleovskými operátory spojují stavové formule do formulí *temporálních*.

Platnost temporální formule se vztahuje ke konkrétní sekvenci stavů systému $\sigma : s_1, s_2, s_3, \ldots$ a aktuálně zkoumané pozici v této sekvenci $j \geq 1$. Pokud formule p pro pozici j dané sekvence stavů σ platí, značíme tuto skutečnost jako $(\sigma, j) \models p$ nebo také $s_j \models p$. Sémantika základních temporálních operátorů je poté definována následovně.

$$s_j \models \bigcirc\, p \quad \Longleftrightarrow \quad s_{j+1} \models p \tag{3.1}$$

$$s_j \models p\, \mathcal{U}\, q \quad \Longleftrightarrow \quad \exists k,\ k \geq j\ s_k \models q \wedge \forall i,\ j \leq i < k\ s_i \models p \tag{3.2}$$

Tyto operátory umožňují, jak popisují např. Bartocci a Falcone [3], definovat další dva, které odpovídají popisu základních temporálních konceptů uvedených na začátku této kapitoly. Neměnnost je popsána operátorem $\Box$ – *Always*, pro který platí, že $\Box\, p \equiv p\, \mathcal{U}\, false$. Temporální implikaci pak lze definovat pomocí operátoru $\Diamond$ – *Eventually*, jehož sémantiku lze vyjádřit buď jako $\Diamond\, p \equiv \neg\Box\neg p$, nebo jako $\Diamond\, p \equiv true\, \mathcal{U}\, p$, tzn. že v některém z budoucích stavů bude p platit.

3.2 Monitorování formulí lineární temporální logiky

LTL budoucího času tedy nabízí dostatečné prostředky pro úplnou specifikaci správného chování systému. Jejím rozšířením mimo jiné i o regulární výrazy dokonce vznikl standardizovaný[2] jazyk pro specifikaci vlastností - PSL (Property Specification Language) používaný v hardwarovém průmyslu.

Vzhledem k tomu, že se ale zabývá popisem budoucích stavů, o kterých za běhu programu nemáme žádné dostupné informace, je její použití pro monitorování poměrně obtížné. Pro ověřování takto definovaných vlastností se často používá překlad specifikace do Büchiho [8, 6], nebo také do alternujícího automatu [7], popř. Havelund a Roşu [14] navrhli způsob vyhodnocování formulí LTL nad konečnými běhy založené na přepisování iniciálních formulí podle dosavadního průběhu vykonávání programu.

Transformace specifikace do automatu rozhodně není triviální úkol a přepisování formulí vede k časově náročnému vyhodnocování. A co je důležitější, už ze sémantiky LTL není pro některé vlastnosti, které jsou specifikované její pomocí, možné rozhodnout, ve kterém stavu nedokončeného vykonávání programu byla vlastnost porušena. Průběžné vyhodnocování běhu programu monitorem by tedy vedlo (a) k velkému množství falešných poplachů, pokud by monitor určoval vlastnosti běhu jen podle dosavadní historie, nebo (b) k neúplným průběžným výsledkům, pokud by monitor čekal, co přijde v následujících stavech. Proto se v praxi používají také další varianty formálních systémů lineární temporální logiky.

3.3 Metrická temporální logika

Problém nedostatečné přesnosti specifikace programů pomocí LTL pro jejich monitorování do určité míry řeší použití *metrické temporální logiky* – *MTL* [12]. Ta používá booleovské operátory $\wedge, \vee$ a $\neg$ v kombinaci s upraveným temporálním operátorem $p\, \mathcal{U}_I\, q$, kde $I \subseteq \langle 0, \infty)$

[2] IEEE 1850 Standard for Property Specification Language (PSL)

je interval následujících stavů, na které je platnost tohoto operátoru omezena. Pokud je tedy $I = \langle 0, \infty)$ jedná se o neomezený operátor *until* se stejnou sémantikou, jak byla definována formulí 3.2.

Jeho pomocí lze poté dodefinovat omezený operátor *eventually* jako $\Diamond_I\ p \equiv true\ \mathcal{U}_I\ p$, který říká, že se událost p musí objevit v rámci daného intervalu následujících stavů. Pokud by tedy např. bylo $I = (0, 1\rangle$, pak sémantika takto omezeného operátoru *eventually* odpovídá sémantice operátoru *next* z formule 3.1. Podobně lze dodefinovat také omezený operátor *always* jako $\Box_I\ p \equiv \neg\Diamond_I \neg p$.

3.3.1 Převoditelnost MTL na LTL

Formule MTL s takto omezenými operátory je možné převádět do LTL pomocí rekurzivního vyjádření operátorů LTL, které Kesten et al. [10] také označuje jako tranzitivní relace mezi dvěma po sobě jdoucími stavy, ve kterých má formule s daným operátorem platit. Pro běh systému $\sigma : s_1, s_2, s_3, \ldots$ je tedy možné sémantiku temporálních operátorů vyjádřit kromě formulí z podkapitoly 3.1 také následovně:

$$s_j \models \bigcirc\ p \iff s_{j+1} \models p \tag{3.3}$$
$$s_j \models p\ \mathcal{U}\ q \iff s_j \models p \vee (j > 1 \wedge s_j \models q \wedge s_{j+1} \models p\ \mathcal{U}\ q) \tag{3.4}$$
$$s_j \models \Box\ p \iff s_j \models p \wedge s_{j+1} \models \Box\ p \tag{3.5}$$
$$s_j \models \Diamond\ p \iff s_j \models p \vee s_{j+1} \models \Diamond\ p \tag{3.6}$$

Pro omezené operátory MTL je pak právě díky tomu, že je jejich platnost omezena na daný počet stavů, možné je vyjádřit pomocí dosazení do těchto rekurzivních vyjádření, kde lze každé $s_{j+1} \models r$ jednoduše přepsat na $\bigcirc\ r$. Každou formuli MTL lze tedy přepracovat zase zpět do základní LTL za použití pouze operátoru *next*, jako v následujících příkladech.

$$p\ \mathcal{U}_{\langle 0,3\rangle}\ q \iff (p \wedge (\bigcirc\ q \vee (\bigcirc\ p \wedge (\bigcirc\bigcirc\ q \vee (\bigcirc\bigcirc\ p \wedge (\bigcirc\bigcirc\bigcirc\ q))))))) \tag{3.7}$$
$$\Diamond_{\langle 0,3\rangle}\ p \iff (p \vee \bigcirc\ p \vee \bigcirc\bigcirc\ p \vee \bigcirc\bigcirc\bigcirc\ p) \tag{3.8}$$
$$\Box_{\langle 0,3\rangle}\ p \iff (p \wedge \bigcirc\ p \wedge \bigcirc\bigcirc\ p \wedge \bigcirc\bigcirc\bigcirc\ p) \tag{3.9}$$

3.3.2 Průmyslové využití MTL pro specifikaci programů

V praxi tuto variantu LTL ke specifikaci požadovaných vlastností vyvíjeného softwaru využívá společnost Honeywell v nástroji *ForReq* (Formalizing Requirements) [1]. Jedná se o specializovaný nástroj pro podporu zápisu požadavků na software do strojově zpracovatelné formy. Jeho vstupem jsou požadavky formulované speciálně formátovaným přirozeným jazykem se sémantikou formulí MTL a výstupem jsou pak odpovídající formule LTL využívající pouze operátory *always* a *next*.

Tento nástroj lze použít např. spolu s nástrojem *DiVinE* (Distributed Verification Environment project) [2], který slouží pro verifikaci systémů na základě požadavků, prováděnou v podobě porovnávání programu s jeho formálním modelem specifikovaným pomocí formulí LTL, nebo jako rozšíření pro *HiLiTE* (Honeywell Integrated Lifecycle Tools and Environment) [5], což je nástroj pro certifikovanou verifikaci leteckých systémů na základě požadavků. Tato práce byla právě díky těmto existujícím nástrojům společností Honeywell do jisté míry podporována, například poskytnutím vzorové specifikace generované nástrojem *ForReq* a také ukázky kódu, který může být na základě této specifikace ověřován. Tyto materiály jsou v kapitole 7.3 použity k demonstraci reálné aplikace nástroje, který byl v rámci této práce implementován.

Podobné úpravy LTL, jakou je tato popsaná úprava vycházející z MTL, situaci z hlediska monitorování výrazně zlepšují, protože je možné chybový stav identifikovat do určitého konečného počtu stavů běhu sledovaného systému. Stále ovšem pro vyhodnocování MTL formulí v každém stavu zůstává nejistota o budoucím průběhu vykonávání sledovaného systému, která podmiňuje použití složitých monitorovacích algoritmů. Existuje ovšem další varianta LTL, která je z tohoto hlediska pro monitorování velmi vhodná a převoditelná na formule LTL se sémantikou MTL, která využívá pouze omezené operátory.

3.4 Lineární temporální logika minulého času

Tato varianta LTL se používá k popisu temporálních vlastností *předchozích* stavů systému a její operátory jsou symetrické s těmi pro budoucí čas. Sémantika operátorů je převzata z článku Havelunda a Roşu [9], protože z daného článku vycházejí i další myšlenky této práce, jak bude uvedeno později, viz kapitola 4. V jiných zdrojích však může být definována s drobnými odlišnostmi, především v inicializaci platnosti formulí v počátečním stavu[3].

Označení	**Syntaxe**		**Sémantika**	
$\odot$ – *Last*	$s_j \models \odot\, p$	$\iff$	pro $j > 1$ $s_{j-1} \models p$, $s_1 \models p$	(3.10)
$\mathcal{S}$ – *Since*	$s_j \models p\, \mathcal{S}\, q$	$\iff$	$\exists n,\ 1 \leq n \leq j,\ s_n \models q \wedge$	(3.11)
			$\forall i,\ \ n < i \leq j,\ s_i \models p$	
$\boxdot$ – *Globally*	$s_j \models \boxdot\, p$	$\iff$	$\forall n,\ 1 \leq n \leq j\ s_n \models p$	(3.12)
$\diamondsuit$ – *Previously*	$s_j \models \diamondsuit\, p$	$\iff$	$\exists n,\ 1 \leq n \leq j\ s_n \models p$	(3.13)

Tyto operátory se často přidávají jako rozšíření k základní variantě LTL, protože umožňují ty některé formule čisté LTL vyjádřit mnohem přehlednějším a kratším zápisem. V literatuře se často jako ptLTL označuje právě tato kombinovaná varianta, která má stejnou vyjadřovací sílu, jako základní LTL. V této práci se ovšem pod ptLTL rozumí používání operátorů vztažených pouze k minulosti.

PtLTL v tomto pojetí není z hlediska vyjadřovací síly s LTL úplně ekvivalentní. Konkrétně se tento rozdíl projevuje např. u formule LTL

$$\Box(p \implies \diamondsuit\, q), \tag{3.14}$$

která vyjadřuje, že vždy, jakmile se objeví událost p, začne se očekávat příchod události q, která se musí od daného okamžiku vyskytnou alespoň jednou do konce běhu systému. Ačkoli jsou operátory ptLTL do určité míry sémanticky symetrické k operátorům LTL, tuto vlastnost v ptLTL specifikovat nelze.

Jak bylo ovšem popsáno v kapitole 3.2, takto neurčitá specifikace vlastnosti není pro monitorování běhu programu příliš žádoucí. To řeší použití výše popsané MTL, která ovšem kvůli nutnosti vytváření predikcí o budoucím průběhu vykonávání sledovaného systému vyžaduje použití monitorovacího algoritmu, který bude srovnatelně komplikovaný jako pro základní LTL. V sekci 3.3.1 však bylo ukázáno, jakým způsobem je možné MTL s pouze omezenými operátory převést na formule LTL, které využívají pouze booleovské operátory v kombinaci s temporálním operátorem *next*, jehož sémantická dualita s ptLTL operátorem *last* je nesporná.

Formule MTL využívající pouze omezené temporální operátory je tedy možné přes LTL převést do ptLTL, což bude blíže prakticky demonstrováno v kapitole 7.3. Ty formule LTL,

[3] Např. Kesten et al. [10] považuje formuli s operátorem *last* v počátečním stavu běhu programu vždy za neplatnou.

které není sémanticky možné v ptLTL vyjádřit, nejsou z hlediska specifikace požadovaných vlastností programu dostatečně určité, a tak ztrátou možnosti jejich využití je jen vynuceno vytváření kvalitnější specifikace. Stejně jako u přepisu MTL do LTL, nemusí být výsledné formule převodu MTL do ptLTL pro člověka příliš čitelné, natož snadno vytvořitelné. Jak však bylo zmíněno dříve, pro tento problém existuje už i praktické řešení ve formě pomocného nástroje *ForReq* pro formalizaci požadavků. Jak bude později nastíněno v kapitole 7.3, není vyloučeno, že nástroj *ForReq* bude v dohledné době použitelný i pro konstrukci specifikace pro nástroj implementovaný v rámci této práce.

Nespornou výhodou ptLTL pro monitorování pak je především to, že k jejímu vyhodnocování stačí vhodným způsobem zaznamenávat předchozí průběh programu a není tedy nutná žádná konstrukce složitých modelů specifikace nebo predikcí pro následující stavy běhu. Odpověď na to, zda pro daný stav běhu programu dané ptLTL formule platí, je možné dát vždy s naprostou určitostí.

