VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PŘEKLAD XTR VÝSTUPU NÁSTROJE UPPAAL DO UŽIVATELSKY PŘÍVĚTIVÉ REPREZENTACE

CONVERSION OF XTR OUTPUT OF UPPAAL TOOL INTO USER-FRIENDLY REPRESENTATION

DIPLOMOVÁ PRÁCE
MASTER’S THESIS

AUTOR PRÁCE **Bc. ANTONÍN MAZÁNEK**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Mazánek Antonín, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Překlad XTR výstupu nástroje UPPAAL do uživatelsky přívětivé reprezentace**
Conversion of XTR Output of UPPAAL Tool into User-Friendly Representation
Kategorie: Algoritmy a datové struktury
Zadání:

1. Shrňte principy a možnosti modelování a analýzy systémů prostředky variant nástroje UPPAAL.
2. Zdokumentujte i) formát XTR používaný pro uložení simulačního běhu, např. protipříkladu, generovaného nástrojem UPPAAL, ii) související formáty dat, např. IF (Intermediate Format), a iii) dostupnou programovou podporu pro práci s XTR a souvisejícími formáty dat, např. UTAP (Uppaal Timed Automata Parser Library).
3. Navrhněte uživatelsky přívětivou reprezentaci UPPAAL běhů pro jejich ukládání, vizualizaci atd. (reprezentace může být odlišná pro jednotlivé případy jejího užití).
4. Implementujte snadno přenositelné a udržovatelné řešení (aplikaci, knihovnu, skript atp.) schopné přeložit XTR data do navržené reprezentace. Součástí překladu nechť je možnost volby filtrace/zúžení dat (např. na zvolenou množinu stavů, přechodů či hodnot proměnných).
5. Vhodně prokažte a zhodnoťte vlastnosti zvolené reprezentace a řešení překladu XTR dat.
6. Diskutujte možné směry pokračování v projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

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

Vedoucí práce: **Strnadel Josef, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 25. října 2019

Abstrakt

Diplomová práce seznamuje s nástrojem Uppaal. Popisuje principy a možnosti modelování a analýzy systémů pomocí tohoto nástroje. Dále se blíže zabývá formáty souborů, které nástroj Uppaal používá. Strukturou souboru XML, sloužící pro uložení vytvořených systémů, formátem XTR, který nástroj používá k ukládání simulačních běhů, a formátem IF, který je nutný k porozumění obsahu souboru ve formátu XTR. V textu je zmíněna i dostupná programová podpora pro práci s těmito formáty. Další částí, kterou se tato diplomová práce zabývá, já návrh uživatelsky přívětivé reprezentace simulačních běhů spolu s návrhem a implementací aplikace, která provádí překlad simulačních běhů nástroje Uppaal do navržené reprezentace. Na konci práce je zmíněno možné pokračování v projektu spolu s hodnocením navržené reprezentace a aplikace pro překlad.

Abstract

The master's thesis introduces the Uppaal tool. It describes the principles and possibilities of modeling and analysis of systems using this tool. It also discusses in more detail the file formats that Uppaal tool uses. The structure of the XML file used to store created systems, the XTR format, which Uppaal uses to store simulation traces, and the IF format, which is necessary to understand the contents of the file in XTR format. The text also mentions available software support for working with these formats. Next part of this master's thesis is about designing user-friendly representation of simulations traces, along with designing application that translates simulation traces into designed representation. At the end of this thesis, the possible continuation of the project is mentioned together with the evaluation of the designed representation and the application.

Klíčová slova

Uppaal, XTR, simulační běh, IF, Intermediate Format, překlad, uživatelsky přívětivá reprezentace, JSON, .NET Core

Keywords

Uppaal, XTR, simulation trace, IF, Intermediate Format, conversion, user-friendly representation, JSON, .NET Core

Citace

MAZÁNEK, Antonín. *Překlad XTR výstupu nástroje UPPAAL do uživatelsky přívětivé reprezentace.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Překlad XTR výstupu nástroje UPPAAL do uživatelsky přívětivé reprezentace

Prohlášení

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

....
Antonín Mazánek
30. května 2020

Poděkování

Děkuji svému vedoucímu diplomové práce Ing. Josefovi Strnadelovi, Ph. D. za odbornou pomoc, cenné rady a profesionální vedení při tvorbě této diplomové práce. Dále bych rád poděkoval rodině a přátelům za morální podporu.

Obsah

1 Úvod **3**
1.1 Časovaný automat 6
1.2 Temporální logika 8
1.3 Ověřování modelů 9

2 Podrobnosti k řešenému problému **12**
2.1 Nástroj Uppaal 12
2.1.1 Návrh systému 13
2.1.2 Simulace 15
2.1.3 Verifikace 16
2.1.4 Nástroje související s Uppaal 18
2.2 Používané typy souborů 19
2.2.1 XML - uložený systém 20
2.2.2 XTR - uložený simulační běh 20
2.2.3 IF - pomocný soubor pro překlad 21
2.3 Dostupná programová podpora 23

3 Návrh a implementace **25**
3.1 Navržená reprezentace simulačních běhů 25
3.2 Použité prostředky 27
3.3 Návrh aplikace 28
3.4 Popis překladu 29
3.5 Filtrace 31
3.6 Problémy a jejich řešení 33

4 Zhodnocení navržené reprezentace a překladu **35**
4.1 Dotazník 35
4.2 Testování 36
4.3 Odhad složitosti 38
4.4 Návrhy pro pokračování v projektu 39

5 Závěr **42**

Literatura **43**

A Soubor XML **45**

B Formát XTR **47**

C Formát IF **48**

D Struktura navržené reprezentace **49**

E Nápověda aplikace **51**

Kapitola 1

Úvod

Uživatelsky přívětivý výstup je v dnešní době jeden z hlavních požadavků při návrhu aplikace. Podobně jako vstup aplikace, který musí být uživatelsky přívětivý, musí být i výstup přehledný a dobře čitelný. U aplikací s grafickým uživatelským rozhraním je důležité, aby vše podstatné bylo snadno přístupné, a zároveň, aby výstup nepůsobil rušivým dojmem. U aplikací bez grafického rozhraní, kde je výstup zpravidla jen výpis znaků do konzole či souboru, je důležité, aby se daný výstup dal dobře číst. K tomu slouží formátování textu. Existuje velké množství textových formátů, přičemž každý z nich je jinak vhodný na jiný typ výstupu.

Tato diplomová práce se zabývá návrhem uživatelsky přívětivé reprezentace simulačních běhů nástroje Uppaal, přičemž reprezentace musí umožňovat přehledné zobrazení i zkrácených běhů pomocí filtračních možností. Dále se zabývá návrhem a implementací aplikace, která bude schopna přeložit uložený simulační běh do navržené reprezentace.

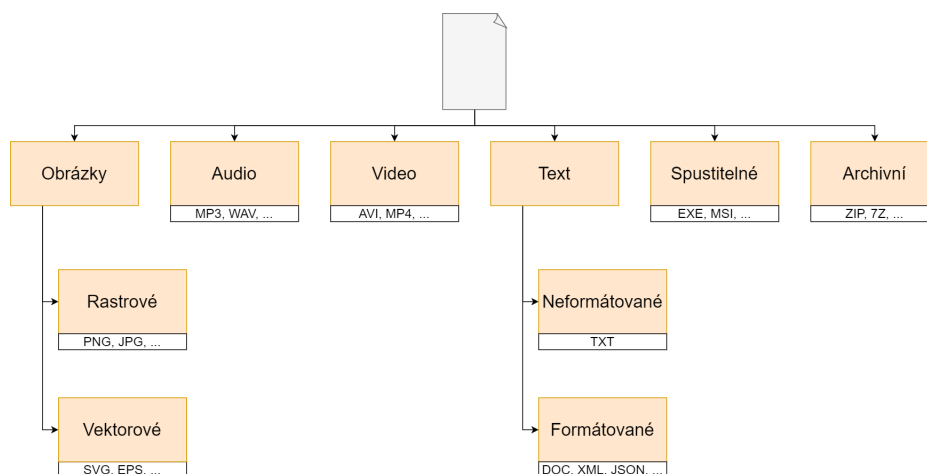
Nástroj Uppaal je, spolu s popisem formátů souborů, které používá, podrobněji popsán v dalších částech tohoto textu. Je zmíněna i dostupná programová podpora pro práci s těmito formáty. Především je popsána knihovna UTAP (Uppaal Timed Automata Parser) a její program `tracer`. Program `tracer` je použit i při kontrole správnosti implementované aplikace, kdy je porovnán výstup aplikace s výstupem nástroje `tracer` (s upraveným formátováním). Dále jsou uvedeny některé další nástroje, které s nástrojem Uppaal souvisí, a které rozšiřují/upravují některé jeho funkce. Mezi tyto nástroje patří např. Stratego, Cora nebo Tron.

Uppaal je nástroj, který slouží k modelování, simulaci, analýze a verifikaci systémů reálného času. Tento nástroj je výsledkem kolaborace dvou univerzit, a to Uppsala University, Sweden a Aalborg University in Denmark. Hlavními konstrukčními cíli byla efektivita a jednoduchost použití[17]. Systémy jsou v tomto nástroji modelovány jako sít časovaných automatů rozšířených o datové typy, proměnné, programovatelné funkce a komunikační kanály[5]. Nejčastější využití tohoto nástroje je u systémů, které lze modelovat jako kolekce nedeterministických procesů s konečných stavových řízením a hodinami reálného času. Tyto procesy spolu mohou komunikovat pomocí předávání zpráv přes kanály nebo pomocí sdílených proměnných. Mezi takové systémy patří například řadiče pracující v reálném čase nebo komunikační protokoly[17].

V následujících podkapitolách jsou blíže popsány časované automaty a jejich propojení do sítě. Dále je popsán úvod k temporální logice, kterou nástroj Uppaal používá k ověřování vlastností systému, a je zmíněn i „model checking" vlastností časovaných systémů, pro vlastnosti popsané formulemi logiky a časované systémy popsané časovanými automaty. Nejdříve je ale nutné zasadit výše zmíněné pojmy do souvislostí.

Prvním pojmem je uživatelská přívětivost. Jak již bylo nastíněno, jedná se o velice důležitý pojem při tvorbě produktu, který je orientovaný na uživatele. Anglicky by se dala uživatelská přívětivost vyjádřit jako user experience (UX). Tento pojem zahrnuje několik oblastí, které přispívají k uživatelské přívětivosti. Podle [11] UX zahrnuje všechny aspekty interakce koncového uživatele s vytvořeným produktem. UX se mimo jiné zabývá celkovou přehledností produktu, intuitivností použití, jednoduchostí a srozumitelností informací. Oblastmi UX jsou následně: užitečnost, nalezitelnost, důvěryhodnost, přitažlivost, přístupnost, hodnotnost a použitelnost[9]. Pro návrh uživatelsky přívětivé reprezentace simulačních běhů je asi nejdůležitější oblastí použitelnost. Hodnocení použitelnosti je však velice subjektivní, a proto vznikly metody, jak uživatelskou přívětivost zhodnotit. Těchto metod existuje opravdu velké množství. Liší se například tím, jestli jsou v testování zahrnuti koncoví uživatelé, zda je u testování moderátor nebo i časovou náročností[9]. Pro každý typ produktu jsou vhodné jiné metody, vždy však platí, že je vhodné testování provézt. Pokud by testování provedeno nebylo, používání produktu by mohlo být velice nepříjemné, což by pro firmu, která produkt vytvořila, mohlo znamenat i ztrátu na popularitě a potenciálně i ztrátu budoucích zákazníků. Mezi typické představitele metod pro ověřování uživatelské použitelnosti patří např.: dotazníky, ankety, skupinové diskuze, A/B testování nebo automatický zápis aktuálního používání[9]. Pro metodu ověření použitelnosti pomocí dotazníků, lze například pro jednotlivé vstupy/výstupy vytvořit konkrétní příklady, a za popisem těchto příkladů položit několik otázek, které hodnotí uživatelskou přívětivost.

Za několik desítek let, kdy lidstvo používá počítače, vzniklo obrovské množství typů formátu dat. Pro různé kategorie máme vlastní typy souboru, které využívají jiných principů, jak jejich obsah zakódovat, a případně i zkomprimovat, do bajtů, které se dají uložit na disk. Formátů je již tolik, že je prakticky nemožné je všechny klasifikovat do oddělených kategorií. Nejpoužívanějšími typy formátů jsou však obrázky, audio a video soubory, textové soubory, spustitelné soubory a archivní soubory. Na obrázku 1.1 je uveden diagram, který popisuje základní rozdělení nejpoužívanějších formátů dat spolu s příklady pro jednotlivé kategorie.