Kapitola 4

Zvolený princip monitorování temporálních vlastností běhů programů

Platforma Testos představená v kapitole 1 má snahu cílit vždy na co nejširší možnosti použití svých nástrojů. Proto se ani nástroj vzniklý v rámci této práce neomezuje na použití pro pouze konečné běhy programů a, aby umožnil kooperaci s nekonečnými programy, provádí vytvářený monitor vyhodnocování temporálních vlastností přímo za běhu prověřovaného programu. Aby toho byl schopen co neefektivněji a umožňoval hlásit průběžné závěry verifikace, je ke specifikaci kontrolovaných vlastností použita lineární temporální logika pro minulý čas (past-time Linear Temporal Logic – ptLTL) v takové formě, jak byla představena v předchozí kapitole.

To umožňuje pro vyhodnocování platnosti formulí specifikace použít algoritmus popsaný Havelundem a Roşu [9], založený na rekurzivním vyjádření platnosti operátorů, které je popsáno v následující sekci. Protože je vzniklý nástroj určen pro jazyky C/C++, je instrumentace monitoru, blíže popsaná v podkapitole 4.3, zajištěna zpřístupněním stavových proměnných ověřovaného programu v hlavičkovém souboru, který bude s monitorem sdílen.

4.1 Princip vyhodnocování platnosti formulí

Hlavní myšlenka algoritmu, který byl vybrán pro účely této práce, vychází z rekurzivního vyjádření operátorů ptLTL, které už bylo pro klasickou LTL popisováno v kapitole 3.3.1. V článku Kestena et al. [10] je jasně ukázáno, že je pomocí tohoto vyjádření možné popsat sémantiku libovolného z temporálních operátorů představených v kapitole 3.

Platnost každé temporální formule lze tedy pro aktuálně zkoumaný stav běhu programu vyjádřit pomocí podmínek, které obsahují pouze (a) booleovské operátory a (b) platnost té samé formule ptLTL v předešlém stavu. Pro operátory ptLTL představené v kapitole 3.4 je obecné rekurzivní vyjádření pro stavy běhu $\sigma = \ldots s_{j-1}, s_j, s_{j+1} \ldots$ následující.

$$s_j \models \odot\, p \iff s_{j-1} \models p \tag{4.1}$$

$$s_j \models p\, \mathcal{S}\, q \iff s_j \models q \vee (j > 1 \wedge s_j \models p \wedge s_{j-1} \models p\, \mathcal{S}\, q) \tag{4.2}$$

$$s_j \models \boxdot\, p \iff s_j \models p \wedge s_{j-1} \models \boxdot\, p \tag{4.3}$$

$$s_j \models \diamondsuit\, p \iff s_j \models p \vee s_{j-1} \models \diamondsuit\, p \tag{4.4}$$

Tyto vztahy platí pro stavy, kde $j > 1$, je tedy ještě nutné správně inicializovat platnost všech částí sledovaných formulí, kde je použit některý temporální operátor, v počátečním stavu. Např. v [10] se předpokládá, že všechny výroky týkající se předchozího stavu v tom počátečním neplatí. V Havelund a Roşu [9] zase platnost výroků v tomto virtuálním stavu před inicializací odvodzují od platnosti v aktuálním stavu, což je také přístup použitý v této práci, a tedy $s_0 \models \odot\, p \Leftrightarrow s_0 \models p$, to samé platí pro $\boxdot\, p$ i $\diamondsuit\, p$, a $s_0 \models p\ \mathcal{S}\ q \Leftrightarrow s_0 \models q$.

Havelund a Roşu [9] také kromě těchto základních operátorů zavádějí použití několika dalších podpůrných operátorů pro monitorování.

Označení	**Syntaxe**		**Sémantika**	
$\uparrow$ – *Up*	$s_j \models \uparrow p$	$\iff$	pro $j > 1$ $s_{j-1} \not\models p \wedge s_j \models p$, $s_1 \not\models \uparrow p$	(4.5)
$\downarrow$ – *Down*	$s_j \models \downarrow p$	$\iff$	pro $j > 1$ $s_{j-1} \models p \wedge s_j \not\models p$, $s_1 \not\models \downarrow p$	(4.6)
$\mathcal{S}_w$ – *Weak Since*	$s_j \models p\ \mathcal{S}_w\ q$	$\iff$	$s_j \models p\ \mathcal{S}\ q \vee s_j \models \boxdot\, p$	(4.7)
$[,)_s$ – *Strong Interval*	$s_j \models [p,q)_s$	$\iff$	$\exists n,\ 1 \leq n \leq j\ s_n \models p \wedge \forall i,\ n \leq i \leq j\ s_i \not\models q$	(4.8)
$[,)_w$ – *Weak Interval*	$s_j \models [p,q)_w$	$\iff$	$s_j \models [p,q)_s \vee s_j \models \boxdot \neg q$	(4.9)

Operátor *Weak Since* je zjemněním původně definovaného *Since*, na rozdíl od kterého za korektní chování považuje i případ, že událost q nikdy nenastane. Pro přehlednost bude dále původně definovaná verze tohoto operátoru z kapitoly 3.4 označována jako *Strong Since* – $\mathcal{S}_s$. Další čtyři operátory nejsou potřebné pro správnou specifikaci temporálních vlastností programů, ale dle Havelunda a Roşu [9] se ukazují být užitečné při vytváření co nejpřehlednějšího zápisu požadovaného chování programu. I tyto lze jednoduše definovat rekurzivně.

$$s_j \models p\ \mathcal{S}_w\ q \iff s_j \vDash q \vee (s_j \vDash p \wedge (j > 1 \Rightarrow s_{j-1} \vDash p\ \mathcal{S}_w\ q)) \tag{4.10}$$
$$s_j \models [p,q)_s \iff s_j \not\models q \wedge (s_j \vDash p \vee (j > 1 \wedge s_{j-1} \vDash [p,q)_s)) \tag{4.11}$$
$$s_j \models [p,q)_w \iff s_j \not\models q \wedge (s_j \vDash p \vee (j > 1 \Rightarrow s_{j-1} \vDash [p,q)_s)) \tag{4.12}$$

Rekurzivní vyjádření operátorů $\uparrow$ a $\downarrow$ není třeba uvádět, protože jejich sémantika sama o sobě mluví pouze o dvou po sobě jdoucích stavech.

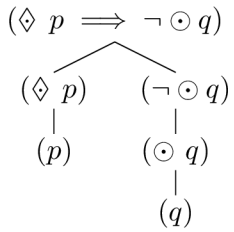
4.2 Práce implementovaného nástroje

Nástroj implementovaný v rámci této práce má za úkol pro danou specifikaci pomocí formulí ptLTL vygenerovat kód monitoru. Ten bude v každém stavu běhu programu schopen vyhodnotit, jestli byly všechny popsané temporální vlastnosti dodrženy, za použití principů představených v předchozí sekci, pouze se záznamem předchozí platnosti každé formule.

Každá formule ptLTL uvedená ve specifikaci je nejprve rozdělena do stromu dílčích podformulí, jak je znázorněno na obrázku 4.1. Každá tato podformule je reprezentována svým uzlem ve vytvořeném stromu a obsahuje buďto (i) právě jeden operátor a odkazy na operandy, na které má být daný operátor aplikován, nebo (ii) čtení hodnoty stavové proměnné, které je vždy listovým uzlem stromu.

Hodnota každého uzlu stromu je poté v paměti monitoru reprezentována párem booleovských hodnot, pro platnost (i) v předchozím a (ii) v aktuálně zkoumaném stavu běhu programu. Hodnota platnosti každé formule, označená jako platnost v předchozím stavu, je pro počáteční stav vhodně inicializována, a před vyhodnocením každého dalšího stavu jsou předešlé hodnoty přepsány hodnotami původně aktuálními.

V každém stavu běhu programu, který monitor sleduje, jsou formule vyhodnocovány od kořenových uzlů směrem vzhůru, kde je buďto (i) kořenová formule specifikace pokud zde



Obrázek 4.1: Příklad rozkladu temporální formule do stromu podformulí.

byla uvedena pouze jediná formule, nebo (ii) konjunkce všech specifikačních formulí. Pokud je aktuální hodnotě kořene stromu při vyhodnocování přiřazena hodnota `false`, znamená to, že byla některá ze specifikovaných temporálních vlastností porušena.

4.3 Instrumentace generovaného monitoru

Aby mohl vytvářený monitor vyhodnocovat platnost formulí specifikace, je potřeba, aby měl v každém stavu běhu k dispozici všechny v té době aktuální hodnoty proměnných, nad kterými je specifikace definována. Protože je vytvořený nástroj určen pro jazyky C a C++, je pro tento účel možné použít sdílení deklarací stavových proměnných zkoumaného systému pomocí hlavičkového souboru, který bude připojen i do kódu generovaného monitoru.

K úplnému propojení monitoru a zkoumaného programu je poté potřeba kód monitoru k programu přikompilovat. Znovu z hlediska cílového programovacího jazyka bylo vybráno uživatelsky nejsnazší řešení, a to vytvořit z kódu monitoru knihovnu, kterou lze při kompilaci k prověřovanému programu připojit pomocí příznaků kompilátoru. Konkrétní postup, potřebný k zapojení monitoru generovaného vytvořeným nástrojem do vlastního projektu, je uveden v příloze B.

Zbývající velmi důležitou otázkou z hlediska monitorování temporálních vlastností je, jak určit, kdy sledovaný program přešel do dalšího stavu a kdy je tedy potřeba znovu vyhodnotit platnost formulí specifikace. Rozlišení jednotlivých stavů není možné univerzálně algoritmizovat, a není tedy možné automaticky, bez jakékoli úpravy kódu zkoumaného programu, tyto okamžiky detekovat. Označení hranic stavů je spíše součástí specifikace, jejíž splnění má být ověřováno.

Je tedy nevyhnutelné upravit kód prověřovaného programu tak, aby bylo zřejmé, kdy má monitor provést ověření aktuálního stavu. Intuitivně tak lze učinit přímo voláním vyhodnocující funkce monitoru, implementovaný nástroj pak navíc umožňuje i použití specifických komentářů[1] ve zdrojových kódech, které budou voláním implementovaného nástroje nahrazeny.

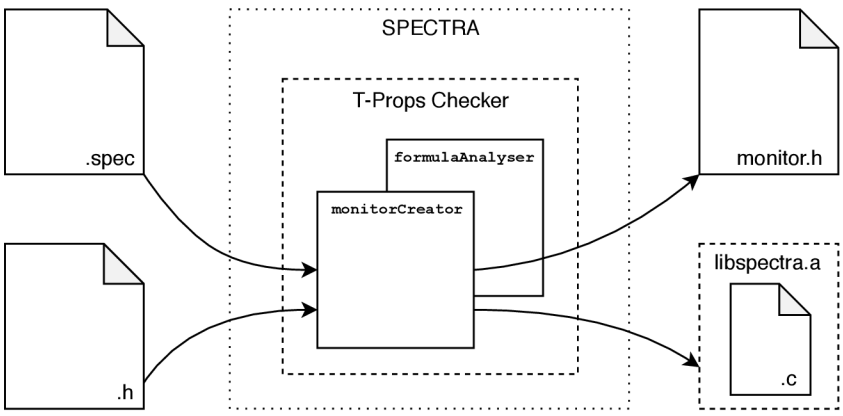
[1] Viz kapitola 5.1

Kapitola 5

Nástroj pro ověřování temporálních vlastností běhů programů

Jak bylo zmíněno v kapitole 1, výsledný nástroj měl být implementován jako podsystém nástroje Spectra [16] platformy Testos [15]. Nástroj Spectra byl však do vytvoření této práce pouze konceptem, proto bylo nejprve nutné vytvořit také snadno rozšiřitelný implementační základ pro tento soubor nástrojů, které mají v budoucnu vzniknout. Až poté bylo možné do vzniklého nastroje Spectra, jako první z jeho subsystémů, implementovat nástroj pro ověřování temporálních vlastností s názvem *T-Props Checker*, který je výsledkem této práce. Jeho vstupem, jak ilustruje obrázek 5.1, jsou dva soubory. Obsahem jednoho z nich je specifikace ověřovaných temporálních vlastností programu, jejíž struktura odpovídá popisu v sekci 5.1. Dalším vstupem je pak hlavičkový soubor s deklarací proměnných, které pro formule specifikace slouží jako stavové, nebo jsou k výpočtu hodnot stavových proměnných používány, jak bude vysvětleno v následující podkapitole.

Výstupem nástroje T-Props Checker je statická knihovna nazvaná `libspectra.a`, obsahující zkompilovaný kód monitoru vygenerovaného pro danou specifikaci, a hlavičkový soubor `monitor.h`. Podrobný návod, jak tyto soubory získat a použít k ověřování temporálních vlastností vlastního projektu je uveden v příloze B.



Obrázek 5.1: Schéma použití implementovaného nástroje T-Props Checker, zobrazující jeho vstupy, výstupy a základní součásti.

Generovaný monitor je implementován tak, aby odhalil jakékoli porušení dané specifikace. V případě, že sledovaný běh programu dosáhne stavu, který porušuje některou z formulí specifikace, je tato skutečnost oznámena výpisem na chybový výstup. Tento výpis obsahuje seznam všech podformulí, jejichž platnost se při jejich posledním vyhodnocování změnila, společně s údajem, od kdy byla do té doby jejich platnost stabilní. Tyto údaje mají přispět k nalezení chyby v logice prověřovaného programu.