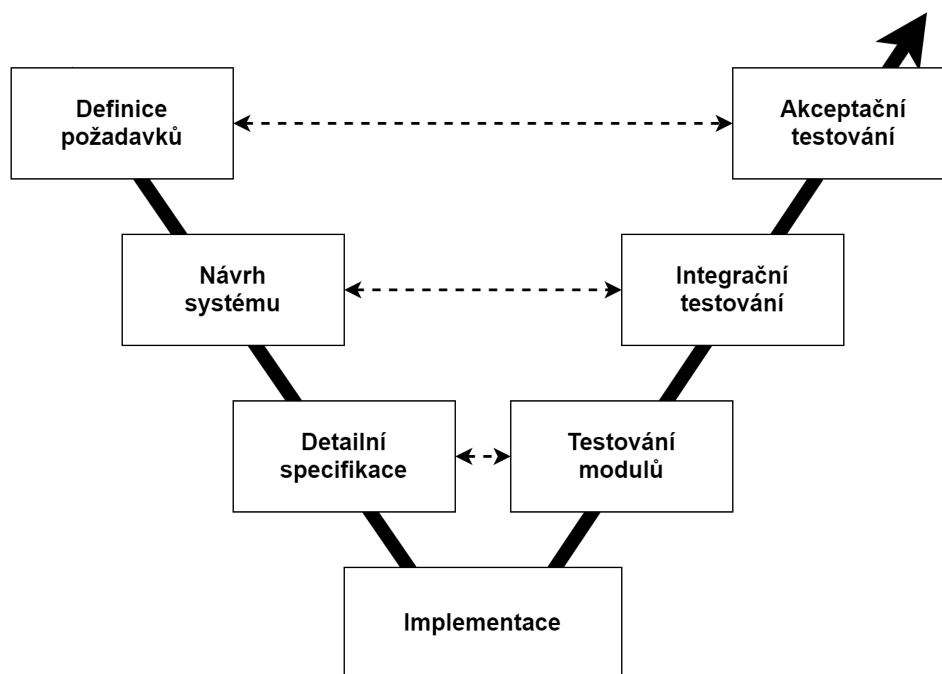
Obrázek 1.1: Základní klasifikace nejpoužívanějších formátů dat.

Jak je vidět i na diagramu, některé kategorie jsou dále děleny (např. text). Neformátované textové soubory obsahují pouze znaky, zatímco formátované soubory obsahují kromě samotného obsahu i řídící značky či příkazy. Konkrétně u formátu JSON se jedná o složené a hranaté závorky, uvozovky, dvojtečky a čárky. Jednotlivé typy formátů dat v daných

kategoriích mohou být jinak vhodné pro použití v aplikaci, která má být uživatelsky přívětivá (tedy orientovaná na uživatele). Například použití neformátovaného textového souboru, který obsahuje oddělitelné informace, bude méně vhodné, než použití formátovaného souboru, ve kterém se lze snadněji orientovat a vyhledávat.

Dříve bylo zmíněno, že nástroj Uppaal slouží mimo jiné k modelování a simulaci. Modelování je proces vytvoření nového systému (modelu), který reprezentuje konkrétní reálný systém. Systém je soubor elementárních prvků, které spolu nějak souvisí. Simulace je proces experimentování s vytvořeným modelem za účelem získání nových informací, případně ověření správnosti. Model může být specifikován různými prostředky, např. konečnými automaty, časovanými automaty nebo Petriho sítěmi. Dále rozlišujeme otevřené a uzavřené systémy. Otevřené systémy vnímají okolí kolem sebe a mohou s ním komunikovat, zatímco uzavřené systémy okolí ignorují. Podle typu systému se dále simulace rozlišují na diskrétní a spojité. Při diskrétní simulaci mají všechny prvky diskrétní chování, zatímco při spojité simulaci mají všechny prvky chování spojité.

Ověřování správnosti vytvořeného produktu je nedílnou součástí vývojového cyklu. Jeden z modelů vývojového cyklu je „V-Model“ zobrazený na obrázku 1.2, ve kterém v levé části probíhá specifikování požadavků a návrh aplikace a po implementaci následuje pravá část, ve které se vytvořené dílo testuje. Jednotlivé části testování postupují od nejmenších celků, tzv. „modulů“, až po kompletní výsledný systém. Každá část je zároveň svázána s etapou návrhu, která bude ovlivněna, pokud test selže.



Obrázek 1.2: V-Model vývojového cyklu aplikace.

Ověřování správnosti je označováno jako verifikace (někdy funkční verifikace). Kromě klasického testování, které lze provádět buď ručně, kdy zkoušíme, co daný produkt dělá, a ověřujeme, jestli to dělá správně, nebo automatizovaně pomocí simulací, lze použít i formální prostředky. Formální verifikace je způsob ověření správnosti či nesprávnosti systému vzhledem k formálním specifikacím daných vlastností. Stejně jako testování lze formální

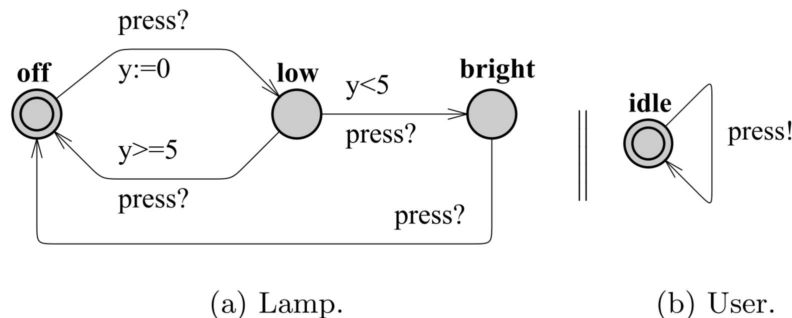
verifikaci provádět ručně nebo automatizovaně. Ruční verifikace spočívá ve vytvoření matematického důkazu, že daná vlastnost je splněna nebo nesplněna, zatímco automatická formální verifikace by se dala popsat jako algoritmus, kterému je předán model a specifikované vlastnosti, a který ověřuje, zda jsou dané vlastnosti splněny. Existuje několik automatických formálních verifikací. Jednou z nich je ověřování modelů („model checking“) zmíněný dále. Velkou výhodou formální verifikace oproti testování je její schopnost přesně určit, že daný systém odpovídá specifikaci. Na druhou stranu cena tohoto ověření je mnohdy podstatně vyšší než cena klasického testování.

1.1 Časovaný automat

Časovaný automat (Timed Automata) je konečný automat rozšířený o hodiny (šestice v 1.1), kde L je konečná množina stavů, l_0 je počáteční stav, C je konečná množina hodin, A je konečná množina akcí a ko-akcí sjednocená s interní τ-akcí, $E \subseteq L \times A \times B(C) \times 2^C \times L$ je množina hran mezi stavy a $I : L \to B(C)$ je funkce přiřazující stavům invarianty. $B(C)$ je množina omezení hodin obsahující prvky tvaru $x \bowtie c$ nebo $x - y \bowtie c$, kde $x, y \in C, c \in \mathbb{N}$ a $\bowtie \in \{<, \leq, =, \geq, >\}$. N-tice $(l, a, \delta, \lambda, l^{'}) \in E$ reprezentuje hranu ze stavu l do $l^{'}$ při akci a, kde δ je omezení hodin a $\lambda \subseteq C$ obsahuje hodiny, které se mají vynulovat po provedení této hrany[5].

$$M = (L, l_0, C, A, E, I) \tag{1.1}$$

Pro lepší pochopení popisu časovaných automatů je na obrázku 1.3 znázorněn systém, který je složen ze dvou časovaných automatů. Tento systém se sestává ze dvou subjektů (každý modelován vlastním časovaným automatem). Subjekt (b) reprezentuje uživatele, který obsahuje pouze jeden stav – `idle`. Z toho stavu vede „self-loop“ hrana, která simuluje, že uživatel zmáčkl tlačítko. Subjekt (a) reprezentuje lampu, která se může nacházet v jednom ze tří stavů: `off` (vypnuto), `low` (svítí s malou intenzitou) a `bright` (svítí s velkou intenzitou). Počátečním stavem je stav `off`, a pokud uživatel zmáčkne tlačítko, lampa vynuluje hodiny y a přejde do stavu `low`. V tomto stavu lampa setrvá a hodiny y poběží. Pokud uživatel zmáčkne tlačítko podruhé do 5 časových jednotek, lampa přejde do stavu `bright` a bude svítit s velkou intenzitou, dokud uživatel lampu nevypne zmáčknutím tlačítka (lampa přejde do stavu `off`). Pokud uživatel zmáčkne tlačítko po více jak 5 časových jednotkách, lampa přejde ze stavu `low` do stavu `off`.



Obrázek 1.3: Příklad časovaných automatů[5].

Zmáčknutí tlačítka je v systému popsáno jako `press!` a `press?`. Toto označení reprezentuje akci a ko-akci, což se dá chápat jako synchronizace po komunikačním kanále „press“, kde `press!` znamená vyslání a `press?` znamená přijímání.

Před popsáním samotné přechodové relace, je třeba zavést některé funkce a označení. Funkce $u : C \to \mathbb{R}_{\geq 0}$ je valuací hodin z množiny C. $\mathbb{R}^C$ je množina valuací všech hodin z množiny C a $\forall x \in C : u_0(x) = 0$. Dále se v následujícím popisu uvažují invarianty jako množiny valuací hodin. Tedy $u \in I(l)$ znamená, že u splňuje $I(l)$[5].

Přechodová relace časovaného automatu $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$, kde $S \subseteq L \times \mathbb{R}^C$ je popsána následovně:

- $(l, u) \xrightarrow{d} (l, u + d)$ pokud $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$ a
- $(l, u) \xrightarrow{a} (l', u')$ pokud $\exists\ (l, a, \delta, \lambda, l') \in E$ takové, že $u \in \delta, u' = [\lambda \mapsto 0]u$ a $u' \in I(l')$,

kde pro $d \in \mathbb{R}_{\geq 0}$, $u + d$ znamená přiřazení všem hodinám $x \in C$ hodnotu $u(x) + d$ a $[\lambda \mapsto 0]u$ znamená vynulování všech hodin z r v souladu s u[5].

Časovaný automat je tedy modelovací prostředek, který má konečné stavové řízení, a zároveň je jím možno modelovat nekonečné systémy. K definici časovaného jazyka (neboli jazyka přijímaného časovanými automaty) je nutné nejdříve uvést pojem časované slovo. Časované slovo je sekvence $(a_0, u_0)(a_1, u_1)\ldots$, kde $a_i \in A$, $u_i \in \mathbb{R}^+ . u_i \geq u_{i-1}$ pro $i > 0$. Časované slovo $w = (a_0, u_0) \ldots (a_k, u_k)$ je přijato časovaným automatem M, pokud existuje běh (posloupnost konfigurací), který začíná v počátečním stavu a po k provedených akcích skončí ve stavu, který patří do množiny koncových stavů (L_{acc}). Tato množina v původní definici časovaného automatu uvedena nabyla, jedná se však o jednoduché rozšíření, kde $L_{acc} \subseteq L$. Jazyk časovaných automatů je $\mathcal{L}(M) = \{w \mid w \text{ je přijato automatem } M\}$.

Příklad běhu automatu, který reprezentuje lampu z obrázku 1.3, by mohl začínat například následovně: (Lamp.off, $y = 0$) $\to$ (Lamp.off, $y = 2$) $\to$ (Lamp.low, $y = 0$) $\to$ (Lamp.low, $y = 1.7$) $\to$ (Lamp.bright, $y = 1.7$) $\to$ (Lamp.bright, $y = 10$) $\to \cdots$

Síť časovaných automatů

Předchozí příklad uvažoval systém se dvěma časovanými automaty. Nástroj Uppaal dovoluje pracovat s několika časovanými automaty paralelně. Tato skupina automatů se označuje jako síť časovaných automatů. V následující části této podkapitoly je popsána přechodová funkce sítě časovaných automatů.

Nechť $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, I_i)$ kde $1 \leq i \leq n$ označuje síť n časovaných automatů. Dále nechť $\bar{l} = (l_1, ..., l_n)$ reprezentuje vektor stavů, ve kterých se nachází jednotlivé automaty v síti $\mathcal{A}_i$. $I(\bar{l}) = \wedge_i I_i(l_i)$ je funkce, která obsahuje všechny funkce přiřazující stavům invarianty z jednotlivých časovaných automatů. A $\bar{l}[l_i'/l_i]$ reprezentuje záměnu i-tého prvku (l_i) v množině $\bar{l}$ za prvek l_i'. Dále je převzata nadefinovaná funkce z popisu časovaného automatu u[5].