Nový monitor je nutné generovat pouze při změně specifikace, na obsahu zdrojových kódů ověřovaného programu monitor závislý není. Při změnách zdrojových kódů je ovšem potřeba se zvláštní opatrností přistupovat k modifikaci hlavičkového souboru, který je monitoru nasdílen, a jehož změna by mohla vést k chybám při následné kompilaci.

V následujících sekcích bude nejprve představena požadovaná struktura vstupního souboru se specifikací a jeho závislosti na přikládaném hlavičkovém souboru. Dále bude nastíněna struktura implementovaného nástroje a také generovaného monitoru temporálních vlastností.

5.1 Struktura souboru se specifikací sledovaných temporálních vlastností

Soubor, ve kterém je vytvořenému nástroji předána specifikace temporálních vlastností k prověření, má typicky následující dvě části:

1. seznam formulí temporální logiky, které definující sledované vlastnosti,
2. definice výpočtů všech proměnných, které jsou v předchozích formulích používány jako stavové, ale v přiloženém hlavičkovém souboru nejsou deklarovány.

Druhá část může být vynechána pouze v případě, že jí není třeba, protože všechny stavové proměnné jsou deklarovány jako proměnné typu `bool` v přiloženém hlavičkovém souboru.

Ověřované vlastnosti jsou specifikovány pomocí formulí lineární temporální logiky minulého času, popsané v kapitole 3.4. Do vstupního souboru pro implementovaný nástroj je formule `F` zapsána ve speciální prefixové notaci, definované následovně.

```
F :=  varName |  (!F) | (-> F F) | (|| F F) | (&& F F) | ( ^ F F)
                 (operátory výrokové logiky)
      (G F) |    (P F) | (L F) | (Ss F F) | (Sw F F)
                 (temporální operátory podle kapitol 3.4 a 4.1)
      (U F) |    (D F) | (Is F F) | (Iw F F)
                 (monitorovací operátory definované v kapitole 4.1)
```

`VarName` představuje stavovou proměnnou typu `bool` a operátor `->` nese význam implikace. Ostatní čtyři operátory na prvním řádku vychází z notací jazyka C a jedná se tedy o operace `or`, `and` a `xor`, které smí mít i více operandů, než pouze dva. První tři temporální operátory odpovídají prvním písmenům svých názvů: `G` – *globally*, `P` – *previously* a `L` – *last*. Další dva jsou potom užší specifikací významu operátoru *since* jako silný (*strong since* – `Ss`) a slabý (*weak since* – `Sw`), jejichž sémantika byla oddělena v kapitole 4.1. V té jsou také popsány významy monitorovacích operátorů, zde `U` – *up*, `D` – *down*, `Is` – *strong interval* a `Iw` – *weak interval*.

V zápisu formulí se nesmí vyskytovat žádné další závorky, a pokud by tedy jako stavová proměnná měla sloužit návratová hodnota funkce, deklarované v přiloženém hlavičkovém

souboru, je potřeba ji vložit do pomocné proměnné, definované ve druhé části tohoto souboru, která bude popsána vzápětí. Před každým jménem stavové proměnné musí v případě, že není ohraničeno závorkami, být alespoň jeden bílý znak a tato jména musí odpovídat standardním konvencím pro pojmenovávání proměnných v jazycích C a C++. Mezi jednotlivými částmi formulí se může vyskytovat libovolný počet bílých znaků, před ani za závorkou nemusí být ani jeden. V této části souboru může být uvedeno libovolné množství formulí, výsledkem vyhodnocování splnění specifikace pak bude konjunkce jejich platností. Validně zapsanou formulí tedy může být např. libovolný z následujících zápisů formule $\Diamond\, p \implies \neg \odot q$.

```
(->
  (P p)
  (!
    (L q)
  )
)
```

```
(->(P(p))(!(L(q))))
```

Druhá část specifikačního souboru je od první oddělena řádkem

```
DEFINITIONS:
```

Za ním pak následují řádky s definicemi výpočtů všech proměnných `varName` použitých v předchozích formulích, aniž by byly deklarovány v předaném hlavičkovém souboru s deklaracemi. Proměnné, které v hlavičkovém souboru deklarovány jsou, mohou být libovolného datového typu, popř. v něm mohou být deklarovány funkce. Vše, co je deklarováno v přiloženém hlavičkovém souboru, popř. zahrnuto z dalších hlavičkových souborů, a jakoukoli vestavěnou funkci nebo konstantu jazyka C popř. C++, je možné použít pro definici hodnot stavových proměnných formulí, avšak pouze takovým způsobem, aby výsledným datovým typem definice stavové proměnné byl typ `bool`. Každá proměnná musí být definována na samostatném řádku, který odpovídá formátu

```
name : expression
```

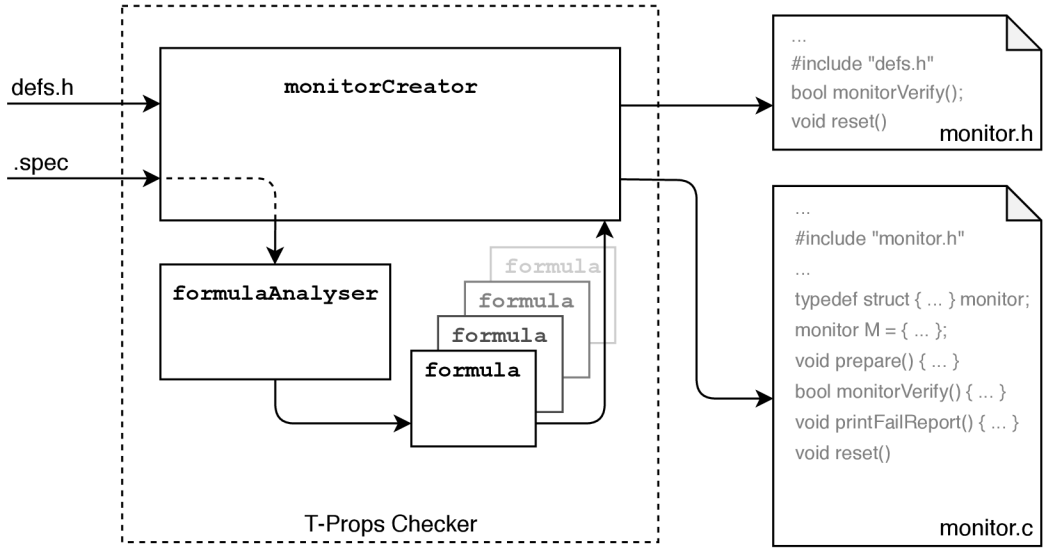
Na konci specifikačního souboru se může volitelně vyskytovat seznam zdrojových souborů, které jsou definovány relativní cestou od souboru se specifikací, oddělený od předchozí části souboru řádkem

```
SOURCES:
```

Tento seznam slouží pro automatizované dokončení instrumentace generovaného monitoru, která je ve jmenovaných souborech prozatím naznačena pouze komentářem

```
// T-Props Checker verify
```

Při nahrazování těchto komentářů spouštěním verifikace je ověřováno, jestli daný zdrojový soubor obsahuje odkaz na generovaný hlavičkový soubor `monitor.h`, a v případě že ne, je i tento automaticky doplněn.



Obrázek 5.2: Schéma práce implementovaného nástroje T-Props Checker, zobrazující způsob zpracování jeho vstupů a podrobnou podobu výstupů.

5.2 Struktura nástroje

Výsledný nástroj je implementován v jazyce C++ a veškerá jeho funkcionalita je tedy formulována do objektových modulů. Mezi použité třídy patří:

`formulaAnalyser` – načítá obsah souboru se specifikací do interních struktur. Formule specifikace jsou rozděleny do stromu podformulí a je z nich extrahován seznam všech použitých jmen stavových proměnných. Ke každé formuli je také vytvořen řetězec s jejím zápisem, kde jsou kompletně vypsány také všechny její podformule, který je používán generovaným monitorem při chybových hlášeních.

`formula` – reprezentace jedné formule specifikace, udávající určení operátoru a odkazy na všechny případné operandy, nebo jméno stavové proměnné. Dále také každá formule obsahuje záznam o počáteční a koncové pozici ve specifikačním souboru, mezi kterými se nachází její zápis.

`monitorCreator` – na základě výsledků zpracování specifikace třídou `formulaAnalyser`, jak ilustruje obrázek 5.2, vygeneruje soubory (i) `monitor.h`, který obsahuje odkaz na poskytnutý hlavičkový soubor a deklarace funkcí volaných pro (a) spuštění verifikace aktuálního stavu běhu programu a (b) pro novou inicializaci ověřování, a (ii) `monitor.c`, který obsahuje implementaci algoritmu ověřujícího vlastnosti předané ve specifikaci. Pokud byl ve specifikaci uveden seznam zdrojových souborů, ve kterých se mohou nacházet instrumentační komentáře, nahradí je voláním funkce pro spuštění verifikace a původní verzi souboru uloží s přidanou příponou `.old`.

Jak již bylo zmíněno dříve, implementovaný nástroj je subsystémem nástroje Spectra. Ten při výběru nástroje T-Props Checker provede potřebné zpracování parametrů, následně s pomocí třídy `monitorCreator` vygeneruje soubory `monitor.h` a `monitor.c`. Ty poté zkompiluje a z výsledného objektového souboru nakonec vytvoří statickou knihovnu `libspectra.a`.

5.3 Struktura kódu generovaného monitoru

Kód monitoru je na rozdíl od samotného nástroje generován v jazyce C, který je kompatibilní i s programy implementovanými a kompilovanými jako C++, avšak ne naopak. Jsou v něm nejprve uvedeny definice funkcí pro výpočet stavových proměnných, které nebyly deklarovány v předaném hlavičkovém souboru. Poté jsou vytištěny další části kódu naznačené v obrázku 5.2.

Nejdříve je definována datová struktura vytvořená přímo na míru dané specifikaci. Ta obsahuje a pojmenovává veškerý paměťový prostor potřebný pro ověřující výpočty. Jedná se např. o aktuální a předešlou hodnotu platnosti pro každou formuli specifikace, jejich řetězcové reprezentace pro případná chybová hlášení, nebo také pomocné proměnné pro výpočty délky neměnnosti předchozí hodnoty platnosti pro každou formuli.

Dále jsou definovány čtyři funkce obsahující veškerou zbývající potřebnou logiku monitoru.

`prepare` – provede inicializaci předchozích hodnot platnosti pro všechny formule před prvním vyhodnocením splnění specifikace.

`monitorVerify` – nachystá hodnoty předchozích platností formulí buďto, pokud je volána poprvé, pomocí funkce `prepare`, nebo přesunem hodnot platností, které byly v předchozím kole ověřování aktuální. Poté provede vyhodnocení platností všech formulí v aktuálním stavu podle rekurzivního vyjádření operátorů z kapitoly 4.1. Nakonec zkontroluje platnost kořenové formule a v případě potřeby vytiskne chybové hlášení. Vrací booleovskou hodnotu odpovídající platnosti kořenové formule, která může sloužit k reakci ověřovaného systému na detekovanou chybu např. tak, že se restartuje.

`reset` – v případě, že by byl ověřovaný systém za běhu znovu inicializován, je možné zavoláním této funkce monitoru nastavit, že další kolo verifikace má hodnoty předchozích platností formulí připravit pomocí funkce `prepare`, a tedy zapomenout na jakoukoli předchozí chybu.

`printFailReport` – vytiskne na chybový výstup zprávu o chybovém stavu. Pro každou formuli, jejíž hodnota se v aktuálním stavu běhu programu vyhodnotila jinak než v předchozím, a tato změna mohla mít vliv na platnost kořenové formule, jsou vytištěny následující řádky:

```
specification/file/path:<line(s)>:<columns>:
    <formula>
    turned <bool> after being <!bool> for last <k> states
```

Je tedy nejprve identifikována přesná původní lokace dané formule ve specifikačním souboru a poté je proveden výpis obsahu formule vygenerovaný na základě její sémantiky. Jeho formátování se proto může mírně lišit od původní formy ve specifikačním souboru, obsahuje např. vždy jen jednu mezeru mezi každou dvojicí operátor – operand, nebo operand – operand. Nakonec je uvedena aktuální hodnota platnosti dané formule společně s údajem, jak dlouho platil předchozí stav, který by měl sloužit k jednodušší identifikaci sémantického problému, který by tato změna platnosti dané formule mohla indikovat.

5.4 Vzorový příklad fungování implementovaného nástroje

Nástroj Spectra, ve kterém je implementovaný nástroj integrován, očekává na vstupu v případě volby nástroje T-Props Checker jako parametry cestu (i) k souboru se specifikací a (ii) k hlavičkovému souboru s deklaracemi pro stavové proměnné. Je spuštěn příkazem

```
$ ./spectra --tpc -s cesta/ke/specifikaci -d cesta/k/definicim.h
```

kde volba `--tpc` vybere použití nástroje T-Props Checker a parametry `-s` a `-d` slouží k určení cesty ke vstupním souborům se specifikací a definicemi. Po spuštění Spectry jsou provedeny následující kroky.

1. Hlavní logika nástroje Spectra zkontroluje a zpracuje vstupní parametry. Pokud jsou v pořádku, je vykonávání předáno třídě `monitorCreator`.
2. Objekt třídy `monitorCreator` nejprve nechá předanou specifikaci zpracovat třídou `formulaAnalyser`.
3. `FormulaAnalyser` po znacích načte specifikační soubor a jeho obsah zpracuje do vektoru formulí, popisujících všechny temporální vlastnosti, které mají být následně vygenerovaným monitorem ověřovány. Zároveň extrahuje všechny ostatní doplňující údaje, které mohou být ve specifikaci uvedeny, jako definice výpočtů stavových proměnných a seznam zdrojových kódů. Po dosažení konce souboru se specifikací je vykonávání vráceno zpět objektu třídy `monitorCreator`.
4. Pokud `formulaAnalyser` načetl neprázdný seznam zdrojových kódu ověřovaného programu, kde se mohou vyskytovat komentáře pro instrumentaci, nyní `monitorCreator` provede nahrazení těchto komentářů voláním funkce `monitorVerify`, přičemž původní verze zdrojových souborů zachová vedle těch nových, jak bylo popsáno v podkapitole 5.2. Pokud tyto zdrojové soubory neobsahují odkaz na generovaný hlavičkový soubor monitoru, je automaticky doplněn i ten.
5. Je vygenerován hlavičkový soubor `monitor.h`.
6. Postupně jsou sestavovány jednotlivé části implementace monitoru popsané v předchozí podkapitole na základě informací extrahovaných ze specifikačního souboru třídou `formulaAnalyser`. Hotový kód monitoru je uložen do souboru `monitor.c` a vykonávání se vrací do hlavní logiky nástroje Spectra.
7. Je provedena kompilace vygenerovaného monitoru do objektového souboru, který je následně zabalen do statické knihovny `libspectra.a`.
8. Nyní jsou již nepotřebné soubory `monitor.c` a `monitor.o` smazány a vykonávání programu nástroje Spectra se ukončí.

V souborovém umístění odkud byl nástroj Spectra spuštěn se tedy po jeho vykonání nachází dva nové soubory: `monitor.h` a `libspectra.a`. Ty je nyní možné vhodně umístit do adresáře s programem, který má být vygenerovaným monitorem ověřován, a začít testovat.

Kapitola 6

Implementační detaily nástroje T-Props Checker

Při vývoji nástroje byl kladen důraz především na snadnou rozšiřitelnost a pochopitelnost kódu, dokumentace i automatizovaných testů. Například, z důvodu lepší čitelnosti zápisu kódu generování monitoru třídou `monitorCreator`, bylo původní použití třídy `ofstream` z jazyka C++ nahrazeno voláními funkce `printf` jazyka C. Většinu částí kódu monitoru tak bylo v implementaci nástroje možné definovat jako konstantní řetězec s komentovanými parametry. Tím je maximalizován podíl celistvých kusů šablon generovaného kódu, které jsou jednodušší pro orientaci a jejichž úprava je jednodušší, než kdyby bylo nutné místo pro úpravu hledat mezi krátkými úseky téměř nic neříkajících řetězců, střídanými názvy proměnných a nespočetným množstvím C++ operátoru `<<`.

Obecně vzato nejsou jazyky C ani C++ pro generování obsahu souboru podle šablony příliš vhodné, zvláště když je nutné tuto šablonu upravovat. Mnohem lépe by zde posloužil nějaký skriptovací jazyk, což by ovšem znamenalo, že nebude možné využívat základní funkcionalitu nástroje na stroji, který další programovací jazyky nepodporuje.

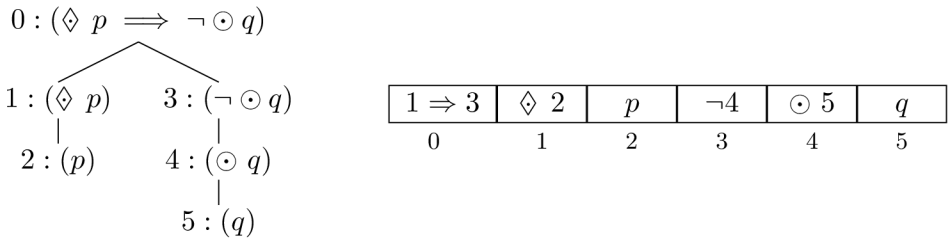
Do obou generovaných souborů – `monitor.c` i `monitor.h` – je na začátku vložena hlavička s komentářem, popisujícím kdy a jakým způsobem byl tento soubor vygenerován. Je zde tedy uveden přímo příkaz s jakým byl nástroj Spectra spuštěn a datum a čas vzniku daného souboru.

Všechny zdrojové kódy nástroje obsahují komentáře formátované pro zpracování nástrojem *Doxygen*[1]. Následující sekce se zabývají konkrétním řešením některých implementačních otázek, jako je efektivita ukládání a zpracování formulí specifikace, nebo práce se stavovými proměnnými v rámci generovaného monitoru.

6.1 Reprezentace stromu formulí

Jak bylo popisováno již v kapitole 4.2, formule specifikace jsou při analýze rozloženy do stromu podformulí. Z hlediska výpočtů následně generovaného monitoru ani samotné analýzy však není nutné tuto stromovou strukturu explicitně udržovat. Místo toho jsou jednotlivé formule, reprezentované třídou `formula`, která byla popsána v kapitole 5.3, ukládány do vektoru v pořadí, v němž se jejich začátky vyskytují v souboru se specifikací. Jejich zpracovávání probíhá za pomoci zásobníku a jako odkazy na operandy slouží indexy do vektoru, do něhož jsou již zpracované formule postupně ukládány.

[1] viz `http://www.doxygen.nl/`



Obrázek 6.1: Příklad rozkladu temporální formule do stromu podformulí a jeho odpovídající symbolická reprezentace v poli.

Způsob převodu stromu formulí do vektoru je inspirován Havelundem a Roşu [9] a ilustruje ho obrázek 6.1. Na tom je také vidět, že otcovská formule ve vektoru vždy předchází všechny své synovské formule. Uložení hodnot platnosti formulí ve vygenerovaném monitoru kopíruje toto uspořádání v poli hodnot typu `bool`. Díky tomu pak při vyhodnocování platnosti jednotlivých podformulí není nutné složitě procházet stromovou strukturu, stačí pouze hodnoty pole určovat od konce směrem k počátku a bude zajištěno, že hodnoty všech synovských uzlů budou dopočítány předtím, než je bude otcovský uzel potřebovat k výpočtu vlastní hodnoty.

6.2 Správa stavových proměnných

Generovaný monitor má za běhu sledovaného programu ověřovat temporální vlastnosti specifikované předanými formulemi, které jsou přímo zapsány do kódu monitoru. K tomu potřebuje (i) přistupovat přímo k hodnotám deklarovaných proměnných zkoumaného programu a (ii) dopočítávat stavové proměnné formulí z proměnných programu podle předpisů daných ve specifikaci. Vzhledem k tomu, že v době zpracovávání formulí ještě není nástroji T-Props Checker známo, která proměnná se vyskytuje v přikládaném hlavičkovém souboru, a která bude dodefinována v souboru specifikačním, je potřeba, aby mohl monitor s oběma typy proměnných zacházet stejným způsobem.

Stavové proměnné jsou proto v monitoru uloženy v poli a jejich hodnoty jsou reprezentovány odkazem na funkci s návratovou hodnotou `bool`. Všechny tyto funkce jsou generovány do kódu monitoru jako samostatná funkce s názvem `f`k před deklarací struktury monitoru, kde k je celé číslo z $\langle 0, K \rangle$, K je počet všech stavových proměnných v analyzovaných formulích. Tyto funkce vrací buďto přímo hodnotu proměnné deklarované v hlavičkovém souboru, nebo výsledek odpovídajícího výpočtu uvedeného pro danou proměnnou ve specifikaci.

K adresování tohoto pole je používán výčet `enum VarNames`, který je vygenerován ze seznamu všech jmen stavových proměnných, které byly načteny při analýze formulí. Aby nedocházelo ke konfliktům mezi deklarovanými proměnnými z hlavičkového souboru a jmény ve výčtu, je k výčtovým názvům proměnných přidávána předpona `_e_`. Příkaz k přečtení hodnoty stavové proměnné p z obrázku 6.1 pro vyhodnocení aktuální hodnoty odpovídající formule s indexem 2 pak vypadá následovně.

```
monitor.now[2] = monitor.variables[_e_p]();
```

Kapitola 7

Evaluace implementovaného nástroje

Správná funkčnost implementovaného nástroje byla ověřována na dvou úrovních – jednotkovými a systémovými testy. Veškeré testy mají za úkol dokázat, že (i) implementované funkce se chovají přesně tak, jak je definováno v jejich popisu, a že (ii) nástroj dokáže korektně zpracovat všechny operátory, jejichž použití specifikace umožňuje. Pokud totiž nástroj funguje pro libovolný operátor, bude fungovat i pro jakkoli komplexní specifikaci, dokud bude používat pouze tyto operátory.

Mezi těmito testy pro vyhodnocení správného fungování implementace jsou vloženy také testy ověřující robustnost vůči chybám formátování specifikace. Kromě samotných testů byla implementace také vyzkoušena s reálným kódem pro ovládání výtahu, poskytnutým ze společnosti Honeywell jak bylo zmiňováno v kapitole 3.2.

7.1 Jednotkové testy

K vytvoření a spouštění jednotkových testů pro implementovaný nástroj je použit framework *Google Test*[1], umožňující implementovat velmi snadno rozšiřitelné sady testů s integrovanou přípravou prostředí pro jednotlivé testy i jeho úklidem, jak po každém testu, tak i po celé sadě. Jednotkové testy byly vytvořeny pro třídy `formulaAnalyser` a `monitorCreator` popsané v kapitole 5.2, ostatní třídy implementace nástroje T-Props Checker obsahují téměř výlučně pouze jednoduché funkce `get` a `set`, jejichž správnost není nutné zvlášť prověřovat.

Testy pro třídu `formulaAnalyser` jsou rozděleny do dvou sad.

1. Ověřující funkci jednotlivých metod při všech přípustných kontextech, z nichž mohou být volány.

2. Testující korektní reakce zpracování na různé varianty zápisu specifikace.

V první sadě jsou volány všechny dílčí metody třídy `formulaAnalyser` s různými obsahy zásobníku pro zpracovávané formule a různým stavem formule na vrcholu tohoto zásobníku. Po provedení dané metody je prověřován znovu jak obsah zásobníku, tak i zpracování formule na jejím vrcholu. Druhá sada pak volá hlavní metodu funkce `formulaAnalyser` nad

[1] viz https://github.com/google/googletest

různými soubory se specifikací, kde se kontrolují jak typické, tak hraniční případy formátování specifikace, které nástroj je, nebo není ochoten akceptovat. Součástí této sady jsou jak formátovací varianty zápisu formulí, tak sémantické chyby např. v počtu operandů pro jednotlivé operátory.

Jednotkové testy pro třídu `monitorCreator` jsou pak implementovány pro tu část jejích metod, jejichž účelem není čistě vyplnit část šablony pro kód monitoru, která by byla závislá především na počtu a vyjádření zpracovaných formulí. Takové metody nejsou testovány jednotlivě, nýbrž společně v rámci testů na celkové zpracování specifikace do kódu generovaného monitoru.

Testy na celkové zpracování specifikace probíhají nad jednoduchými specifikačními soubory, které obsahují vždy právě jednu formuli s jediným operátorem a které byly vytvořeny zvlášť pro každý podporovaný operátor. Ve vygenerovaném kódu monitoru je poté vždy ověřena správnost výrazů pro inicializaci a výpočet hodnoty platnosti dané formule při verifikaci, určených podle sémantiky zpracovávaného operátoru.

Všechny jednotkové testy jsou používány společně s nástrojem *LCOV*, který sleduje a ilustruje dosažené pokrytí zdrojových kódů. U všech součástí nástroje T-Props Checker bylo dosaženo 100% pokrytí řádků kódu i volání funkcí.

7.2 Systémové testy

Kromě jednotkových testů pracujících s částmi implementovaného a generovaného kódu byly vytvořeny také systémové testy kontrolující správné ověřování sémantiky používaných operátorů. Ty jsou spouštěny nad jednoduchým programem v jazyce C, který počítá od 0 do 10, kterému jsou postupně přiřazovány specifikace založené na definicích sémantiky jednotlivých operátorů. Např. pro operátor $\uparrow$ je ve specifikaci uvedena formule

$$\uparrow p \implies (p \wedge \odot\neg p) \wedge (p \wedge \odot\neg p) \implies \uparrow p, \tag{7.1}$$

nebo pro operátor $\Diamond$ formule

$$\Diamond\ p \implies \neg \boxdot \neg p\ \wedge \neg \boxdot \neg p \implies \Diamond\ p. \tag{7.2}$$

Pro tyto formule je p vždy definováno např. tak, že se čítač testovacího programu rovná nějaké konkrétní hodnotě, popř. je větší nebo menší než nějaká hodnota z intervalu $\langle 0, 10\rangle$.

Tyto testy díky své jednoduchosti mohou pomoci uživatelům nástroje T-Props Checker pochopit sémantiku používaných operátorů. Jsou spouštěny pomocí `bash` skriptu, ve kterém je tím pádem také uveden vzorový příklad, jak kompilovat vlastní projekt společně s monitorem, který byl vygenerován implementovaným nástrojem.

7.3 Zhodnocení reálné aplikace implementovaného nástroje

Jak bylo zmíněno v kapitole 3.2, společnost Honeywell reálně používá LTL logiku ke specifikaci vlastností jimi vyvíjeného softwaru [1], což zjednodušuje jeho následnou validaci a verifikaci. Tato specifikace může být vytvořena s asistencí nástroje *ForReq*, který slouží k transformaci požadavků, formulovaných speciálně formátovaným přirozeným jazykem, do formulí LTL, které používají pouze operátory *next* a *always*.