Přechodová relace sítě časovaných automatů $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$, kde $S \subseteq (L_1 \times ... \times L_n) \times \mathbb{R}^C$ je popsána následovně[5]:

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ pokud $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(\bar{l})$
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i], u')$ pokud $\exists\ l_i \xrightarrow{\tau \delta \lambda} l_i'$ taková, že $u \in \delta, u' = [\lambda \mapsto 0]u$ a $u' \in I(\bar{l}[l_i'/l_i])$,
- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_j'/l_j, l_i'/l_i], u')$ pokud $\exists\ l_i \xrightarrow{c?\delta_i \lambda_i} l_i'$ a $l_j \xrightarrow{c!\delta_j \lambda_j} l_j'$ takové, že $u \in (\delta_i \wedge \delta_j), u' = [(\lambda_i \cup \lambda_j) \mapsto 0]u$ a $u' \in I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

Časté jevy v časovaných automatech

Následující sekce podkapitoly o časovaných automatech se zabývá častými jevy, které mohou nastat při modelování systémů pomocí časovaných automatů. Konkrétně se jedná o pojmy časově konvergentní a divergentní cesta, timelock a zeno běh (cesta).

Funkce *ExecTime* uvedená ve vzorci 1.2 reprezentuje čas kroku a funkce uvedená ve vzorci 1.3 reprezentuje čas běhu automatu, kde ρ je nekonečný běh[1].

$$ExecTime(\tau) = \begin{cases} 0 & \texttt{pokud} \quad \tau \in A \\ d & \texttt{pokud} \quad \tau = d, d \in \mathbb{R}_{>0} \end{cases} \tag{1.2}$$

$$ExecTime(\rho) = \sum_{i=0}^{\infty} ExecTime(\tau_i) \tag{1.3}$$

Nekonečný běh ρ je časově divergentní, pokud $ExecTime(\rho) = \infty$, v opačném případě je běh časově konvergentní[1].

Následující vzorec (1.4) udává popis množiny, která obsahuje časově divergentní cesty ze stavu l časovaného automatu M. Stav automatu l obsahuje timelock, podud $Paths_{div}(l) = \emptyset$[1].

$$Paths_{div}(l) = \{\rho \in Paths(l) \mid \rho \texttt{ je časově divergentní cesta}\} \tag{1.4}$$

Zeno běh časovaného automatu M je takový nekonečný běh, který je časově konvergentní a zároveň obsahuje nekonečně mnoho akcí[1].

1.2 Temporální logika

Temporální logika je speciálním typem modální logiky[3]. Modální logiky přebírají všechny axiomy a odvozovací pravidla z predikátové logiky. Zavádí však také dva nové operátory. Operátor nutnosti, značený $\Box$, a operátor možnosti, značený $\Diamond$. Jazyk modální logiky pak obsahuje kromě všech vět predikátové logiky prvního řádu i řetězce podle následujícího zápisu:

- $\Box f \in L, f \in L$ - formule f je nutná
- $\Diamond f \in L, f \in L$ - formule f je možná

Význam $\Box f \in L, f \in L$ je tedy takový, že formule f vždy platí a $\Diamond f \in L, f \in L$ znamená, že formule f může platit.

Temporální logika, oproti modální, navíc poskytuje formální systém pro kvalitativní popis a zdůvodnění toho, jak se hodnoty pravdivosti tvrzení mění v průběhu času. Přidává další operátory, které se vztahují k času. Mezi typické operátory patří *někdy* P, který říká, že někdy v budoucnu bude P pravdivé, a *vždy* Q, který říká, že Q je v budoucnu vždy pravdivé[3].

Podle typu systému je vhodné použít jiné varianty temporálních logik. Pro deterministické systémy, či simulace konkrétních běhů, lze použít logiku s lineárním časem LTL (Linear Temporal Logic). Pokud je potřeba tvořit formule pro nedeterministické systémy je potřeba použít jiný typ logiky, např. logiku s výpočetním stromem CTL (Computational Tree Logic). Nástroj Uppaal používá MITL (Metric Interval Temporal Logic) logiku rozšířenou o váhy [2] a zjednodušenou variantu TCTL (Timed Computation Tree Logic) logiky[5].

MITL

Logika MITL je zlomkem logiky MTL (Metric Temporal Logic). Tato logika je postavena na typu LTL logik, ale operátory vztahující se ke konkrétní hodnotě času, jsou nahrazeny operátory nad intervaly. Varianta MITL pak nedovoluje intervaly tzv. *singleton*, ve kterých je hodnota spodní i horní hranice intervalu stejná, a hodnota hranice je buď přirozené číslo, nebo nekonečno.

Formule MITL logiky lze sestavit podle následující gramatiky[2]:

$$\varphi ::= \texttt{ap} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{O}\varphi \mid \varphi_1 \mathrm{U}^x_{\leq d}\varphi_2$$

kde $\texttt{ap}$ je konjunkce predikátů přes všechny stavy v síti časovaných automatů, $d \in \mathbb{N}$ a x jsou hodiny. Logické operátory mají tradiční význam a O je operátor následujícího stavu. Formule $\mathrm{U}^x_{\leq d}\varphi_2$ využívá váhového rozšíření, které UPPAAL používá. Tato formule je pravdivá, pokud formule φ_1 je pravdivá po celou dobu běhu, než se stane pravdivou formule φ_2, přičemž hodnota hodin x nebude mít větší hodnotu než d[2].

Dále platí $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$ a jsou použity standardní MITL zkratky $\texttt{tt} = \varphi_1 \vee \neg\varphi_2$, $\Diamond_{x\leq d}\varphi = \texttt{tt}\ \mathrm{U}^x_{\leq d}\varphi$ a $\Box_{x\leq d}\varphi = \neg\Diamond_{x\leq d}\neg\varphi$[2].

TCTL

TCTL logika je rozšířením CTL logiky. CTL pracuje nad stromovou strukturou, u které má pro aktuální stav lineární minulost i budoucnost. Obsahuje modální operátory temporálních logik a zavádí dva kvantifikátory nad cestami, A (pro všechna) a E (existuje). Spolu s modálními operátory $\Box$ a $\Diamond$ lze sestavit následující formule: $A\Box$, $A\Diamond$, $E\Box$ a $E\Diamond$. Význam formule $A\Box\varphi$ je „všechny cesty z aktuálního stavu obsahují pouze stavy, ve kterých platí φ“ a význam formule $E\Box\varphi$ je „existuje cesta z aktuálního stavu, která obsahuje pouze stavy, ve kterých platí φ“. Pro variantu s $\Diamond$ budou věty končit „…obsahuje alespoň jeden stav, ve kterém platí φ“. TCTL přebírá vše od CTL a přidává časová omezení.

Zjednodušení TCTL, které nástroj UPPAAL specifikoval, spočívá v neumožnění vnořovat formule nad cestami[5]. Formule TCTL logiky lze sestavit podle následující gramatiky[12]:

$$\varphi ::= a \mid g \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid E(\varphi_1 \mathrm{U}^J \varphi_2) \mid A(\varphi_1 \mathrm{U}^J \varphi_2)$$

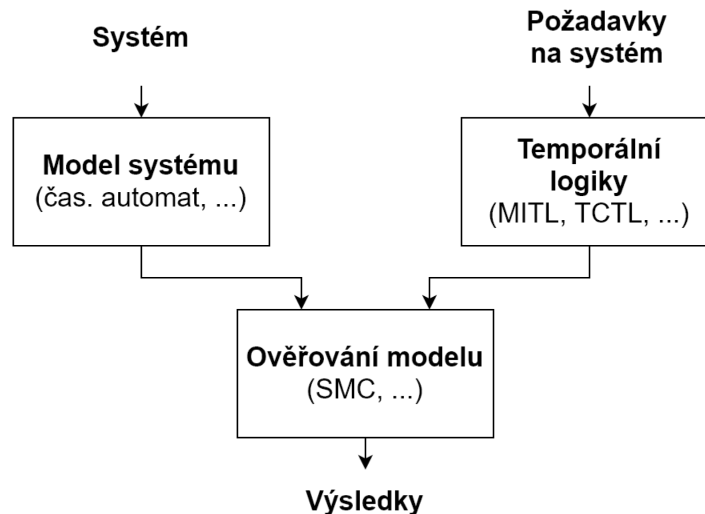
kde a je atomický výrok, g je omezení hodin, A a E jsou operátory uvedeny výše a J je interval, jehož hranice jsou přirozená čísla. Díky možnosti specifikovat formule v TCTL pomocí časových omezení, umožňuje tato logika rozhodovat problém na základě těchto omezení a hodnot hodin[12].

1.3 Ověřování modelů

Ověřování modelů (anglicky „model checking“) je metoda, která slouží k zjišťování, zda daný model s konečným stavovým řízením splňuje specifikované požadavky. Na obrázku 1.4 je znázorněn základní diagram principu ověřování modelu. Na začátku je systém, který bude modelován a následně ověřován, a požadavky na tento systém. Po vytvoření modelu daného systému, například sítí časovaných automatů, a specifikování požadavků na systém ve formě, které je schopen porozumět i počítač, například formule některé temporální logiky, dochází k ověřování modelu. Po dokončení ověřování jsou reprezentovány výsledky, často ve formě splněno/nesplněno.

V kontextu vývojového cyklu aplikace, představeného na začátku kapitoly, se ověřování modelů dá provádět, jak při testovaní jednotlivých částí systému („modulů“), kterými jsou pro síť časovaných automatů konkrétní automaty, tak v integračním testování, ve kterém se ověřuje systém jako celek (tedy celá síť). Hlavní výhodou ověřování modelů je fakt, že se jedná o typ formální verifikace. Lze tedy s určitostí potvrdit či vyvrátit, že specifikovaná vlastnost je splněna. Nevýhodou je pak složitost tohoto procesu.

Existuje několik metod ověřování modelu. Nástroj Uppaal od verze `4.1` využívá SMC (Statistical Model Checking) pro formule MITL logiky[2]. Stále však podporuje formule TCTL logiky, které byly jako jediné podporovány do verze `4.0` (včetně). Výhoda statistického ověřování modelu je možnost ověřovat tzv. „black-box“ systémy[10]. Tento typ ověřování by se dal ve V-Modelu vývojového cyklu znázornit jako testování (pravá část), které je prováděno z vrchu dolu, namísto zdola nahoru. Nejdřív je systém ověřován jako celek v podobě černé skřínky a pokud ověření selže, je skřínka rozebrána na menší černé skřínky. Pro formule TCTL logiky spočívá ověřování modelu v průchodu stavového prostoru reprezentovaného jako strom a ověřování jednotlivých částí systému. Pro rozsáhlé systémy je toto velkou nevýhodou, protože se strom může rychle rozrůst a je zapotřebí velké množství operační paměti. Na druhou stranu pokud ověřování zjistí, že specifikovaná vlastnost není splněna, lze vygenerovat protipříklad, který se skládá z posloupnosti přechodů, které vedou do stavu, ve kterém vlastnost neplatí.



Obrázek 1.4: Princip ověřování modelů.

Formule nad cestami stromu lze klasifikovat na problémy dosažitelnosti, bezpečnosti a životnosti[5]. Pro dosažitelnost lze použít formuli ve tvaru $E\Diamond\varphi$. Při specifikování bezpečnosti lze použít $A\Box\varphi$, pokud je třeba zajistit, aby φ platilo vždy, případně $E\Box\varphi$, pokud má φ platit vždy alespoň na jedné cestě stromem. Konečně pro životnost lze použít formuli ve tvaru $A\Diamond\varphi$.

Následující popis se vztahuje pro formule MITL logiky a nástroj Uppaal od verze `4.1`. Pro síť časovaných automatů M je $\mathbb{P}_M(\varphi)$ specifikováno jako pravděpodobnost, že náhodný běh sítě M splňuje formuli φ. Kontrola vlastnosti $\mathbb{P}_M(\varphi) \geq p$, kde $p \in \langle 0, 1\rangle$ není možná, protože tento problém je nerozhodnutelný. Pro logiky, které využívají omezení hodin, lze

vyhodnocení $\mathbb{P}_M(\Diamond_{x \leq C}\ \texttt{ap})$, kde x jsou hodiny a C omezení hodin, aproximovat použitím algoritmu založeného na simulaci. Jeden z těchto algoritmů je právě SMC[2].

Otázky, které lze pomocí UPPAAL a SMC specifikovat jsou následující[2]:

1. Odhad pravděpodobnosti
   „Jaká je pravděpodobnost $\mathbb{P}_M(\Diamond_{x \leq C}\ \texttt{ap})$ pro síť časovaných automatů M?"

2. Testování hypotéz
   „Je pravděpodobnost $\mathbb{P}_M(\Diamond_{x \leq C}\ \texttt{ap})$ pro síť časovaných automatů M větší nebo rovna zadané minimální hodnotě $p \in \langle 0, 1 \rangle$?"

3. Porovnání pravděpodobností
   „Je pravděpodobnost $\mathbb{P}_M(\Diamond_{x_1 \leq C_1}\ \texttt{ap}_1)$ větší než pravděpodobnost $\mathbb{P}_M(\Diamond_{x_2 \leq C_2}\ \texttt{ap}_2)$?"