V této podkapitole bude demonstrováno a zhodnoceno použití implementovaného nástroje T-Props Checker k ověření temporálních vlastností kódu poskytnutého jako příklad

firmou Honeywell, spolu se specifikací vygenerovanou pomocí nástroje *ForReq*. Jedná se o kód vestavěného systému určeného k ovládání výtahu, který byl vygenerován nástrojem *Simulink*[2].

Právě kvůli tomu, že byl daný kód automaticky vygenerován, navíc pro vestavěný systém, nebyl v původní podobě dostatečně snadno čitelný a obsahoval mnoho proměnných, které mimo vestavěnou platformu nejsou nutně potřebné. Proto byl pro účely této práce přepsán do zjednodušené formy se zachováním veškeré funkcionality, která je na základě přiložené specifikace kontrolována. Provedená zjednodušení se týkala především sloučení příznakových proměnných, jejichž hodnoty vedou ke stejným tokům vyhodnocování, nebo úplného vypuštění proměnných indikujících stav provádění, které ovšem nejsou použity v logice programu, ani v přiložené specifikaci. Tato úprava poskytnutého kódu nebyla nutná pro použití nástroje T-Props Checker, ale spíše vhodná pro poskytnutí jednoduše srozumitelného příkladu budoucím uživatelům tohoto nástroje.

7.3.1 Popis ověřovaného programu

Vykonávání původního kódu výtahu probíhá po inicializaci v potenciálně nekonečném cyklu, ve kterém je vždy proveden jeden krok logiky výtahu, implementovaný ve funkci `step`. V tom je v závislosti na aktuální činnosti výtahu provedena jedna elementární akce následujícím způsobem.

1. Pokud se výtah pohybuje, pak:
   a) v případě, že se nachází nad patrem, do kterého byl zavolán, popř. poslán, posune se o patro níž,
   b) pokud je níže než aktuální cílová destinace, posune se o patro výš, nebo
   c) v případě, že se nachází v cílovém patře, zastaví.
2. Pokud výtah stojí, pak jsou rozlišovány tyto stavy:
   a) výtah zastavil v cílovém patře, otevřou se dveře,
   b) výtah stojí s otevřenými dveřmi, dveře jsou v tomto kroku zavřeny,
   c) výtah stojí se zavřenými dveřmi a čeká na zavolání, zkontroluje tedy, jestli nepřišel nový povel, a pokud ano, nastaví si novou cílovou pozici,
   d) výtah se ještě nerozjel, avšak už má nastavenou novou cílovou pozici, vyhodnotí tedy kterým směrem se má rozjet, nebo jestli „zastavil v cílovém patře“,
   e) výtah vyhodnotil, kterým směrem se má pohybovat a připraví se na uvedení do pohybu v dalším kroku,
   f) výtah se po potřebných přípravách, které následovaly po přijetí pokynu, dá do pohybu[3].

Z předchozího vyplývá, že pokud je výtah nachystaný na přijetí nového pokynu, trvá mu 4 kroky vyhodnocování, než se uvede do pohybu. Pro reálný vestavěný systém řídící mechanický stroj je to smysluplné, avšak pro účely demonstrace v rámci této práce by dávalo smysl kroky 2.d)–f) sloučit do jednoho. Potom by ovšem bylo nutné adekvátně upravit

[2] viz https://www.mathworks.com/products/simulink.html

[3] Skutečný posun o patro správným směrem je však proveden až v následujícím kroku

i specifikaci, která s touto prodlevou počítá, proto byla tato logika ponechána v autentické podobě.

Daný kód výtahu je určen pro simulaci a její vyhodnocování s frekvencí kroků výtahu 1 Hz. Výtah je řízen dvěma vstupními proměnnými: (i) booleovskou hodnotou `liftCalled` a (ii) číslem požadovaného patra `wantedDestination`. Jejich hodnoty jsou simulovány náhodnými čísly jako

```
liftCalled = rand() % 2;
if (previousDestinationReached)
    wantedDestination = rand() % MAX_FLOOR;
```

Nové cílové patro je generováno pouze pokud bylo dosaženo předchozí cílové pozice, aby na jejím základě bylo možné ověřovat správné nastavení v logice výtahu. Pokud by byly hodnoty generovány i mezi ostatními kroky výtahu, neovlivnily by nijak vykonávání programu, protože dokud výtah nedorazí do cílové stanice, je přijímání nových pokynů blokováno.

V původní verzi kódu simulace práce výtahu dokonce hodnota `wantedDestination` není modifikována vůbec a všechny specifikované vlastnosti jsou tedy zkontrolovány na jednom přesunu výtahu a jeho následném setrvání v daném cílovém patře. Pro účely demonstrace v rámci této práce však byla simulace rozšířena o tyto změny cílového patra a navíc byly přidány také výpisy aktuální a cílové pozice. Počet cyklů vykonaných v rámci simulace byl omezen na 100.

7.3.2 Princip zpracování specifikace požadavků

Jak bylo popisováno dříve, ke kódu simulace výtahu byla poskytnuta také specifikace temporálních vlastností, které by měl tento program splňovat. Ta je vygenerována nástrojem *ForReq* a používá označení `G(p)`[4] jako $\Box\, p$ a `X(p)` místo $\bigcirc\, p$. Uvedené formule bylo nutné převést do ptLTL, kterou nástroj T-Props Checker bude umět zpracovat.

Nejdůležitější požadovanou temporální vlastností výtahu uvedenou ve specifikaci je, že pokud je obecně řídící signál `liftCalled` nastaven po dobu alespoň 2 s, výtah by měl dorazit do cílového patra nejdéle za dalších $n + 5$ s, kde n je počet pater obsluhovaných výtahem a 5 s odpovídá potřebné časové rezervě pro zpracování příkazu popisované v předchozí podkapitole. Obdobně jsou pak definovány konkrétnější vlastnosti, a to že pokud je řídící signál `liftCalled` nastaven po dobu alespoň 2 s a rozdíl mezi aktuální a cílovou pozicí výtahu je roven k, výtah by měl dorazit do cílového patra nejdéle za dalších $k + 5$ s. Např. pro $k = 1$ tedy specifikace obsahuje formuli

```
G(((lc && X(lc && X(lc))) && X(X(diff_current_wanted_is_1))) ->
  X(X(
    arrived || X(arrived || X(arrived ||
    X(arrived || X(arrived || X(arrived)))))
  )))
```

kde `lc` odpovídá signálu `liftCall`, `diff_current_wanted_is_1` indikuje, že absolutní hodnota rozdílu mezi aktuální a cílovou pozicí výtahu je rovna 1, a `arrived` je stavová proměnná nastavovaná v okamžiku, kdy výtah zastavil v požadovaném patře.

[4] V literatuře se pro operátor, který je v této práci označován jako *always*, běžně používá název *globally*. Ten však je z těchto dvou názvů pro duální operátor z ptLTL o něco vhodnější a z důvodu přehlednosti není vhodné, aby byly oba nazývány stejně.

Ve specifikaci jsou popsány i další požadované vlastnosti výtahu, jako časový limit pro uvedení výtahu do pohybu, nebo vyhodnocení správného směru pohybu po přijetí pokynu, popř. souvislosti mezi pohybem a aktivací brzd, nebo dosažením cílové pozice a otevíráním dveří. Formule, které je specifikují jsou však syntakticky jednodušší, proto bude pro ukázku převodu podobných formulí do požadované ptLTL použita formule uvedená výše.

Prvním krokem převodu formule LTL omezené na operátory $\Box\, p$ a $\bigcirc\, p$ do ptLTL je určit maximální počet po sobě jdoucích stavů, ke kterým je formule vztažena, daný maximálním rozdílem úrovní zanoření do formule s operátorem $\bigcirc$. Předpoklad implikace v uvedené formuli se týká tří stavů – aktuálního a dvou následujících. Důsledek této implikace je potom vztažen k přespříštímu stavu a dalším 5ti. Maximální počet popisovaných po sobě jdoucích stavů pro tuto formuli je tedy 7.

PtLTL se na rozdíl od LTL nevyhodnocuje nad stavy budoucími, ale minulými, proto je druhým krokem převodu LTL formulí do ptLTL obrácení zanoření proměnných. Formule $\odot p$ je ve specifikaci pro T-Props Checker zapisována jako `(L p)` a důsledek implikace je tedy převeden na

```
(L(L(L(L(L arrived) || arrived) || arrived) ||
arrived) || arrived) || arrived
```

což sémanticky přesně odpovídá původnímu

```
arrived || X(arrived || X(arrived ||
X(arrived || X(arrived || X(arrived)))))
```

Konec časového intervalu, kterého se týká předpoklad této implikace, se nachází o 5 stavů dříve, než důsledek, jehož časový interval končí stejným stavem, jako interval celé formule. Samotný obsah předpokladu lze přepsat jako

```
((L(L lc) && lc) && lc) && diff_current_wanted_is_1
```

avšak právě kvůli posunutí koncového bodu intervalu této podformule oproti kořenové formuli musí být ještě posunuta o 5 stavů do minulosti:

```
(L(L(L(L(L ((L(L lc) && lc) && lc) && diff_current_wanted_is_1)))))
```

V tuto chvíli je původně použitý operátor `G(p)` sémanticky ekvivalentní se svou duální ptLTL verzí. Posledním potřebným krokem úprav LTL formulí specifikace je tedy pouze převod do prefixové formy požadované nástrojem T-Props Checker. Vzhledem k tomu, že je sémantika booleovských operátorů vůči zanoření do temporálních formulí invariantní, je pro přehlednost formulí volitelně možné minimalizovat zanořování formulí s booleovskými operátory do temporálních podformulí. Výsledkem demonstrovaného převodu pak je

```
(G (->
  (L(L(L(L(L(&&
    (L (L lc))
    (L lc)
    lc
```

```
    diff_current_wanted_is_1
  ))))))
  (|| arrived (L arrived ) (L(L arrived )) (L(L(L arrived )))
     (L(L(L(L arrived )))) (L(L(L(L(L arrived ))))) )
))
```

Po převedení všech formulí do formy požadované nástrojem T-Props Checker je ještě potřeba uvést definice výpočtů stavových proměnných, např.

```
DEFINITIONS:
lc : liftCalled
...
```

popř. do specifikačního souboru doplnit cestu ke zdrojovému kódu, aby mohla být instrumentace zajištěna z větší části automaticky.

7.3.3 Použití nástroje T-Props Checker s kódem simulace řízení výtahu

Jedinou opravdu potřebnou úpravou zdrojových kódů, nutnou pro použití implementovaného nástroje, je zajištění instrumentace generovaného monitoru do ověřovaného programu. Toho lze jednoduše dosáhnout doplněním komentářů podle kapitoly 5.1 na vhodná místa v kódu. Takovým je intuitivně konec smyčky, ve které provádění výtahu po inicializaci běží.

Doplněním tohoto jednoho řádku je po provedené úpravě specifikace příklad připraven na použití s nástrojem T-Props Checker. Je tedy možné z příkazové řádky ve složce obsahující všechny potřebné soubory spustit

```
$ ./spectra --tpc -s lift.spec -d lift.h
```

kde soubor `lift.spec` obsahuje zpracovanou specifikaci a jmenovaný hlavičkový soubor poskytuje definice potřebných datových struktur a deklarace stavových proměnných používaných ve specifikaci ve tvaru

```
extern <type> <name>;
```

Nástrojem Spectra jsou vytvořeny soubory `monitor.h` a `libspectra.a`. Vložený řádek s instrumentačním komentářem je ve zdrojovém kódu výtahu, uloženém v souboru `lift.cpp`, nahrazen voláním funkce `monitorVerify` popsané v kapitole 5.3. Navíc je za poslední řádek s `#include` vloženo

```
extern "C" {
#include "monitor.h"
}
```

Pokud by kód výtahu byl formulován čistě v jazyce C a uložen jako `lift.c`, nebylo by obalení do `extern "C"{ ... }` použito. Jedná se o informaci pro kompilátor jazyka C++, který takto ví, že odkazovaný kód je kompilován jako program v jazyce C.

Nyní je možné ověřovaný program spolu s monitorem zkompilovat příkazem[5]

```
CXX CXXFLAGS -L. -I. SUT/lift.cpp -lspectra -o lift
```

a spustit výsledný program `lift`. Ten provede simulaci 100 vyhodnocovacích cyklů a ukončí se bez jediného projevu zapojení monitoru, protože je naimplentován tak, že splňuje všechny požadované vlastnosti.

7.3.4 Zhodnocení použití nástroje T-Props Checker

Protože je logika provádění ověřovaného programu poměrně jednoduchá a vyhodnocování prováděné monitorem je voláno v poměru k relativnímu objemu ostatního prováděného kódu poměrně často, dá se očekávat určité prodloužení doby výpočtu ověřovaného programu, do něhož je program zapojen. Pro tento účel byla naměřena doba provedení simulace výtahu 10 000[6] krát za sebou jak s, tak bez volání vygenerovaného monitoru. Naměřené zpomalení dosahuje necelých 2 %, což je vzhledem k malému rozsahu kódu ověřovaného programu a relativní četnosti volání monitoru velmi pozitivní výsledek.

Závěrečným experimentem s programem simulace výtahu bylo vnášení chyb do jeho implementace s cílem zjistit, jestli je monitor odhalí. Pro každou formuli specifikace byla postupně vyzkoušena chyba, která vede k jejímu porušení, a každou z nich monitor při první příležitosti ohlásil. Např. pro formuli, na které byl v podkapitole 7.3.2 demonstrován postup pro zpracování původní specifikace ověřovaného programu, byla vyměněna prováděná úprava aktuální pozice výtahu, takže pokud jel nahoru, od aktuální pozice odčítal (maximálně do 0), a při pohybu dolů zase aktuální pozici inkrementoval. Aby byl dodržen konkrétní rozdíl cílového a aktuálního patra o 1, byl pro tento experiment snížen počet pomyslných pater obsluhovaných výtahem na 2. Po prvním splnění předpokladu implikace dané formule monitor na chybový výstup vytiskl

```
Verification failed after round #21!
Relevant changes:
specifications/lift.spec:11:5-10:
       (L (L (L (L (L (&& (L (L (lc))) (L (lc)) (lc) (diff_current...
       turned true after being false for last 21 states

specifications/lift.spec:10:4-1:
       (-> (L (L (L (L (L (&& (L (L (lc))) (L (lc)) (lc) (diff_cur...
       turned false after being true for last 21 states

specifications/lift.spec:10:1-2:
       (G (-> (L (L (L (L (L (&& (L (L (lc))) (L (lc)) (lc) (diff_...
       turned false after being true for last 21 states
```

odkud lze pochopit, že ve stavech 14.–16. se poprvé vyskytla sekvence 3 stavů, kdy je řídící proměnná `liftCall` po celou dobu nastavena na `true`, společně rozdílem mezi cílovou a aktuální pozicí rovným 1 v posledním z těchto tří stavů. O 7 stavů později je detekována chyba, protože nebylo dosaženo požadovaného důsledku v toleranci $k+5$ stavů (zde $k = 1$).

[5] Pro přesný návod jak tento příkaz sestavit viz příloha B.

[6] Experimentálně určeno, jako hodnota, kdy se při opakování tohoto měření výsledek téměř nezmění.

Implementovaný nástroj je tedy schopen vygenerovat monitor, který dokáže nad poskytnutým programem spolehlivě monitorovat všechny vlastnosti uvedené v přiložené specifikaci, a to v tomto případě s nutností přidání pouze jediného komentáře do původního kódu. Jinou kapitolou je v tomto směru potřebná úprava specifikace, která ačkoli je její princip snadno pochopitelný, její provedení není úplně triviální. Avšak vzhledem k možné další spolupráci s firmou Honeywell není vyloučeno, že by v budoucnu mohlo existovat rozšíření nástroje *ForReq*, popř. samostatný nástroj na podobném principu, jehož výstupem by místo formulí LTL byla požadovaná forma ptLTL. Z hlediska vnitřní logiky nástroje *ForReq* by se totiž jednalo pouze o nepříliš složitou úpravu formátování výstupu.

Kapitola 8

Napojení nástroje na platformu Testos

Již v kapitole 1 bylo uvedeno, že implementovaný nástroj má být snadno použitelný jak samostatně, tak společně s ostatními nástroji platformy Testos. Aby tedy byla implementovaná funkcionalita jednoduše zpřístupněna k použití z jiných programů, bylo pro ni vytvořeno webové REST API implementované v jazyce Python s využitím frameworků *Flask* a *Jinja*[1].

Implementovaná webová aplikace je koncipována tak, aby k ní bylo možné jednoduše přistupovat jak manuálně, pomocí uživatelského rozhraní z webového prohlížeče, tak především programově, pomocí HTTP požadavků. Po instalaci Pythonu a frameworku Flask je možné ji spustit buďto lokálně, nebo např. v *Dockeru*[2], příkazem

```
$ flask run
```

volaným z adresáře[3] s kódem této aplikace. Ta umožňuje správu testování projektů nástrojem T-Props Checker. Do aplikace je možné nahrávat soubory a spravovat více oddělených projektů, nad nimiž lze definovat testovací případy, spouštět je nad monitorovaným programem a získávat výsledky verifikace. Seznam a popis všech použitelných HTTP požadavků pro tuto aplikaci je uveden v příloze D.

8.1 Správa projektů

Aplikace umožňuje vytvářet a spravovat projekty, jejichž součástí jsou

a) zdrojové a jiné soubory,

b) příkaz ke kompilaci projektu,

c) identifikace, které ze souborů projektu mají být nástroji T-Props Checker pomocí parametrů při testování nastavovány jako specifikace a hlavičkový soubor s deklaracemi,

d) definice testovacích případů nad daným projektem a

[1] viz http://flask.pocoo.org/ a http://jinja.pocoo.org/
[2] viz https://docker-curriculum.com/
[3] V rámci této práce se jedná o adresář `api`.

e) spouštění těchto testovacích případů a čtení výsledků.

Každý projekt má svou vlastní složku, pod kterou sou umístěny všechny soubory tohoto projektu a která obsahuje speciální soubor `status.json`. Jeho podrobná struktura je popsána v příloze C a slouží k uchovávání informace z bodů b)–d). Projekt lze vytvořit buďto pouze jako čistou složku s daným názvem, do které potom budou všechny ostatní součásti nahrávány zvlášť, nebo je možné založit kompletně inicializovaný projekt. Požadavek k jeho vytvoření totiž může volitelně obsahovat soubory (i) se specifikací a (ii) s definicemi, příkaz k sestavení projektu a archiv typu `zip`, který bude do složky projektu rozbalen. Testovací případy pak lze při dodržení předepsaného obsahu specifikovat v souboru `status.json`, který bude přibalen do archivu se soubory projektu.

Pokud není v požadavku explicitně specifikováno jméno projektu, musí být přítomen archiv se soubory, jehož název se použije i jako jméno pro vytvářený projekt. Pokud je přiložen jak archiv, tak soubory se specifikací a/nebo hlavičkový soubor pro generovaný monitor, pak v případě, že jsou tyto soubory zahrnuty i v archivu, slouží jejich nahrání i touto cestou pouze k jejich označení jako vstupních souborů pro T-Props Checker.

Kromě názvu lze všechny ostatní údaje projektu možné později změnit, avšak se soubory lze později manipulovat pouze po jednom. Soubory, včetně nových složek, je možné vkládat, odstraňovat, nebo získat jejich obsah. Je možné kdykoli nahrát nový soubor označený jako specifikace nebo hlavičkový soubor pro T-Props Checker a změnit příkaz ke kompilaci daného projektu. Dokud projekt neobsahuje označení obou těchto souborů a příkaz ke kompilaci, není možné spouštět testovací případy. Připravenost projektu na spouštění testů lze ověřit samostatným požadavkem, který vrátí nejen odpověď, zda je projekt připraven, ale případně také seznam chybějících náležitostí, které pro spouštění testů nutné doplnit.

8.2 Správa testovacích případů

Každému projektu je možné přiřadit neomezené množství testovacích případů jejichž údaje obsahují:

a) číselný identifikátor testovacího případu,

b) příkaz ke spuštění testovacího případu,

c) seznam číselných identifikátorů spuštění, které prováděly tento testovací případ, dále označovány jako testy,

d) čas posledního spuštění testu,

e) výsledek verifikace posledního spuštění testu,

f) soubor, který má být testům připojen na vstup,

g) čas maximální doby běhu testů, po kterém budou případně násilně ukončeny a

h) indikátor, zda aktuálně běží nějaký test nad tímto testovacím případem, nebo ne.

Údaje a) a b) jsou povinné a údaje b), f) a g) lze kdykoli bezpečně aktualizovat, popř. je možné testovací případy odstraňovat a po jednom zakládat nové. Je možné si také vyžádat seznam všech testovacích případů daného projektu, ten modifikovat a zaslat zpět, čímž bude původní seznam přepsán novým. Toto je však pokročilý způsob úprav, jehož použití

nelze při seznamování se s touto aplikací doporučit. Aktuální údaje konkrétního testovacího případu lze získat požadavkem jako JSON objekt.

Pokud k testovacímu případu není připojen vstupní soubor, ale testovaný projekt ze vstupu potřebuje číst, pak bude případné vykonávání daného testovacího případu neúspěšně ukončeno po zadaném, nebo výchozím časovém limitu, který je stanoven na 5 min. Pokud je projekt plně inicializován, lze požadavkem spustit nové vykonávání libovolného testovacího případu určeného svým číselným identifikátorem.

8.3 Spouštění testů

Pokud aplikace zaznamená požadavek na spuštění nějakého testovacího případu, zařadí ho do fronty mezi ostatní požadavky na provádění testů. Tuto frontu obsluhuje vlákno, určené výhradně k vykonávání testovacích případů. Jeho synchronizace s hlavním vláknem aplikace, které obsluhuje HTTP požadavky, je zajištěna pomocí semaforu indikujícího počet požadavků čekajících ve frontě na zpracování.

Pro každý prováděný testovací případ je nejdříve provedena kontrola, že je dostupná verze programu Spectra aktuální, popř. je proveden jeho nový překlad. Poté je s využitím nástroje T-Props Checker na základě parametrů nastavených pro daný projekt vygenerován monitor. Výsledný kód monitoru je následně při kompilaci projektu napojen na implementaci ověřovaného programu a nakonec je spuštěn příkaz samotného testovacího případu. Záznam sloučeného chybového a standardního výstupu z průběhu kompilace i testu jako takového je později možné zvlášť získat požadavkem.

Následné vyhodnocení úspěšnosti testovacího případu se nezabývá průběhem testovacího případu z hlediska uživatelského projektu, ale pouze z hlediska prováděné verifikace. Vykonávání testovacího případu tedy bude označeno za neúspěšné pokud (a) nebylo možné projekt sestavit, nebo (b) se na chybovém výstupu testovaného programu objeví chybové hlášení monitoru.

Při dokončení provádění testu jsou výsledky zapsány jak do aktuálního stavu prováděného testovacího případu, tak do stavu daného spuštění. To má vždy přiděleno číselný identifikátor, který je vracen v odpovědi na požadavek na spuštění testovacího případu. Čím nedávnější spuštění, tím vyšší identifikátor je mu přidělen. U testovacích případů je pak uveden seznam identifikátorů spuštění, která prováděly daný testovací případ, s jejichž pomocí lze získávat historické záznamy o provádění daného testovacího případu.

Kapitola 9

Závěr

Hlavním výsledkem této práce je návrh a implementace nástroje T-Props Checker, který slouží ke generování monitorů pro ověřování temporálních vlastností programů jazyka C a C++. Zkoumané vlastnosti jsou specifikovány pomocí lineární temporální logiky vztažené k historii sledovaného běhu programu, která byla vybrána pro své vhodné vlastnosti pro tento konkrétní účel. Základní myšlenkou vyhodnocujícího algoritmu vytvářených monitorů je rekurzivní vyjádření operátorů používaných ve specifikaci.

Popis vlastností, které mají být ověřovány, je nástroji předáván v souboru s přesně specifikovaným formátem. Ověřování temporálních vlastností monitorem probíhá za běhu zkoumaného programu, s využitím sdílení hodnot vybraných proměnných pomocí jejich deklarace v hlavičkovém souboru, který je dán monitoru k dispozici. Správné fungování nástroje je ověřeno jednotkovými i systémovými automatickými testy.

Tato práce vznikla v rámci platformy Testos, jejíž cílem je automatizace testování softwaru. Implementovaný nástroj T-Props Checker je vytvořen jako první subsystém nového nástroje Spectra této platformy, jehož základní logika byla také implementována v rámci této práce. Nástroj Spectra bude výhledově sbírkou nástrojů sloužících k transformaci specifikace programů do vhodných forem strukturovaných dat, kterou je v případě T-Props Checkeru vygenerovaný kód monitoru temporálních vlastností.

S ohledem na předpoklad, že do nástroje Spectra budou v blízké době doplňovány další subsystémy, byla implementace veškerých součástí této práce vytvářena s důrazem na její snadnou pochopitelnost a rozšiřitelnost. To platí i pro webové rozhraní tohoto nástroje, které bylo vytvořeno jako napojení nástroje Spectra na platformu Testos a umožňuje snadné použití tohoto nástroje i z jiných programů.

Na příkladu reálné aplikace implementovaného nástroje bylo zhodnoceno, že jeho použití pro ověřování vlastního projektu, ke kterému je dostupná specifikace v požadovaném formátu, je jednoduché a spolehlivé. Vytváření požadované specifikace může být poměrně komplikované, avšak ve firmě Honeywell existuje reálný nástroj, který tento proces značně zjednodušuje, a v budoucnu by potenciálně mohl sloužit i k jeho plné automatizaci.

Literatura

[1] Barnat, J.; Beran, J.; Brim, L.; aj.: Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. *Formal Methods for Industrial Critical Systems*, 2012: s. 78–92, ze série Lecture Notes in Computer Science, díl 7437.

[2] Barnat, J.; Brim, L.; Černá, I.; aj.: DiVinE – A Tool for Distributed Verification. In *Computer Aided Verification*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-37411-4, s. 278–281.

[3] Bartocci, E.; Falcone, Y.: *Lectures on Runtime Verification. Introductory and Advanced Topics*. Springer International Publishing AG, 2018, ISBN 978-3-319-75631-8, ze série Lecture Notes in Computer Science, díl 10457.

[4] Bartocci, E.; Falcone, Y.; Francalanza, A.; aj.: *Introduction to Runtime Verification*. Springer, 2018, ISBN 978-3-319-75632-5, 1–33 s., ze série Lecture Notes in Computer Science, díl 10457.

[5] Bhatt, D.; Madl, G.; Oglesby, D.; aj.: Towards Scalable Verification of Commercial Avionics Software. 2010, doi:10.2514/6.2010-3452.