K vyřešení těchto otázek, nástroj UPPAAL kóduje běh systému do Bernoulliho náhodné proměnné, která je rovna hodnotě `pravda`, pokud daný běh splňuje specifikovanou vlastnost. V opačném případě je hodnota náhodné proměnné rovna `nepravda`. Pro odhad pravděpodobnosti využívá statistického algoritmu založeného na metodě Monte Carlo, zatímco pro otázky 2 a 3 využívá sekvenčního testování hypotéz[2].

Kapitola 2

Podrobnosti k řešenému problému

V této kapitole je blíže popsán nástroj Uppaal, který byl již vícekrát zmíněn v úvodu této práce. Podrobněji jsou popsány jednotlivé části tohoto nástroje, konkrétně návrh systému, simulace a verifikace. U návrhu systému je uveden jednoduchý příklad, který nese název „bridge“. Tento příklad je součástí distribuce nástroje Uppaal, a je tak vhodným příkladem pro popsání možností nástroje. V podkapitole verifikace jsou uvedeny jednotlivé typy dotazů, které lze pokládat k ověřování navrženého modelu. Tato podkapitola má úzkou vazbu na předchozí část textu – temporální logika. Ke konci popisu nástroje Uppaal jsou uvedeny některé další nástroje, které s Uppaal souvisí, například Stratego, Cora nebo Tron. Tyto nástroje buď upravují, nebo rozšiřují možnosti nástroje Uppaal.

Druhá podkapitola se zabývá popisem souborů, které jsou potřeba k překladu simulačního běhu do navržené uživatelsky přívětivé reprezentace. Prvním souborem je soubor ve formátu XML. V tomto souboru nástroj Uppaal ukládá navržené systémy. Dále soubor ve formátu XTR, ve kterém jsou uloženy simulační běhy, a konečně soubor ve formátu IF (Intermediate Formát), který je potřeba k porozumění obsahu souboru ve formátu XTR.

V poslední podkapitole je uvedená dostupná programová podpora pro práci s těmito soubory. Zejména je uvedena knihovna UTAP (Uppaal Timed Automata Parser Library) a její program `tracer`, který provádí překlad simulačních běhů do čitelnější podoby.

2.1 Nástroj Uppaal

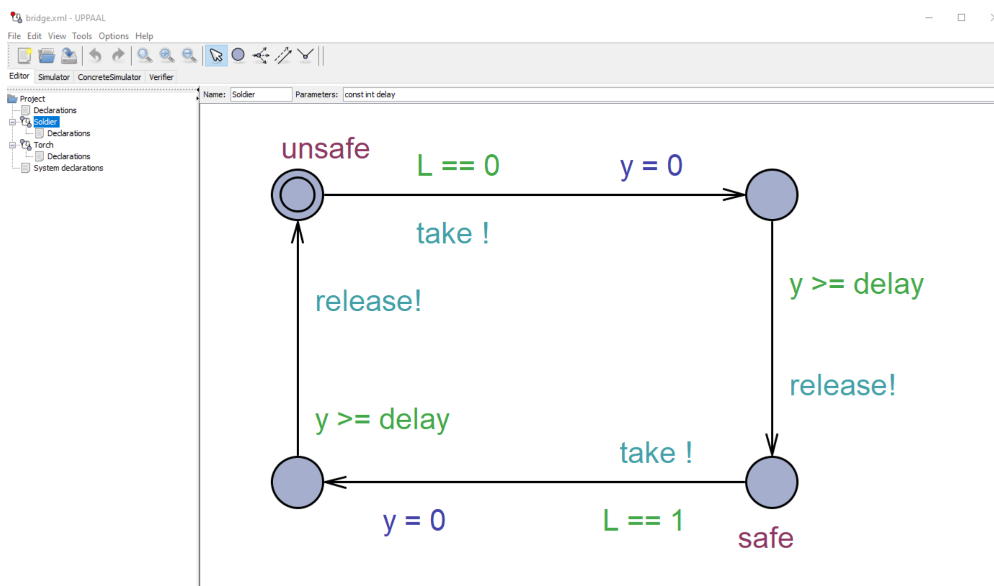
Jak již bylo zmíněno v úvodu, nástroj Uppaal slouží k modelování, simulaci, analýze a verifikaci systémů reálného času. Využívá k tomu časované automaty, které jsou propojovány do sítě. Tyto automaty jsou rozšířeny o datové typy, proměnné, programovatelné funkce a komunikační kanály. Komunikace mezi jednotlivými automaty probíhá pomocí synchronizačních zpráv po specifikovaných kanálech, případně pomocí globálně definovaných proměnných.

Uppaal se skládá ze tří hlavních části zmíněných výše: editor pro návrh systému, simulátor a verifikátor. Uživatelská aplikace je napsána v jazyce Java a „engine“ pro průchod stavového prostoru systému je napsaný v jazyce C++[5, 13].

Čas v systémech namodelovaných v nástroji Uppaal běží pouze ve stavech jednotlivých automatů. Při změně stavu využitím některé proveditelné hrany se hodnota hodin nezmění.

2.1.1 Návrh systému

Návrh systému se v nástroji Uppaal skládá z namodelování tzv. šablon (angl. „template“) představující jednotlivé složky systému. Tyto složky jsou samostatné časované automaty a finální systém, je pak spojením instancí těchto šablon, čímž je vytvořena síť časovaných automatů. Návrh těchto šablon lze v aplikaci provádět v záložce s názvem „Editor“. Příklad návrhu jedné šablony je uveden na obrázku 2.1. V levé části editoru je průzkumník souborů projektu. Na prvním místě v tomto průzkumníku je uveden soubor s názvem `Declarations`. V tomto souboru lze specifikovat globální proměnné, globální hodiny, případně i definovat vlastní funkce. Syntaxe použitá v tomto souboru je blízká jazyku C. V této práci tato syntaxe uvedena není, ale nástroj Uppaal obsahuje vestavěnou nápovědu, ve které ji lze nalézt. Vzápětí za souborem `Declarations` je seznam uživatelem navržených šablon. Každá šablona se skládá z grafického modelu a souboru `Declarations`. Obsah souboru `Declarations` je podobný prvnímu souboru s globálními definicemi. Tentokrát se ale definice vztahují pouze k dané šabloně, jedná se tedy o lokální definice. Každé šabloně lze také definovat vstupní parametry. Posledním souborem v průzkumníku souborů je `System declarations`. V tomto souboru jsou vytvářeny instance navržených šablon a sestaven finální systém. Pokud šablona obsahuje vstupní parametry, je nutné je v tomto souboru specifikovat.



Obrázek 2.1: Ukázka okna návrh systému v aplikaci Uppaal.

Hlavní částí `Editoru` je návrh modelu – časovaného automatu. Nástroj Uppaal umožňuje při návrhu používat různé typy stavů. Kromě běžných stavů lze použít i urgentní nebo tzv. `committed` stavy. Každý typ stavu je v Uppaal graficky odlišen od ostatních. Běžné stavy jsou reprezentovány pouze kolečkem, zatímco urgentní stavy mají uprostřed kolečka napsané velké U a `committed` stavy velké C. Iniciální stav automatu má navíc zdvojený okraj. Urgentní a `committed` stavy se od běžných vyznačují tím, že v nich neběží hodiny. Stavy `committed` navíc ovlivňují systém tím, že výstupní hrany z těchto stavů mají vyšší prioritu než ostatní hrany. Je-li nějaká instance šablony v `committed` stavu, pak jsou proveditelné pouze hrany, které z tohoto stavu vystupují (za předpokladu, že jsou splněny

podmínky samotného přechodu). Všem stavům v šabloně lze nastavit invariant. Ten zapříčiní to, že systém v tomto stavu může setrvat pouze po dobu, kdy daný invariant platí.

Na hranách v šabloně je v nástroji Uppaal umožněno specifikovat 3 typy vlastností: stráže (`guard`), synchronizace (`sync`) a aktualizace (`update`). Stráže obsahují podmínky, které musí platit, aby byla hrana proveditelná, např. hodnota proměnné je menší než zadaná hranice. Synchronizace může být dvojího typu, ale vždy probíhá na definovaném synchronizačním kanále. Pokud se jedná o naslouchání, následuje za jménem kanálu znak `?` (otazník). V případě, že jde o vysílání, je uveden znak `!` (vykřičník) za jménem kanálu. V aktualizacích jsou uvedeny všechny změny, které se mají po provedení hrany provést, např. vynulování hodin nebo změna hodnoty proměnné.

Hrany mohou být i násobné. Této funkcionality lze docílit použitím nastavení výběru (`select`). Při provádění přechodu se nedeterministicky vybere jedna hrana z dané množiny. Pomocí násobných hran se dají modelovat například stochastické systémy.

Kanály, které jsou použity v synchronizacích, slouží čistě k tomuto účelu. Nelze je použít k předávání dat. K tomu je potřeba definovat globální proměnné a data předávat přes ně. Kromě běžných kanálů, nástroj Uppaal obsahuje urgentní (`urgent`) a všesměrové (`broadcast`) kanály. Běžné kanály jsou specifické tím, že jsou blokující ve smyslu, pokud přijímač není schopen přijmout, pak ani vysílač nemůže vysílat. Při použití urgentních kanálů, synchronizace probíhá v okamžiku, kdy jsou splněny podmínky přechodu (`guard`). Všesměrové kanály jsou neblokující. Vysílač může vysílat kdykoliv, zatímco přijímač přijme, jen pokud je toho schopen. Tento typ kanálů je nutný pro modelování stochastických systémů. Zároveň nástroj podporuje specifikaci urgentních všesměrových kanálů.

V následující části této podkapitoly je podrobněji popsán příklad „bridge“, jak bylo zmíněno dříve. Příklad lze nalézt ve složce „demo“ distribuce nástroje Uppaal.

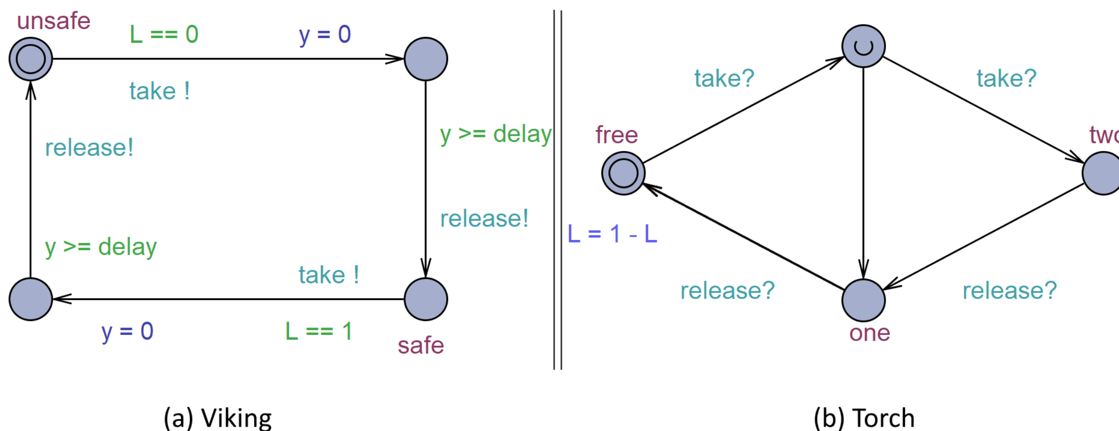
Příklad bridge

Tento příklad je reprezentací problému čtyř Vikingů a mostu, přes který musí přejít za určitý čas. Problém by se dal popsat následovně. Čtyři Vikingové se potřebují v noci dostat přes most. Každému Vikingovi cesta trvá jinak dlouho. První Viking dokáže most přejít za 5 minut, další Vikingové pak za 10, 20 a 25 minut. Most je ale poničený a obsahuje díry, a také neunese více jak dva Vikingy v jednom okamžiku. Vikingové disponují pouze jednou pochodní, která musí být nesena při přechodu přes most. Otázkou v tomto problému je: Existuje nějaké pořadí přechodů Vikingů přes most takové, aby se nezdrželi více jak jednu hodinu?

Systém, který tento příklad modeluje, je složen ze čtyř instancí šablony modelu Vikinga a jedné instance šablony modelu pochodně. Tyto modely lze vidět na obrázku 2.2. V globálních definicích jsou uvedeny hodiny `time`, synchronizační kanály `take` a `release` a proměnná `L`, která je datového typu `int[0,1]`, tedy ekvivalent booleovské proměnné. Hodiny `time` slouží k měření celkového času běhu systému. Kanály jsou použity k synchronizaci mezi Vikingy a pochodní, kde `take` představuje získání pochodně a `release` uvolnění pochodně. Proměnná `L` udává, na které straně mostu se nachází pochodeň.