[6] Courcoubetis, C.; Vardi, M.; Wolper, P.; aj.: Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, ročník 1, č. 2-3, 1992: str. 275–288.

[7] Finkbeiner, B.; Sipma, H.: Checking Finite Traces using Alternating Automata. *Formal Methods in System Design*, ročník 24, č. 2, 2004: s. 101–127.

[8] Fredrikson, M.; Platzer, A.: Lecture Notes on LTL Model Checking and Büchi Automata. online skripta, 2018.
URL https://pdfs.semanticscholar.org/6b5e/b26e1368c523bf2d23cf3df7f5a331deca65.pdf

[9] Havelund, K.; Roşu, G.: Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, ročník 6, č. 2, 2004: s. 158–173.

[10] Kesten, Y.; Pnueli, A.; Raviv, L.: *Algorithmic Verification of Linear Temporal Logic Specifications*. Springer, Heidelberg, 1998, 1–16 s., ze série Lecture Notes in Computer Science, díl 1443.

[11] Moller, F.; Struth, G.: *Modelling Computing Systems*. Springer-Verlag London, 2013, ISBN 978-1-84800-321-7, ze série Undergraduate Topics in Computer Science.

[12] Ouaknine, J.; Worrell, J.: Some Recent Results in Metric Temporal Logic. In *Formal Modeling and Analysis of Timed Systems*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ISBN 978-3-540-85778-5, s. 1–13.

[13] Pnueli, A.: The Temporal Logic of Programs. *Foundations of Computer Science*, září 1977: s. 46–57.

[14] Roşu, G.; Havelund, K.: Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, ročník 12, č. 2, 2005: str. 151–197.

[15] Skupina Testos: Domovská stránka projektu Testos. FIT VUT v Brně, 2017, [Online; navštíveno 3.1.2019].
URL http://testos.org

[16] Skupina Testos: Gitlab repozitář nástroje Spectra. FIT VUT v Brně, 2019, [Online; navštíveno 3.1.2019].
URL https://pajda.fit.vutbr.cz/testos/spectra

Příloha A

Obsah přiloženého paměťového média

```
/
├── doc/ zdrojové soubory této technické zprávy pro LaTeX
├── spectra/ kořenový adresář projektu Spectra, v němž je implementovaný
│            nástroj T-Props Checker integrován
│   ├── api/ implementace webového rozhraní, které lze odtud spouštět
│   │        příkazem flask run
│   ├── examples/T-Props Checker/ složka s materiály k příkladu
│   │                             poskytnutého firmou Honeywell
│   ├── lib/ importované zdrojové kódy frameworku Google Test
│   ├── tests/
│   │   ├── coverage/ report pokrytí zdrojových kódů jednotkovými testy ve
│   │   │             formě html vygenerovaného nástrojem LCOV
│   │   └── T-Props Checker/ skripty a programy ověřující funkčnost
│   │                        implementovaného nástroje spolu se soubory,
│   │                        které jsou k tomu zapotřebí
│   ├── src/ hlavní logika nástroje Spectra
│   │   └── T-Props Checker/ zdrojové soubory nástroje T-Props Checker
│   │       └── doc/ dokumentace zdrojových kódů ve formě html
│   │                vygenerovaného nástrojem Doxygen
│   ├── spectra spustitelný program nástroje Spectra
│   └── Makefile
```

Příloha B

Manuál

Implementovaný nástroj T-Props Checker je integrován v nástroji Spectra, přes který je spouštěn následujícím příkazem:

```
$ ./spectra --tpc -s cesta/ke/specifikaci -d cesta/k/definicim.h
```

Volba `--tpc` vybere použití nástroje T-Props Checker a parametr `-s` slouží k určení cesty ke vstupnímu souboru se specifikací temporálních vlastností ve formátu podle kapitoly 5.1 a `-d` pak udává hlavičkový soubor s definicemi stavových proměnných. K implementaci nástroje Spectra je přiložen soubor `Makefile` s definovanými příkazy `make`, `make clean` a `make test` se standardními významy.

Výstupem takto použitého programu nástroje Spectra jsou dva soubory `monitor.h` a `libspectra.a`. Pro připojení tohoto vygenerovaného monitoru k vlastnímu projektu je potřeba provést následující kroky.

1. Označit místa ve zdrojových souborech, kde by měl monitor provádět verifikaci. Jednodušší možností je vložení komentáře `// T-Props Checker verify` na vhodná místa před spuštěním nástroje Spectra a seznamu zdrojových kódů do specifikace. Alternativně je poté potřeba (i) do daných míst vložit volání funkce `monitorVerify()` a navíc (ii) přidat řádek s `#include "[cesta/k/]monitor.h"`, který je jinak nástrojem T-Props Checker doplněn automaticky. V případě použití s programem v jazyce C++ je tento řádek navíc potřeba obalit označením `extern "C"{ ... }`, aby kompilátor rozuměl obsahu `libspectra.a`, který je kompilován jako jazyk C.

2. Pro kompilaci ověřovaného programu je dále nutné rozšířit

   a) `C[XX]FLAGS` o `-I.`, což kompilátor informuje, aby v načítaných knihovnách hledal objektové soubory přímo v kořenovém adresáři,

   b) `LDFLAGS` o `-L[adresář/s/libspectra.a]`, typicky např. `-L.`, pokud je soubor `libspectra.a` umístěn ve složce, odkud je prováděna kompilace a

   c) `LDLIBS` o `-lspectra` pro připojení knihovny `libspectra.a` ke kompilovanému programu.

Přeložený program lze poté spouštět běžným způsobem a v případě, že monitor objeví chybu, vytiskne na chybový výstup hlášku informující o všech formulích, které v tomto stavu změnily hodnotu, a proto by potenciálně mohly souviset s nalezeným chybovým stavem.

```
Verification failed after round #N!
Relevant changes:
[
cesta/ke/specifikaci:radek(ky):sloupce:
       formule
       turned <bool> after being !<bool> for last K states
]*
```

`N` je celkový počet doposud ověřených stavů a `K` je počet stavů, ve kterých byla předtím hodnota dané formule stabilní. Chyby verifikace lze odchytit i v ověřovaném programu jako návratovou hodnotu funkce `monitorVerify()`, která je v případě chybového stavu rovna `false`. V případě, že je to žádoucí, je možně verifikaci prováděnou monitorem nově inicializovat funkcí `reset()`, která způsobí, že monitor zapomene veškerý dosavadní sledovaný průběh programu.

Příloha C

Popis entit používaných v aplikaci webového REST API

C.1 Spuštění testovacího případu

Záznamy o každém spuštění libovolného testovacího případu jsou vždy uloženy v podadresáři `test-logs` adresáře příslušného projektu. Hlavní informace jsou uvedeny v souboru `teid.json`, kde `teid` je číselný identifikátor daného spuštění, a jehož obsah formátovaný jako JSON.

```
{
    "run": string,
    "passed": bool,
    "test-case": int
}
```

`Run` udává datum spuštění daného testu ve formátu `mm/dd/YYY, HH:MM`. `Passed` vyjadřuje, jestli tento test prošel verifikací prováděnou nástrojem T-Props Checker a `test-case` pak udává identifikátor testovacího případu, který byl tímto testem spuštěn. Ke každému spuštění pak existují další dva záznamy: (i) `teid.log` s výstupem provedeného testu a (ii) `compile_teid.log` s výstupem překladu projektu provedeného před zahájením vykonávání daného testu.

C.2 Testovací případ

Testovací případy jsou definovány pro daný projekt jako seznam objektů s následujícími atributy, v souboru `status.json` v kořenovém adresáři projektu.

```
{
    "cmd" : string
    "executions" : [ teid ]
    "id" : int
    "input" : string
    "last-run" : string
```

```
    "running" : bool
    "passed" : bool
    "timeout": int
}
```

`Cmd` je příkaz ke spuštění testovacího případu, a `id` je jeho jednoznačný číselný identifikátor. Seznam `executions` pak udává identifikátory dosavadních spuštění testů, které prováděly tento testovací případ. `Input` je název souboru, který má být testovacímu případu připojován na vstup a `timeout` udává počet sekund, po kterém bude vykonávání testovacího případu ukončeno. `Last-run` obsahuje datum posledního spuštění daného testovacího případu ve formátu `mm/dd/YYY, HH:MM`, `passed` sděluje výsledek posledního testu a `running` říká, jestli tento test při posledním spuštění prošel verifikací.

C.3 Projekt

Potřebné údaje ke každému projektu jsou definovány v souboru `status.json` v kořenovém adresáři projektu. Formát obsahu souboru `status.json` odpovídá následujícímu JSON objektu.

```
{
    "compile-cmd" : string
    "definitions" : {
        "path" : string
        "uploaded" : string
    }
    "missing" : [ string ]
    "name" : string
    "ready" : bool
    "specification" : {
        "path" : string
        "uploaded" : string
    }
    "test-cases" : [ test-case ]
}
```

`Compile-cmd` určuje příkaz ke kompilaci projektu a `name` udává jeho název. Soubory pro nástroj T-Props Checker určují atributy `definitions`, s umístěním a datem aktualizace hlavičkového souboru se stavovými proměnnými ve formátu `mm/dd/YYY, HH:MM`, a `specification`, s těmi samými atributy pro soubor se specifikací temporálních vlastností . Booleovská hodnota `ready` udává připravenost projektu na spouštění testů a seznam `missing` obsahuje řetězce popisující případné chybějící náležitosti pro zahájení testů. Těmi může být

- Specification,
- Definitions a
- Command for compilation.

Příloha D

Popis požadavků webového REST API

Atributy označené * jsou povinné.

`/files/add-dir/{project_id}/{name}/{path}`			`PUT, POST`
Popis:	Ve složce projektu **`project_id`** vytvoří novou složku v umístění **`path`** s názvem **`name`**.		
Parametry:		**`project_id`***	Název projektu.
		`name`*	Název nové složky.
		`path`	Souborové umístění nové složky v rámci souborů projektu **`project_id`**. Pokud není dáno, nová složka bude umístěna do kořenového adresáře projektu.
Odpovědi:	`200`	Složka byla úspěšně vytvořena.	
	`406`	Složku daného názvu **`name`** nebylo v umístění **`path`** projektu **`project_id`** možné vytvořit.	

`/files/get-content/{project_id}/{path}`			`GET`
Popis:	Ve složce projektu **`project_id`** najde a vrátí obsah souboru daného cestou **`path`**.		
Parametry:		**`project_id`***	Název projektu.
		`path`*	Cesta k požadovanému souboru.
Odpovědi:	`200`	Je vrácen obsah požadovaného souboru. **`content-type : text/plain`**	
	`404`	Daný soubor nebyl nalezen.	

`/files/ls/{project_id}/{path}`			`GET`
Popis:	Vrátí obsah složky projektu **`project_id`** dané cestou **`path`**.		
Parametry:		**`project_id`***	Název projektu.
		`path`*	Umístění složky jejíž obsah je požadován.
Odpovědi:	`200`	Je vrácen obsah složky jako seznam objektů.	

`content-type : application/json`

[{

„**filename**“: Jméno souboru.

„**is_dir**“: Booleovská hodnota nastavená na `true`, pokud je daný soubor složkou, jinak `false`.

„**uploaded**“: Datum nahrání souboru ve formátu `mm/dd/YYY, HH:MM`.

}]

`406` Obsah dané složky nebylo možné získat.

`/files/remove/{project_id}/{path}` **`DELETE, POST`**

Popis: Z projektu **`project_id`** odstraní soubor daný umístěním **`path`**.

Parametry:
- **`project_id`*** — Název projektu.
- **`path`*** — Cesta k odstraňovanému souboru.

Odpovědi:
- `200` Soubor byl úspěšně odstraněn.
- `406` Soubor se nepodařilo odstranit.

`/files/upload/{project_id}/{what}/{path}` **`POST`**

Popis: Do souborového systému projektu **`project_id`** nahraje do umístění určeného **`path`** nový soubor, daný obsahem přiloženého formuláře, a označí ho jako entitu danou hodnotou **`what`**.

Parametry:
- **`project_id`*** — Název projektu.
- **`what`*** — Označení nahrávaného souboru. Možné hodnoty: „specification“|„definitions“|„source“
- **`path`** — Souborové umístění nového souboru v rámci souborů projektu **`project_id`**. Pokud není dáno, nový soubor bude umístěn do kořenového adresáře projektu.

Data formuláře:
- „**`file`**“* — Nahrávaný soubor.

Odpovědi:
- `200` Soubor byl úspěšně nahrán.
- `400` Neplatný obsah formuláře.
- `404` Projekt s daným **`project_id`** nebyl nalezen.

`/project/add/{name}` **`POST`**

Popis: Vytvoří nový projekt pojmenovaný **`name`** a nastaví mu další parametry dané obsahem formuláře.

Parametry:
- **`name`** — Název nového projektu. Pokud není dán, musí formulář obsahovat buďto archiv .zip jehož jméno bude pro identifikaci nového projektu použito, nebo parametr **`name`** definovaný jako položku formuláře.

Data formuláře:

„**compile**“	Příkaz ke kompilaci nového projektu.
„**definitions**“	Hlavičkový soubor se stavovými proměnnými.
„**name**“	Název nového projektu. Použije se pouze pokud není definován parametr **name** v adrese použité ve specifikaci.
„**src**“	Zip archiv, který bude rozbalen do kořenového adresáře zakládaného projektu. Pokud není název projektu specifikován jinak, použije se jméno tohoto archivu.
„**specification**“	Soubor se specifikací temporálních vlastností.

Odpovědi:

200 Projekt byl úspěšně vytvořen, je vrácen objekt reprezentující stav projektu.

content-type : application/json

[{

„**compile-cmd**“:	Příkaz ke kompilaci projektu.
„**definitions**“:	Umístění a datum aktualizace hlavičkového souboru se stavovými proměnnými.
„**missing**“:	Seznam řetězců popisujících chybějící náležitosti pro zahájení testů.
„**name**“:	Název projektu.
„**ready**“:	Booleovská hodnota říkající, jestli je projekt připraven na testování.
„**specification**“:	Umístění a datum aktualizace souboru se specifikací temporálních vlastností.
„**test-cases**“:	Seznam objektů testovacích případů popsaných např. v odpovědi následujícího požadavku.

}]

406 Projekt nebylo možné vytvořit, nebyl specifikován jeho název.

/project/get-test-cases/{project_id} **GET**

Popis: Vrátí seznam objektů reprezentujících testovací případy projektu **project_id**.

Parametry:

project_id*	Název projektu.

Odpovědi:

200 Je vrácen seznam objektů reprezentujících testovací případy projektu.

content-type : application/json

[{

„**cmd**“:	Příkaz ke spuštění testovacího případu.
„**executions**“:	Seznam číselných identifikátorů spuštění, které prováděly tento test.
„**id**“:	Číselný identifikátor testovacího případu.
„**input**“:	Soubor, který má být testovacímu případu

připojen na vstup.

„**last-run**“: Datum posledního spuštění testu ve formátu `mm/dd/YYY, HH:MM`.

„**running**“: Booleovská hodnota určující, jestli je tento test aktuálně vyhodnocován.

„**passed**“: Booleovská hodnota říkající, jestli tento test při posledním spuštění prošel verifikací.

„**timeout**“: Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.

}]

404 Projekt s daným **project_id** nebyl nalezen.

/project/get-status/{project_id} **GET**

Popis: Vrátí objekt reprezentující projekt **project_id**.

Parametry: **project_id*** Název projektu.

Odpovědi: **200** Je vrácen objekt reprezentující daný projekt.
content-type : application/json
[{

„**compile-cmd**“: Příkaz ke kompilaci projektu.

„**definitions**“: Umístění a datum aktualizace hlavičkového souboru se stavovými proměnnými.

„**missing**“: Seznam řetězců popisujících chybějící náležitosti pro zahájení testů.

„**name**“: Název projektu.

„**ready**“: Booleovská hodnota říkající, jestli je projekt připraven na testování.

„**specification**“: Umístění a datum aktualizace souboru se specifikací temporálních vlastností.

}]

404 Projekt s daným **project_id** nebyl nalezen.

/project/is-ready/{project_id} **GET**

Popis: Zjistí stav projektu **project_id** z hlediska připravenosti na spouštění testů.

Parametry: **project_id*** Název projektu.

Odpovědi: **200** Je vrácen stav projektu z hlediska připravenosti na testování.
content-type : application/json
[{

„**missing**“: Seznam řetězců popisujících chybějící náležitosti pro zahájení testů.

„**ready**“: Booleovská hodnota říkající, jestli je projekt připraven na testování.

}]

404 Projekt s daným **project_id** nebyl nalezen.

`/project/remove/{project_id}` **`DELETE, POST`**

Popis:		Nevratně odstraní projekt **`project_id`**.	
Parametry:		**`project_id`***	Název projektu.
Odpovědi:	`200`	Projekt byl úspěšně odstraněn.	
	`406`	Projekt nebylo možné odstranit.	

`/project/update-compile-cmd/{project_id}` **`POST`**

Popis:		Přepíše příkaz ke kompilaci projektu **`project_id`** novým příkazem z formuláře.	
Parametry:		**`project_id`***	Název projektu.
Data formuláře:		„**`cmd`“	Nový příkaz ke kompilaci projektu.
Odpovědi:	`200`	Příkaz ke kompilaci byl úspěšně upraven.	
		`content-type : application/json`	
		[{	
		„**`compile-cmd`“:	Příkaz ke kompilaci projektu.
		„**`definitions`“:	Umístění a datum aktualizace hlavičkového souboru se stavovými proměnnými.
		„**`missing`“:	Seznam řetězců popisujících chybějící náležitosti pro zahájení testů.
		„**`name`“:	Název projektu.
		„**`ready`“:	Booleovská hodnota říkající, jestli je projekt připraven na testování.
		„**`specification`“:	Umístění a datum aktualizace souboru se specifikací temporálních vlastností.
		}]	
	`400`	Neplatný obsah formuláře.	
	`404`	Projekt s daným **`project_id`** nebyl nalezen.	

`/project/update-test-cases/{project_id}` **`POST`**

Popis:		Nahradí seznam testovacích případů projektu **`project_id`** seznamem z formuláře.	
Parametry:		**`project_id`***	Název projektu.
Data formuláře:		„**`test-cases`“	Seznam objektů s atributy:
		[{	
		„**`cmd`“***:	Příkaz ke spuštění testovacího případu.
		„**`executions`“**:	Seznam číselných identifikátorů spuštění, které prováděly tento test.
		„**`id`“***:	Číselný identifikátor testovacího případu.
		„**`input`“**:	Soubor, který má být testovacímu případu připojen na vstup.
		„**`last-run`“**:	Datum posledního spuštění testu ve formá-

tu `mm/dd/YYY, HH:MM`.

„**running**“: Booleovská hodnota určující, jestli je tento test aktuálně vyhodnocován.

„**passed**“: Booleovská hodnota říkající, jestli tento test při posledním spuštění prošel verifikací.

„**timeout**“: Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.

}]

Odpovědi:
- **200** Seznam testovacích případů byl úspěšně aktualizován.
- **400** Neplatný obsah formuláře.
- **404** Projekt s daným **project_id** nebyl nalezen.

/test-case/add/{project_id}/{cmd} — **PUT, POST**

Popis: Pro projekt **project_id** vytvoří nový testovací případ, který bude spuštět předaný příkaz **cmd**.

Parametry:
- **project_id*** — Název projektu.
- **cmd*** — Příkaz nového testovacího případu.

Odpovědi:
- **200** Testovací případ byl úspěšně přidán, je vrácen číselný identifikátor **id**, který mu přidělen.
  content-type : text/plain
- **404** Projekt s daným **project_id** nebyl nalezen.

/test-case/execute/{project_id}/{tc_list} — **POST**

Popis: Spustí vyhodnocování testovacích případů projektu **project_id**, daných seznamem jejich **id** v **tc_list**.

Parametry:
- **project_id*** — Název projektu.
- **tc_list*** — Seznam identifikátorů testovacích případů, které mají být spuštěny, oddělených čárkou.

Odpovědi:
- **200** Vyhodnocování testovacích případů bylo úspěšně spuštěno, je vrácen seznam číselných identifikátorů vytvořených spuštění.
  content-type : application/json
- **404** Projekt s daným **project_id** nebyl nalezen.
- **406** Projekt s daným **project_id** nemá nastaveny všechny náležitosti pro spouštění testů, viz **/project/is-ready/{project_id}**.

/test-case/remove/{project_id}/{tc_list} — **DELETE, POST**

Popis: Odstraní z projektu **project_id** všechny testovací případy dané seznamem jejich **id** v **tc_list**.

Parametry:
- **project_id*** — Název projektu.
- **tc_list*** — Seznam identifikátorů testovacích případů, které mají být odstraněny, oddělených čárkou.

Odpovědi: **200** Testovací případy byly úspěšně odstraněny. Je vrácen aktualizovaný seznam objektů, které reprezentují testovací případy pro daný projekt **project_id**.

content-type : application/json

[{

„**cmd**“:	Příkaz ke spuštění testovacího případu.
„**executions**“:	Seznam číselných identifikátorů spuštění, které prováděly tento test.
„**id**“:	Číselný identifikátor testovacího případu.
„**input**“:	Soubor, který má být testovacímu případu připojen na vstup.
„**last-run**“:	Datum posledního spuštění testu ve formátu `mm/dd/YYY, HH:MM`.
„**running**“:	Booleovská hodnota určující, jestli je tento test aktuálně vyhodnocován.
„**passed**“:	Booleovská hodnota říkající, jestli tento test při posledním spuštění prošel verifikací.
„**timeout**“:	Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.

}]

404 Projekt s daným **project_id** nebyl nalezen.

/test-case/status/{project_id}/{tcid} **GET**

Popis: Vrátí objekt reprezentující testovací případ projektu **project_id** daný jeho číselným identifikátorem **tcid**.

Parametry:

project_id*	Název projektu.
tcid*	Číselný identifikátor testovacího případu.

Odpovědi: **200** Je vrácen objekt reprezentující požadovaný testovací případ.

content-type : application/json

{

„**cmd**“:	Příkaz ke spuštění testovacího případu.
„**executions**“:	Seznam číselných identifikátorů spuštění, které prováděly tento test.
„**id**“:	Číselný identifikátor testovacího případu.
„**input**“:	Soubor, který má být testovacímu případu připojen na vstup.
„**last-run**“:	Datum posledního spuštění testu ve formátu `mm/dd/YYY, HH:MM`.
„**running**“:	Booleovská hodnota určující, jestli je tento test aktuálně vyhodnocován.
„**passed**“:	Booleovská hodnota říkající, jestli tento test při posledním spuštění prošel verifikací.
„**timeout**“:	Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.

}

404 Projekt s daným **project_id** nebyl nalezen, nebo neobsahuje testovací případ s daným **tcid**

`/test-case/update/{project_id}/{tcid}` **`POST`**

Popis: Aktualizuje reprezentaci testovacího případu projektu **`project_id`** s číselným identifikátorem **`tcid`** podle dat zaslaných ve formuláři.

Parametry:

`project_id`*	Název projektu.
`tcid`*	Číselný identifikátor testovacího případu.

Data formuláře:

„**`cmd`“**:	Příkaz ke spuštění testovacího případu.
„**`timeout`“**:	Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.
„**`input`“**:	Soubor, který má být testovacímu případu připojen jako standardní vstup.

Odpovědi:

`200` Je vrácen objekt aktualizovaného testovacího případu.
`content-type : application/json`

{

„**`cmd`“**:	Příkaz ke spuštění testovacího případu.
„**`executions`“**:	Seznam číselných identifikátorů spuštění, které prováděly tento test.
„**`id`“**:	Číselný identifikátor testovacího případu.
„**`input`“**:	Soubor, který má být testovacímu případu připojen na vstup.
„**`last-run`“**:	Datum posledního spuštění testu ve formátu `mm/dd/YYY, HH:MM`.
„**`running`“**:	Booleovská hodnota určující, jestli je tento test aktuálně vyhodnocován.
„**`passed`“**:	Booleovská hodnota říkající, jestli tento test při posledním spuštění prošel verifikací.
„**`timeout`“**:	Počet sekund, po kterém bude vykonávání testovacího případu ukončeno.

}

`404` Projekt s daným **`project_id`** nebyl nalezen, nebo neobsahuje testovací případ s daným **`tcid`**

`/test-execution/get-compile-log/{project_id}/{teid}` **`GET`**

Popis: Vrátí obsah souboru se záznamem standardního i chybového výstupu kompilace projektu **`project_id`**, prováděné v rámci spuštění testu s číselným identifikátorem **`teid`**.

Parametry:

`project_id`*	Název projektu.
`teid`*	Číselný identifikátor spuštění testu.

Odpovědi:

`200` Je vrácen obsah odpovídajícího souboru.
`content-type : text/plain`

`404` Daný soubor nebyl nalezen.

`/test-execution/get-log/{project_id}/{teid}` **`GET`**

Popis: Vrátí obsah souboru se záznamem standardního i chybového výstupu spuštění testu projektu **`project_id`**, prováděného s číselným identifikátorem **`teid`**.

Parametry:

`project_id`*	Název projektu.
`teid`*	Číselný identifikátor spuštění testu.

Odpovědi:

`200` Je vrácen obsah odpovídajícího souboru.
`content-type : text/plain`

`404` Daný soubor nebyl nalezen.

`/test-execution/get-status/{project_id}/{teid}` **`GET`**

Popis: Vrátí objekt reprezentující spuštění testu projektu **`project_id`**, prováděného s číselným identifikátorem **`teid`**.

Parametry:

`project_id`*	Název projektu.
`teid`*	Číselný identifikátor spuštění testu.

Data formuláře:

`file`	popis

Odpovědi:

`200` Je vrácen objekt odpovídajícího spuštění testu.
`content-type : application/json`
{

„**`run`“**:	Datum spuštění testu ve formátu `mm/dd/YYY, HH:MM`.
„**`passed`“**:	Booleovská hodnota říkající, jestli tento test prošel verifikací.
„**`test-case`“**:	Číselný identifikátor testovacího případu, který byl v rámci tohoto testu spuštěn.

}

`404` Projekt s daným **`project_id`** nebyl nalezen, nebo neobsahuje spuštění testu s daným **`teid`**

`/test-execution/remove/{project_id}/{teid}` **`DELETE, POST`**

Popis: Smaže veškeré soubory vytvořené v rámci spuštění testu projektu **`project_id`**, prováděného s číselným identifikátorem **`teid`**.

Parametry:

`project_id`*	Název projektu.
`teid`*	Číselný identifikátor spuštění testu.

Odpovědi:

`200` Soubory byly úspěšně odstraněny.

`406` Soubory se nepodařilo odstranit.