Šablona pro Vikinga navíc v lokálních definicích obsahuje hodiny `y`, které slouží k měření doby přechodu přes most. Vstupním parametrem této šablony je konstanta `delay`, která udává, jak dlouho se Viking na cestě přes most zdrží. Šablona pro pochodeň neobsahuje žádné lokální definice ani vstupní parametry.

Iniciálními stavy pro dané modely jsou `unsafe` pro Vikinga a `free` pro pochodeň, což reprezentuje situaci, ve které žádný Viking nepřešel do bezpečí, a nikdo nedrží pochodeň.



Obrázek 2.2: Šablony nástroje Uppaal použité v příkladu „bridge“.

Stráž přechodu ze stavu `unsafe` obsahuje podmínku $L == 0$, tedy, že se pochodeň nachází na straně mostu, kde není bezpečno. Jakmile je tato stráž splněna, Viking může přejít do následujícího stavu (na obrázku 2.2 stav vpravo nahoře). Při této akci dojde k synchronizační zprávě po kanále `take` a vynulují se lokální hodiny `y`. Pochodeň přijme synchronizaci po kanále `take` a přejde do dalšího stavu, který je urgentní. Pokud další Viking vezme pochodeň a proběhne další synchronizace po kanále `take`, pochodeň přejde do stavu `two`. Jinak pochodeň přejde do stavu `one`, který reprezentuje, že pochodeň drží pouze jeden Viking. Protože je stav po prvním přijetí synchronizace po kanále `take` urgentní, neběží v tomto stavu čas, a sebrání pochodně je jak pro jednoho, tak dva Vikingy stejně dlouhé.

Jeden nebo dva Vikingové zůstanou ve stavu mezi `unsafe` a `safe` alespoň tak dlouho, jak je specifikován jejich vstupní parametr `delay` (stráž $y >= delay$ na dalším přechodě), a poté přejdou do stavu `safe`. Při provedení přechodu provedou synchronizaci na kanále `release`, čímž se model pochodně dostane opět do počátečního stavu. Těsně před přechodem do počátečního stavu se aktualizuje hodnota globální proměnné L na opačnou hodnotu, což reprezentuje, že pochodeň je na opačně straně mostu.

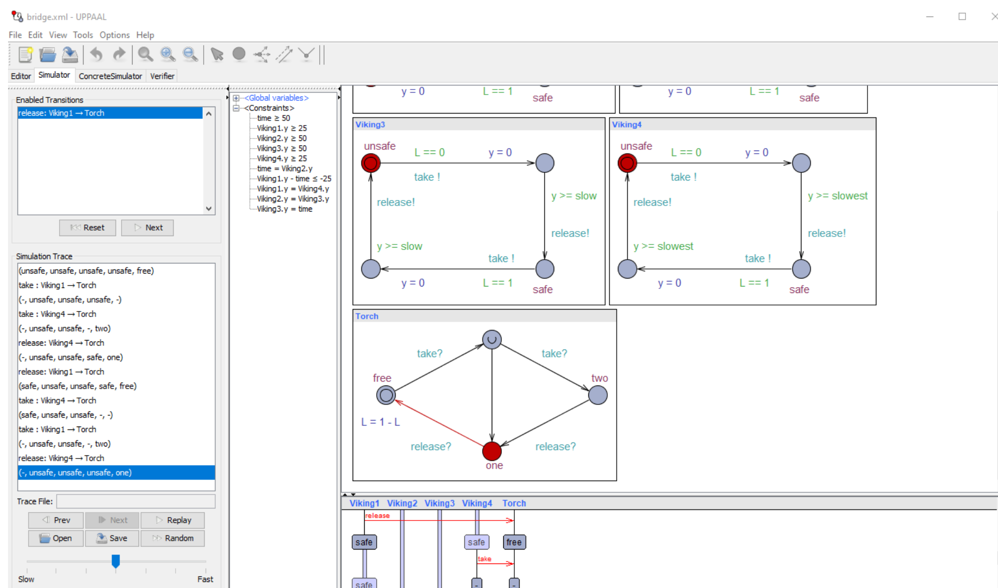
Opačná cesta (ze stavu `safe` do `unsafe`) probíhá téměř stejně, jen je hodnota proměnné `L` kontrolována na hodnotu 1, protože se pochodeň nachází na druhé straně mostu.

V následujících částech (simulace a verifikace) je uvedeno několik dalších zmínek o tomto příkladu. V části verifikace je také uveden příklad dotazu na systém, který odpovídá otázce problému čtyř Vikingů a mostu.

2.1.2 Simulace

Simulace se v nástroji `Uppaal` provádí v záložce s názvem „Simulator“. Na obrázku 2.3 je zobrazena ukázka simulace systému pro příklad „bridge“ z předchozí podkapitoly. Simulátor je rozdělen na celkem 4 části. V levé části je ovládání simulace, kde jsou nahoře uvedeny všechny proveditelné hrany a výběrem jedné z nich, lze danou hranu provést. Pod sekcí výběru dalšího přechodu v simulaci je uvedena historie. V této historii se dá navigovat a případně i pokračovat od zobrazeného stavu. Pod historii je ovládání rychlosti simulace spolu s možnostmi načíst uložený simulační běh, uložit právě prováděný běh či zapnutí automatické simulace, při které se další hrana vybírá náhodně.

V prostřední části jsou uvedeny hodnoty globálních proměnných a omezení nad hodinami v systému. V pravé části jsou zobrazeny jednotlivé instance šablon, které jsou v sys-



Obrázek 2.3: Ukázka simulace v aplikaci Uppaal

tému definovány. Aktuální stav je v každé šabloně zvýrazněn červeně. V poslední části je uveden graf synchronizací procesů pomocí kanálů.

Konkrétní simulace

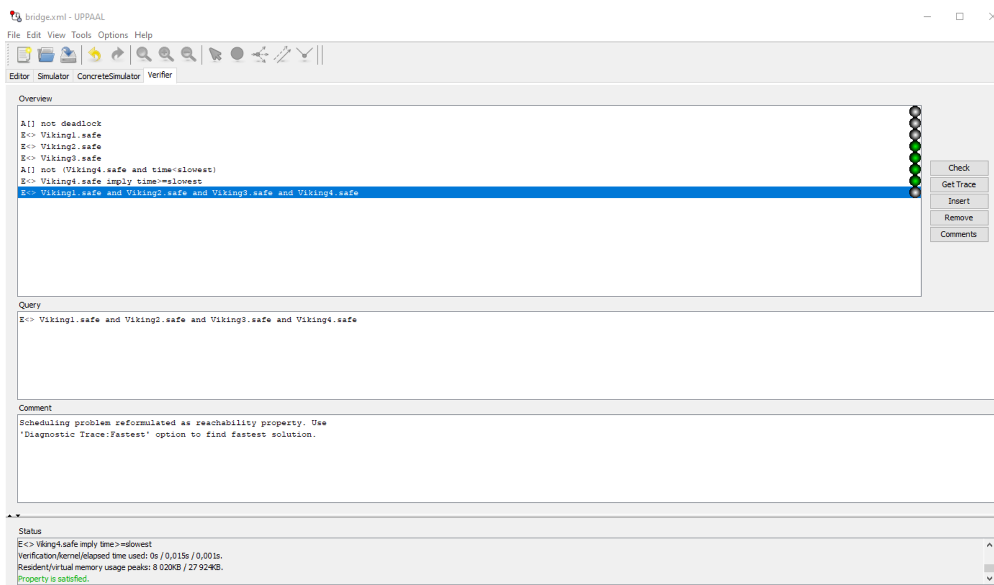
Novější verze nástroje Uppaal obsahují kromě klasického simulátoru i tzv. „konkrétní simulátor“. Tento simulátor byl původně součástí nástroje Uppaal-Tiga[2]. Na rozdíl od klasického simulátoru, uživatel může specifikovat, jak dlouho je systém v daném stavu. Opět lze využít automatické simulace, ale vytvořené běhy nelze ukládat.

2.1.3 Verifikace

Verifikace se v nástroji Uppaal provádí v záložce „Verifier“. Tato záložka je ukázána na obrázku 2.4. V horní části lze specifikovat dotazy na systém v syntaxi, která je popsána a vysvětlena níže v této podkapitole. Jedná se o seznam definovaných dotazů, které se dají ověřovat. Samotné napsání dotazu se provádí v části pod tímto seznamem, označené jako „Query“. Ke každému dotazu lze také přidat komentář v sekci „Comment“. Poslední částí této záložky je čistě informační pole „Status“, ve kterém jsou vypisovány aktuální stavy aplikace, například který dotaz se právě aplikace snaží ověřit.

Po výběru a označení dotazu lze tlačítkem s názvem „Check“ zahájit ověřování tohoto dotazu. Pokud je výsledek dotazu pozitivní, šedé kolečko napravo od dotazu zezelená. Naopak, pokud je výsledek dotazu negativní, kolečko zčervená. Kolečko může i zežloutnout, pokud je v nastavení zvolena možnost `Under Approximation` nebo `Over Approximation` a verifikační nástroj není přesvědčen o výsledku dotazu[5]. Pro dotazy MITL logiky je vedle kolečka vypsán i interval, ve kterém je výsledná pravděpodobnost specifikovaného dotazu.

Pro formule TCTL logiky je syntaxe dotazů velice podobná matematickému zápisu. Jedinou změnou je nahrazení znaku $\Box$ za `[]` a nahrazení znaku $\Diamond$ za `<>`. Jak již bylo zmíněno v úvodu, verifikační nástroj pro tento typ dotazů prochází stavový prostor a ověřuje, zda po



Obrázek 2.4: Ukázka verifikace v aplikaci UPPAAL

cestě ve stromě platí jednotlivé části dotazu. Následující výčet popisuje význam jednotlivých dotazů v syntaxi, kterou nástroj UPPAAL podporuje.

- `A[]` φ: pro všechny cesty stromem stavového prostoru vždy platí φ
- `A<>` φ: pro všechny cesty stromem stavového prostoru někdy platí φ
- `E[]` φ: existuje min. jedna cesta stromem stavového prostoru, ve které vždy platí φ
- `E<>` φ: existuje min. jedna cesta stromem stavového prostoru, ve které někdy platí φ
- φ `->` ψ: pokud platí φ, pak někdy platí ψ

kde φ a ψ jsou formule, které mohou být ověřeny v daném stavu, např. podmínka, že proměnná není větší než specifikovaná hodnota. Speciálním typem dotazu je `A[] not deadlock`. Tento dotaz ověřuje, že pro všechny cesty stromem stavového prostoru je systém vždy bez uváznutí[13]. Dotaz, který by se ptal na otázku z příklad „bridge“, zmíněného výše, by vypadal následovně:

$$\text{E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe and time} <= 60$$

Pro formule MITL logiky a dotazovaní se na tři typy dotazů uvedených v úvodu (odhad pravděpodobnosti, testování hypotéz a porovnání pravděpodobností), nástroj UPPAAL zavádí následující syntaxi. U každého typu dotazu je také uveden jeho popis a možný příklad.

Odhad pravděpodobnosti: `Pr[bound](<>` φ`)`

Dotaz pro zjištění pravděpodobnosti, že φ nastane před `bound` časových jednotek[2]. Možný příklad pro systém z příkladu „bridge“ by mohlo vypadat následovně:

$$\text{Pr[<=30](<> Viking1.safe)}$$

Testování hypotéz: `Pr[bound](<>` φ`) >=` p

Dotaz pro ověření, zda pravděpodobnost, že φ nastane před `bound` časových jednotek, je větší nebo rovno zadané hranici p. V tomto typu dotazu, může mít část `bound` jednu z následujících podob: 1) `<=`d, 2) x`<=`d nebo 3) `#<=`d, kde $d \in \mathbb{N}$ a x jsou hodiny. První typ pouze specifikuje časové omezení, zatímco druhý přidává možnost omezení podle ceny. Ve třetím případě se jedná o omezení podle počtu diskrétních kroků v simulaci[2]. Možný příklad pro systém z příkladu „bridge“ by mohlo vypadat následovně:

Pr[<=30](<> Viking1.safe) >= 0.2

Porovnání pravděpodobností: `Pr[bound1](<>` φ`1) >= Pr[bound2](<>` φ`2)`

Dotaz pro ověření, že pravděpodobnost toho, že $\varphi 1$ nastane před `bound1` časových jednotek, je větší nebo rovno pravděpodobnosti, že $\varphi 2$ nastane před `bound2` časových jednotek. Možný příklad pro systém z příkladu „bridge“ by mohlo vypadat následovně:

Pr[<=30](<> Viking1.safe) >= Pr[<=25](<> Viking2.safe)

Tyto typy dotazů fungují jen nad systémy, ve kterých jsou všechny synchronizační kanály všesměrové (`broadcast`). Uvedené příklady pro systém z příkladu „bridge“ jsou tedy čistě ilustrační, neboť tento systém nemá všesměrové kanály.

Další typ dotazu, který nástroj Uppaal podporuje má následující formát:

$$\text{simulate N [bound] } \{\varphi_1, \ldots, \varphi_k\}$$

Tento dotaz dává uživateli možnost zjistit další podrobnější informace o navrženém systému. Dotaz provede `N` simulací, kde `N` je přirozené číslo, do maximálního času omezeného částí `bound`, a poté vytvoří grafický výstup v podobě grafu. V tomto grafu jsou vizualizovány monitorované vlastnosti, které byly specifikovány v části $\{\varphi_1, \ldots, \varphi_k\}$[2].

Pod tlačítkem s názvem „Check“ je tlačítko označené „Get Trace“. Jeho zmáčknutím si lze nechat vygenerovat simulační běh, který například popisuje, proč výsledek dotazu není pozitivní. Tento běh lze otevřít v simulátoru a ručně jej procházet. Ze simulátoru jej lze také uložit. Touto akcí vznikne soubor ve formátu XTR, který je popsán dále v této práci.

Součástí distribuce nástroje Uppaal je program `verifyta`. Tento nástroj má více využití. Jedním z využití je získání souboru ve formátu IF, který je nutný pro překlad souboru XTR do čitelné podoby. Formát IF je také popsán v další části tohoto textu. Druhým využitím programu `verifyta` je ověřování dotazů, které funguje stejně jako vestavěné ověřování. Výhodou použití tohoto programu je možnost ověřovat dotazy na vzdáleném počítači. Toto je obzvlášť výhodné, pokud je potřeba ověřit složité dotazy nad rozsáhlými systémy, kdy je potřeba využít vzdálený počítač s větší operační pamětí[5].

2.1.4 Nástroje související s Uppaal

V této podkapitole jsou uvedeny související nástroje s Uppaal, které jsou uvedeny na stránce nástroje Uppaal. Jedná se o Stratego, Cora, Tron, Tiga, Cover a Port. U každého nástroje je krátce popsáno, jak se tento nástroj liší od Uppaal nebo jak s ním souvisí.

Stratego

Uppaal Stratego kombinuje různé techniky analýzy z různých odvětví nástroje Uppaal a zaměřuje se především na strategie. Tento nástroj usnadňuje především generování, optimalizaci, srovnání a zkoumání důsledků a výkonu strategií pro stochasticky hodnocené časované hry. Dovoluje také efektivně a flexibilně prohledávat stavový prostor strategií[18].

Cora

Uppaal Cora je varianta nástroje Uppaal, která se zaměřuje na optimalizaci ceny cesty stromem stavového prostoru, aby byly splněny cílové podmínky. Na rozdíl od nástroje Uppaal, který používá časované automaty, Uppaal Cora používá variantu časovaných automatů, kterým se říká LPTA. Tyto automaty dovolují přiřadit modelu představu o ceně[15].

Tron

Uppaal Tron je testovací nástroj založený na nástroji Uppaal. Používá se k testování shody časovaných systémů, které vidí jako tzv. „black-box“. Zaměřuje se především na vestavěný software, který lze často nalézt v různých řadičích[14].

Tiga

Uppaal Tiga je rozšíření nástroje Uppaal. Implementuje efektivní „on-the-fly“ algoritmus, který je schopen řešit hry založené na časovaných automatech s ohledem na dosažitelnost řešení a omezující podmínky[16].

Cover

Uppaal CoVer je nástroj, který umožňuje vytvářet testovací sady pro modely vytvořené v nástroji Uppaal. Které vlastnosti modelu se mají testovat je určeno pomocí tzv. „pozorovatelů“, např. `observer automata`[6].

Port

Uppaal Port je varianta nástroje Uppaal, která se zaměřuje na simulaci a verifikaci systémů, které lze popsat jako komponenty systému reálného času[7].

2.2 Používané typy souborů

V následující části této kapitoly jsou popsány soubory XML, XTR a IF, které byly v předchozí části textu několikrát zmíněny. Formát XML je použit pro uložení navrženého systému, soubor ve formátu XTR obsahuje uložený simulační běh a soubor ve formátu IF obsahuje informace potřebné pro překlad XTR do čitelné podoby. XTR je sice textový formát a lze jej otevřít v jednoduchém textovém editoru, jak ale bude ukázáno dále, bez použití obsahu souboru IF, nelze o simulačním běhu nic říct. Jak již bylo zmíněno výše, soubor IF lze vygenerovat pomocí nástroje `verifyta`, který je součástí distribuce nástroje Uppaal.

2.2.1 XML - uložený systém

Struktura XML souboru, dovolených elementů a atributů těchto elementů, je přesně specifikována v DTD souboru. Odkaz na tento soubor je uveden v hlavičce každého XML souboru, který nástroj Uppaal vytvoří. Ukázka struktury XML souboru je uvedena v příloze A. Tato ukázka obsahuje namodelovaný systém pro příklad „bridge“, je však značně zkrácena. Místo „. . . “ soubor obsahuje konkrétní elementy, které byly v ukázce už představeny. Jak je vidět, struktura souboru se velice podobá té z průzkumníku projektu v záložce „Editor“, která byla popsána v podkapitole o návrhu systému. Následující text obsahuje pouze zkrácený popis obsahu tohoto souboru. Nejsou uvedeny všechny elementy a všechny atributy, ale jsou popsány podstatné části, které jsou důležité pro překlad simulačního běhu do navržené reprezentace.

Kromě kódování a specifikování DTD souboru, je celý obsah v elementu `nta`. Autorovi práce se nepodařilo dohledat, co přesně tato zkratka znamená. Velmi pravděpodobně se ale jedná o „network of timed automata“, neboli „síť časovaných automatů“. Prvním synem elementu `nta` je element `declaration`. V tomto elementu jsou uloženy všechny globální definice tak, jak byly uvedeny při návrhu systému. Za elementem `declaration` následují všechny šablony, každá ve vlastním elementu `template`. Na konci souboru je uveden element `system`, ve kterém je vložena definice systému, a za tímto elementem následuje poslední element `queries`, ve kterém jsou definovány všechny dotazy uvedené v záložce „Verifier“.

Každá šablona obsahuje element `name`, ve kterém je uloženo jméno šablony. Dále maximálně jedenkrát elementy `parametr`, `declaration` a `init`. V elementu `parametr` jsou uloženy vstupní parametry šablony, v `declaration` jsou uloženy lokální definice a v `init` je odkaz na počáteční stav v této šabloně. Dále každá šablona obsahuje libovolné množství elementů `location` a `transition`, které obsahují informace o stavech a přechodech v časovaném automatu. Konkrétně jméno a specifikovaný invariant pro stavy a stráže, synchronizace a aktualizace pro přechody.

2.2.2 XTR - uložený simulační běh

XTR je strukturovaný textový formát, který nástroj Uppaal používá k ukládání simulačních běhů. Jeho obsahem je kolekce záznamů stavů systému při simulaci. Prvním záznamem je počáteční stav, za kterým následuje sekvence záznamů reprezentující další stavy systému. Tyto záznamy obsahují kromě popisu stavu i seznam přechodů, které byly provedeny, aby se systém dostal do tohoto stavu. První záznam (počáteční stav) tedy seznam přechodů neobsahuje. V příloze B je ukázán a popsán jeden ze záznamů simulačního běhu, který byl získán simulací dříve zmíněného systému – příklad „bridge“. Konkrétně se jedná o stav systému, ve kterém druhý Viking sebral pochodeň a přešel do stavu mezi `unsafe` a `safe` (na obrázku 2.2 (a) stav vpravo nahoře). Popisky jsou v této ukázce rozděleny do sekcí, které budou popsány dále. Původní obsah souboru XTR obsahuje pouze čísla, tečky a od novější verze nástroje `Uppaal` i středníky.

Struktura souboru XTR by se dala popsat následující gramatikou:

$$\begin{aligned}\langle trace\rangle \quad = \quad & \langle state\rangle\, (\langle state\rangle\, \langle transition\rangle)^*\, \langle nl\rangle\, \langle dot\rangle \\ & \langle state\rangle = \langle locations\rangle\, \langle zone\rangle\, \langle variables\rangle \\ & \langle locations\rangle = (\langle location\rangle\, \langle nl\rangle)^*\, \langle dot\rangle\, \langle nl\rangle \\ & \langle zone\rangle = (\langle clock\rangle\, \langle nl\rangle\, \langle clock\rangle\, \langle nl\rangle\, \langle bound\rangle\, \langle dot\rangle\, \langle nl\rangle)^*\, \langle dot\rangle\, \langle nl\rangle \\ & \langle variables\rangle = (\langle NUM\rangle\, \langle nl\rangle)^*\, \langle dot\rangle\, \langle nl\rangle \\ & \langle transition\rangle = ((\langle process\rangle\, \langle spc\rangle\, \langle edge\rangle\, (\langle spc\rangle\, \langle select\rangle)^*\, \langle sem\rangle\, \langle nl\rangle)^+\, \langle dot\rangle\, \langle nl\rangle\end{aligned}$$

kde $\langle trace\rangle$ je finální simulační běh. $\langle state\rangle$ reprezentuje stav a $\langle transition\rangle$ provedené přechody. $\langle locations\rangle$ je vektor míst systému, kde $\langle location\rangle$ je číselná hodnota odpovídající identifikaci místa, $\langle nl\rangle$ je znak nového řádku a $\langle dot\rangle$ je znak `.` (tečka). $\langle zone\rangle$ je sekce „zóna", ve které jsou uvedena omezení hodin, kde $\langle clock\rangle$ je identifikace hodin a $\langle bound\rangle$ je jejich omezení. $\langle variables\rangle$ je, podobně jako vektor míst, vektor proměnných. $\langle transition\rangle$ se skládá z jednoho či více provedených přechodů, kde $\langle process\rangle$ je identifikace procesu, $\langle edge\rangle$ je identifikace hrany a $\langle select\rangle$ je index v případě, že byla použita násobná hrana. $\langle spc\rangle$ reprezentuje mezeru a $\langle sem\rangle$ je znak `;` (středník).

Jak již bylo naznačeno, každý záznam by se dal rozdělit na čtyři části[8]. Tyto části jsou odděleny znakem tečka. V první části je seznam míst procesů systému, každé na vlastním řádku. Tato část je seřazena podle identifikačního čísla procesu a samotné číslo uvedené v souboru je identifikací místa, ve kterém se proces právě nachází. Další částí je sekce tzv. „zóna"[8]. V ní jsou uvedeny omezení nad hodinami, přičemž každé omezení je složeno ze tří čísel (opět každé na vlastním řádku). Formát tohoto omezení je následující:

$$x - y < d \text{ nebo } x - y \leq d$$

kde x a y jsou hodiny a $d \in \mathbb{N}$. První dvě čísla jsou identifikace hodin x a y a třetí číslo je hodnota d. Do třetího čísla je také zakódována informace jestli se jedná o striktní porovnání, či nikoliv. Toto zakódování je provedeno bitovým posuvem hodnoty doprava o jednu pozici a nastavením nejméně významného bit na hodnotu `1`, pokud se jedná o striktní omezení. V opačném případě je hodnota nejméně významného bitu rovna `0`. Každé omezení je odděleno řádkem, na kterém je pouze tečka. Třetí část obsahuje hodnoty proměnných definovaných v systému. Podobně jako seznam míst, je tato sekce seřazena podle identifikačního čísla proměnné a každé číslo je uvedeno na vlastním řádku. Toto číslo odpovídá hodnotě dané proměnné. V poslední části záznamu v simulačním běhu jsou uvedeny přechody, které byly provedeny, aby se systém do tohoto nového stavu dostal, přičemž iniciální stav tuto část neobsahuje. Každý přechod, je uveden na vlastním řádku, je složen z minimálně dvou čísel a je ukončen středníkem. První číslo odpovídá procesu, který daný přechod provedl, a druhé je identifikační číslo hrany v navrženém modelu. Pokud jsou na řádku uvedeny další čísla, jedná se o indexy, které byly použity u vícenásobných hran. Konec souboru je označen tečkou na novém řádku.

Tento popis XTR je platný pro simulační běhy vytvořené v Uppaal ve verzi `4.1.24`. Verze `4.0` neobsahuje na konci popisu přechodu středníky a identifikační číslo hran je o 1 větší než v nové verzi.

2.2.3 IF - pomocný soubor pro překlad

V popisu formát XTR bylo vícekrát zmíněno identifikační číslo některé části modelu. Tyto identifikace jsou uvedeny v souboru IF. V této podkapitole je soubor IF podrobně popsán a je uveden i způsob, jak tento soubor z navrženého systému získat.

Formát IF je stejně jako formát XTR textový. Jeho obsahem jsou tabulky objektů z navrženého modelu. Ukázka zkráceného obsahu souboru IF je uvedena v příloze C. Ukázka byla vytvořena ze systému pro příklad „bridge“ a obsahuje začátky všech tabulek, které se v souboru IF vyskytují. Každá tabulka má pod názvem uveden jeden či více řádků začínající znakem #. Jedná se o komentáře, které obsahují popis syntaxe jednotlivých záznamů v dané tabulce. Následující výčet obsahuje jména všech tabulek.

- `layout`
- `instructions`
- `processes`
- `locations`
- `edges`
- `expressions`

Tabulka `layout` obsahuje konkrétní objekty použité v návrhu modelu. Jsou zde uvedeny názvy a hodnoty konstant, jména hodin, datové typy a jména proměnných, jména míst, meta informace a ceny. Každý záznam této tabulky začíná identifikačním číslem. Například pro místa je toto číslo použito v první části záznamu v XTR souboru. Pro konstanty, které jsou označeny `const`, je uvedena jejich hodnota (`value`). U hodin, je kromě jejich jména (`name`) uvedeno i číslo `nr`. Číslo `nr` je oddělené identifikační číslo, které se inkrementuje pouze pro hodiny. Proměnné a meta informace obsahují kromě názvu a čísla `nr`, které je opět oddělené pro jednotlivé typy záznamů, minimální (`min`), maximální (`max`) a počáteční (`init`) hodnotu. Minimální a maximální hodnota je dána použitým datovým typem. U míst (`location`), je kromě jména uveden i `flag`. Ten obsahuje informaci o tom, jestli je dané místo urgentní nebo `committed`. Jednotlivé části záznamů jsou odděleny dvojtečkou, a pokud některá část chybí, je použita výchozí hodnota.

Další tabulkou je `instructions`. V této tabulce jsou v podobě jazyka Assembler vypsány instrukce, které popisují výpočet výrazů. Tabulka obsahuje i operační kód uživatelem definovaných funkcí[8].

Třetí tabulka, s názvem `processes`, obsahuje výčet všech procesů, které jsou v systému použity. Každý záznam se skládá z identifikačního čísla `index`, odkazu na počáteční stav `initial` a jména. Například pro proces s identifikačním číslem `0` z ukázky v příloze C, záznam obsahuje jméno „Viking1“ a iniciálním stavem je stav s indexem 15. V tabulce `layout` lze vidět, že stav s indexem 15 má jméno „unsafe“. Iniciálním stavem pro proces prvního Vikinga je tedy stav `unsafe`.

Tabulka `locations` obsahuje podrobnější informace o místech v systému. Každý záznam začíná číslem `index`, který je odkazem do tabulky `layout`. Dále tato tabulka přiřazuje k místu proces, jehož identifikační číslo je uvedeno v části `process`. Poslední částí záznamu (`invariant`) je odkaz do tabulky `expressions`, která bude popsána dále, a reprezentuje výraz invariantu daného místa.

Následující tabulka `edges` obsahuje seznam všech hran v systému. U každé hrany je uvedeno číslo procesu, ke kterému se hrana váže, počáteční místo (`source`) a koncové místo (`target`), mezi kterými je daná hrana namodelována. Číslo procesu odkazuje do tabulky `processes` a počáteční a koncové místo jsou indexy v tabulce `layout`. Dále záznamy obsahují odkazy do tabulky `expressions`, konkrétně vyraz pro stráže (`guard`), synchronizaci (`sync`) a aktualizaci (`update`).

Poslední tabulka `expressions` obsahuje všechny výrazy v textové podobě, které byly v modelu uvedeny. Jedná se především o výše zmíněné invarianty, stráže, synchronizace a aktualizace. U každého záznamu je kromě textu uvedeno jeho identifikační číslo, na které se ostatní tabulky odkazují, a pole `reads` a `writes`. Tato pole obsahují indexy do tabulky `layout` a uvádí, které proměnné, konstanty nebo meta informace se načítají, a do kterých proměnných nebo meta informací se zapisuje.

Soubor ve formátu IF lze z navrženého systému vygenerovat programem `verifyta`, který byl představen dříve. Tento program se nachází ve složkách `bin-Linux` a `bin-Windows` distribuce nástroje UPPAAL. Před použitím je nutné nastavit proměnnou prostředí `UPPAAL_COMPILE_ONLY` na hodnotu `1`. Poté lze vygenerovat soubor IF například následujícím příkazem:

```
> bin-Windows\verifyta.exe demo\bridge.xml >bridge.if
```

kde `demo\bridge.xml` je cesta k uloženému navrženému modelu. U verze `4.0` nástroje UPPAAL je nutné specifikovat i cestu k souboru s koncovkou „.q“, ve kterém jsou uvedeny dotazy pro verifikaci.

2.3 Dostupná programová podpora

V době psaní tohoto textu, ke znalosti autora, existují pouze dva nástroje, které pracují s formáty XTR a IF. Prvním nástrojem je `UPPAAL2OCTOPUS`, který je dílem instituce ESI (Embedded System Institute) v Nizozemsku. Tento nástroj je napsaný v jazyce C++ a překládá simulační běhy nástroje UPPAAL do formátu `octopus`. K překladu vyžaduje soubory XTR a IF, protože, jak bylo uvedeno, pouze ze souboru XTR nelze získat konkrétní informace o simulačním běhu. Formát `octopus` je dále používán v aplikaci `ResVis`[4].

Druhým nástrojem je knihovna UTAP (Uppaal Timed Automata Parser Library). Tato knihovna obsahuje programy pro analýzu systémů namodelovaných v nástroji UPPAAL[19]. Knihovna je distribuována pod licencí LGPL a nástroj UPPAAL ji také používá[19]. Všechny tři formáty, se kterými `Uppaal` umí pracovat, jsou podporovány. Knihovna je napsána v jazyce C++ a obsahuje hlavičkové soubory pro práci s `Uppaal` modely. Například jsou v ní hlavičkové soubory s třídami pro sestavení systému nebo ověření typů. Pro účely tohoto dokumentu je ale nejdůležitější program `tracer`.

Program `tracer` překládá simulační běhy ve formátu XTR, za pomocí souboru ve formátu IF, do uživatelsky čitelné podoby. Tento program, na rozdíl od většiny v knihovně, nepoužívá další části (struktury, funkce, ...) knihovny. Program je přeložitelný pomocí standardního překladače jazyka C++, za použití normy alespoň C++11. Následující část obsahuje ukázku výstupu programu `tracer` pro začátek simulačního běhu demonstračního příkladu „bridge“.

```
State: Viking1.unsafe Viking2.unsafe Viking3.unsafe Viking4.unsafe Torch.
    free L=0 t(0)-time<=0 t(0)-Viking1.y<=0 t(0)-Viking2.y<=0 t(0)-Viking3.
    y<=0 t(0)-Viking4.y<=0 time-Viking1.y<=0 Viking1.y-Viking2.y<=0 Viking2
    .y-Viking3.y<=0 Viking3.y-Viking4.y<=0 Viking4.y-time<=0

Transition: Viking2.unsafe -> Viking2._id0 {L == 0; take!; y = 0;} Torch.
    free -> Torch._id5 {1; take?; 1;}
```

```
State: Viking1.unsafe Viking2._id0 Viking3.unsafe Viking4.unsafe Torch._id5
     L=0 t(0)-time<=0 t(0)-Viking1.y<=0 t(0)-Viking2.y<=0 t(0)-Viking3.y<=0
     t(0)-Viking4.y<=0 time-Viking1.y<=0 Viking1.y-Viking3.y<=0 Viking2.y-t
    (0)<=0 Viking3.y-Viking4.y<=0 Viking4.y-time<=0
...
```

Tento výstup je strukturovaný podobně jako soubor ve formátu XTR. Na začátku je popsán počáteční stav systému, a poté se opakuje dvojice (přechod, stav), kde přechod uvádí, které hrany byly v systému provedeny, aby se systém dostal do nového stavu. Každý stav a přechod je popsán na vlastním řádku. Oproti formátu XTR doplňuje namísto čísel informace získané ze souboru IF a upravuje odřádkování.

Výstup programu `tracer` je podstatně čitelnější než formát XTR. Na druhou stranu není moc uživatelsky přívětivý. V následující kapitole je, spolu s popisem implementace aplikace pro překlad, představena navržená uživatelsky přívětivá reprezentace simulačních běhů nástroje UPPAAL.

Kapitola 3

Návrh a implementace

Jak bylo zmíněno v předchozí kapitole, tato kapitola obsahuje popis navržené uživatelsky přívětivé reprezentace simulačních běhů nástroje Uppaal. Jsou popsány a vysvětleny jednotlivé části této reprezentace spolu s vysvětlením, proč jsou některé části navrženy tímto způsobem. Za popisem navržené reprezentace je uveden seznam použitých prostředků. Je uvedeno i k čemu přesně jsou využívány, a pro zvolený programovací jazyk i návod, jak nainstalovat překladač.

Dále je popsán návrh aplikace pro překlad simulačního běhu ve formátu XTR do navržené reprezentace. Implementace této aplikace je popsána ve dvou podkapitolách. První se zabývá samotným překladem a druhá popisem filtrace simulačního běhu. U překladu je uvedeno zpracování jednotlivých souborů a vytvoření výstupu. Popis filtrace obsahuje také všechny možnosti filtrace daného simulačního běhu.

Na konci této kapitoly jsou uvedeny některé problémy, které bylo potřeba řešit. U každého je také uvedeno jeho řešení v této implementaci a vysvětlení, proč bylo použito právě toto řešení.

3.1 Navržená reprezentace simulačních běhů

Hlavním cílem při návrhu uživatelsky přívětivé reprezentace simulačních běhů byla dobrá čitelnost pro uživatele, a to i v případě velmi rozsáhlých simulačních běhů. Druhým cílem bylo navrhnout reprezentaci tak, aby jí bylo možno popsat i běhy, které jsou nějakým způsobem zkráceny, například pouze stavy či přechody, nebo jen části, které mají hodnotu jisté proměnné ve zvoleném rozsahu. S ohledem na tyto dva cíle je navržená uživatelsky přívětivá reprezentace strukturována jako JSON řetězec, ve kterém jsou odděleny popisy objektů reprezentující stavy v simulaci a objekty reprezentující přechody. V příloze D je uvedena struktura navržené reprezentace pro výchozí skládání objektů. Následující část této podkapitoly popisuje jednotlivé části této struktury a jsou uvedeny i další možnosti, jak objekty pro stavy a přechody poskládat.

Jak bylo popsáno v části o formátu XTR, nástroj Uppaal vytváří simulační běh tak, že nejdříve zaznamená počáteční stav systému, a následně opakuje zaznamenání nového stavu a přechodů, které vedly do tohoto nového stavu. Navržená reprezentace tohoto faktu využívá a jednotlivé objekty čísluje globálním identifikačním číslem. První (počáteční) stav systému má vždy identifikační číslo rovno 0. Ostatní stavy jsou označeny sudými čísly a přechody mají lichá identifikační čísla. Výchozí poskládání objektů popisující stavy a přechody, které je znázorněno i v příloze, je uvedení prvně všech stavů a poté všech přechodů. Toto

poskládání má velkou výhodu v případech, ve kterých uživatele zajímají stavy systému, které nastaly, a už ho méně zajímají všechny přechody, které byly mezi stavy provedeny. Zároveň však přechody nejsou ze simulačního běhu odebrány a pomocí identifikačních čísel lze dohledat přesně ten objekt popisující provedené přechody, který patří mezi dva stavy systému. Toto poskládání však není vhodné ve všech případech. Na tento fakt bylo při návrhu reprezentace myšleno, a proto jsou představeny i následující dva typy poskládání objektů. První typ skládá objekty do posloupnosti, která odpovídá samotné simulaci. Tedy za uvedením počátečního stavu následuje objekt, který popisuje provedené přechody do následujícího stavu, za kterým následuje objekt popisující tento nový stav. S tím, že se dvojice (přechody, stav) dále opakuje. Při zkoumání pouze identifikačních čísel popsaných v úvodu odstavce, tento typ skládání vytváří seřazenou posloupnost. Druhý typ skládá objekty podobně, jen je prohozena dvojice (přechody, stav) na (stav, přechody). Pro lepší navigaci v rámci rozsáhlého simulačního běhu je v tomto typu skládání upraveno označení identifikačními čísly, tak aby i pro tento typ tvořilo seřazenou posloupnost. Nové stavy mají tedy lichá čísla a přechody čísla sudá. Tyto dva typy skládání jsou výhodné, pokud uživatele zajímá kompletní simulační běh, s přechody uvedenými před/za novým stavem.

```
1   {
2       "processes": [
3           {
4               "name": "<jmeno procesu>",
5               "location": {
6                   "name": "<aktualni stav>",
7                   "type": "[common|urgent|committed]"
8               },
9               "invariant": "<vyraz>",
10              "vars": [ // lokalni promenne
11                  {
12                      "name": "<jmeno promenne>",
13                      "value": "<hodnota promenne>"
14                  }
15                  // ...
16              ]
17          },
18          // ...
19      ],
20      "vars": [ // globalni promenne
21          {
22              "name": "<jmeno promenne>",
23              "value": "<hodnota promenne>"
24          }
25          // ...
26      ],
27      "clocks": [ // globalni hodiny
28          "<vyraz omezeni>",
29          // ...
30      ]
31  }
```

Obrázek 3.1: Objekt stavu simulace.

```
1   [
2       {
3           "process": "<jmeno procesu>",
4           "from": "<pocatecni stav>",
5           "to": "<cilovy stav>",
6           "guards": "<vyraz>",
7           "sync": "<vyraz>",
8           "updates": "<vyraz>"
9       },
10      // ...
11  ]
```

Obrázek 3.2: Objekt přechodu simulace.

Objekt popisující stav systému v rámci simulace (obrázek 3.1) obsahuje informace o všech procesech v systému, hodnoty proměnných a všechna omezení nad hodinami z XTR souboru s tím, že čísla uvedená v XTR nahrazuje informacemi získaných z XML a IF souboru. Struktura tohoto objektu je složena ze tří částí, které odpovídají předchozímu popisu (procesy, proměnné a omezení hodin). Část procesy je pole objektů, které obsahují informace

o konkrétních procesech v systému. Kromě jména procesu a místa, ve kterém se nachází, tento objekt obsahuje výraz invariantu a seznam lokálních proměnných spolu s jejich hodnotou, pro zlepšení čitelnosti simulačního běhu. U místa je také přidána informace o jeho typu – běžný, urgentní nebo `committed`. Část proměnné obsahuje pouze všechny globální proměnné a jejich hodnotu v tomto stavu simulace a část omezení hodin obsahuje výrazy omezení hodin. I přes to, že hodiny mohou být také lokální, reprezentace omezení hodin nedělí na lokální a globální z důvodu, že hodiny nemají konkrétní hodnotu, ale jen omezení, které je uvedeno jako výraz, ve kterém jsou dvoje hodiny, jak bylo uvedeno dříve. Výraz omezení tedy může obsahovat jedny globální hodiny a jedny lokální hodiny a nejčitelnější pro uživatele je nechat tato omezení na jednom místě.

Objekt popisující přechody (obrázek 3.2) také obsahuje informace získané z XTR souboru, obohacené o informace z IF souborů. Objekt je složen z pole objektů, kde každý objekt odpovídá jednomu přechodu, protože mezi dvěma simulačními stavy může systém provést více přechodů (lépe řečeno hran). Například u výše zmíněného příkladu „bridge“, sebrání pochodně dvěma Vikingy zároveň, je přesně tato situace, a reprezentace ji musí podporovat. Každý objekt pro přechod (hranu) obsahuje jméno procesu, který hranu provedl, počáteční a koncové místo dané hrany a informace o stráži, synchronizaci a aktualizaci, které byly na hraně nastaveny. Jedná-li se o násobnou hranu, objekt obsahuje i položku `select`, která obsahuje pole čísel reprezentující daný výběr.

3.2 Použité prostředky

Jedním z požadavků na aplikaci je její přenositelnost mezi systémy. Z toho důvodu byl zvolen jazyk C# s využitím .NET Core. Aplikace je napsána v .NET Core ve verzi 3.0. Pro práci s formátem JSON je využíván balíček `Newtonsoft.Json`. Jak zprovoznit překlad implementované aplikace je popsáno na konci této podkapitoly. Aplikace byla vyvinuta ve vývojovém prostředí Microsoft Visual Studio Community 2019, zejména proto, že toto vývojové prostředí obsahuje velice dobrý ladící nástroj.

Další prostředek, který byl velmi často využíván a je nutný k použití implementované aplikace, je dříve zmíněný program `verifyta`. Jak vytvořit soubor IF bylo popsáno v kapitole o tomto souboru. Pro automatické testování byl použit skriptovací jazyk Python. Poslední prostředek, který byl použit během implementační části, je program `tracer` z knihovny UTAP, který byl také již představen. Program byl použit ke kontrole správnosti překladu.

Veškeré ilustrace, mimo snímky obrazovky, byly vytvořeny ve webové aplikaci pro tvorbu diagramů https://app.diagrams.net/.

Překlad aplikace

K překladu aplikace je nutné mít nainstalovaný balíček .NET Core ve verzi alespoň 3.0. Tento balíček lze stáhnout na adrese https://dotnet.microsoft.com/download. Po výběru operačního systému, stránka přesměruje na konkrétní návod, jak balíček nainstalovat. Pro linuxové systémy je v horní části výběrové menu, ve kterém lze zvolit konkrétní distribuci a návod se upraví. Pro Windows je nutné stáhnou SDK (Software Development Kit), který obsahuje vše potřebné pro vývoj a sestavení aplikace.

Pro sestavení implementované aplikace je nutné otevřít v příkazové řádce složku, ve které se nachází soubor s koncovkou „.sln“ (`src/TraceConverter`). Pro překlad a sestavení slouží následující příkaz:

```
>dotnet build --configuration Release
```

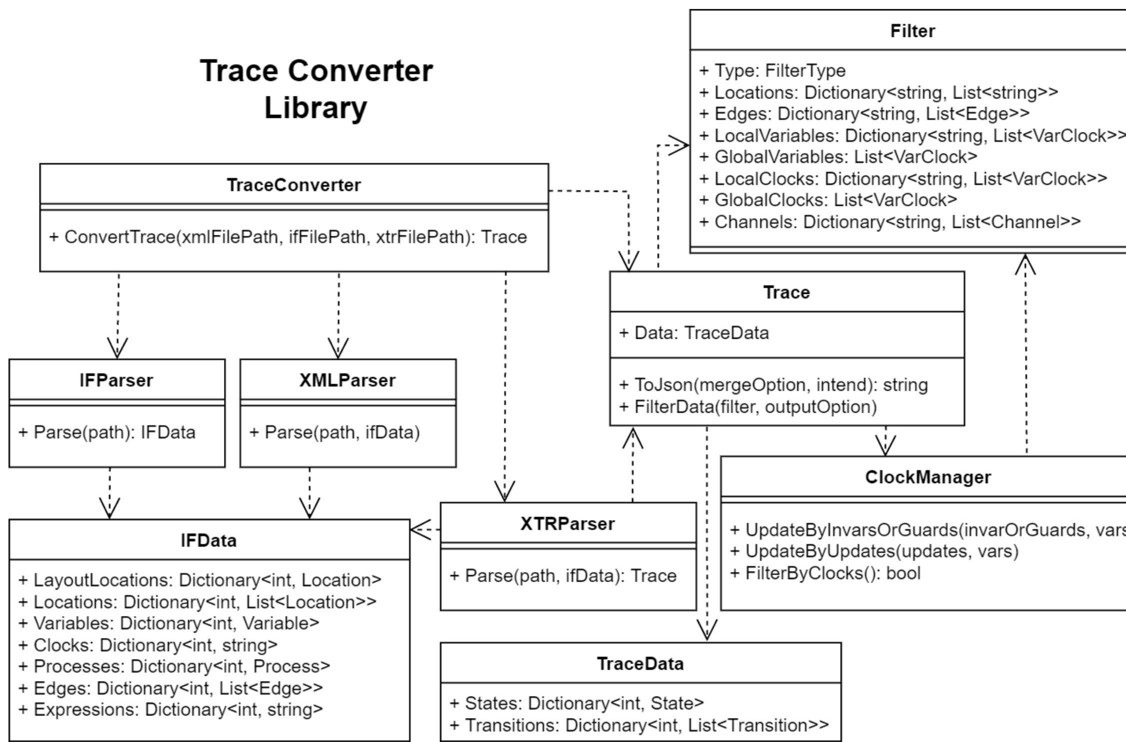
Přepínač `--configuration` není nutné specifikovat a překlad se i tak provede. Rozdíl je v tom, ve které složce bude výsledná přeložená aplikace, a jak bude optimalizována. Výchozí hodnotou konfigurace při překladu je „Debug". Výsledná aplikace se nachází v podsložce dané následující cestou od souboru s koncovkou „.sln": `TraceConverter/bin/Release/netcoreapp3.0/`, kde složka „Release" bude nahrazena složkou „Debug" při nepoužití přepínače `--configuration`. Pro používání aplikace je vhodné přepínač zachovat. Verze aplikace, přeložená bez přepínače, je optimalizována pro ladící účely.

3.3 Návrh aplikace

Aplikace je navržená jako konzolová aplikace. Výstupem je pouze strukturovaný text, a tak není nutné grafické uživatelské rozhraní. Dalším důvodem výběru typu konzolové aplikace byl fakt, že v zadání je požadavek, aby aplikace byla snadno přenositelná mezi systémy. Pro grafické rozhraní je tato přenositelnost zpravidla komplikovanější. Jak již bylo zmíněno, aplikace je napsána v jazyce C# s využitím .NET Core ve verzi 3.0. Struktura řešení je rozdělena na dvě části. První částí je knihovna, která se stará o překlad a filtraci, a druhou částí je samotná konzolová aplikace, která provádí zpracování argumentů a, po zavolání patřičných metod v knihovně, vypisuje výstup.

Na obrázku 3.3 je znázorněn diagram tříd pro knihovnu. Nejsou zde uvedeny třídy, které slouží pouze pro uložení zpracovaných elementárních částí ze souborů XML, XTR a IF z důvodu velikosti konečného diagramu. Je uveden pouze objekt `IFData`, který obsahuje všechny podstatné informace získané z IF souboru, a objekt `TraceData`, který obsahuje indexem očíslované zpracované stavy a přechody simulačního běhu. Vstupním bodem knihovny je třída `TraceConverter` a její metoda `ConvertTrace`, která přebírá cesty ke všem souborům jako parametry. Výstupem této metody je objekt typu `Trace`, který obsahuje přeložené informace simulačního běhu (`Data`) v objektu typu `TraceData`. Třída `Trace` dále obsahuje metody `ToJson` a `FilterData`. Metoda `ToJson` převede vnitřně uložená data do textového výstupu formátovaného jako JSON řetězec. Je možné zvolit, jakým způsobem mají být jednotlivé objekty poskládány (možnosti z popisu návrhu reprezentace), a zda má být výstup formátován pomocí odsazování. Metoda `FilterData` umožňuje zpracovaný simulační běh vyfiltrovat pomocí specifikovaného filtru (třída `Filter`). Během filtrace je používán manažer hodin (třída `ClockManager`), který se stará o intervaly, ve kterých se jednotlivé hodiny nacházejí. Více na toto téma je uvedeno v kapitole o filtraci.

Konzolová aplikace obsahuje pouze vstupní bod v podobě metody `Main` ve třídě `Program`, objekt pro uložení specifikovaných argumentů a třídu, která se stará o zpracování argumentů. Výpis nápovědy, který obsahuje všechny podporované argumenty je uveden v příloze E. Aplikaci lze spustit buď s cestami k souborům XML, XTR a IF, nebo s cestou k souboru JSON, který obsahuje dříve zpracovaný simulační běh touto aplikací. Podmínkou je, aby tento soubor obsahoval objekty poskládané výchozí možností (tedy nejdříve stavy a poté přechody). Na obrázku 3.4 je znázorněn velmi zjednodušený vývojový diagram. U obou typů spuštění lze specifikovat volitelné argumenty, je tedy možné filtrovat již částečně vyfiltrovaný simulační běh. Argument `-o file` přesměruje výstup do souboru namísto do konzole. Typ složení objektů stavů a přechodů lze nastavit pomocí argumentu `-m`. Přepínačem `-I` je možno zamezit formátování JSON řetězce pomocí odsazovaní. Argumenty `-s` resp. `-t` zapříčiní to, že výstup obsahuje pouze stavy, resp. pouze přechody. Ostatní argumenty se



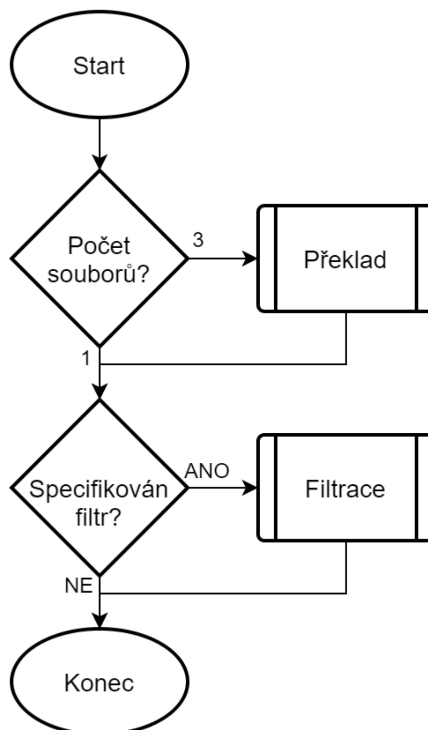
Obrázek 3.3: Diagram tříd knihovny pro překlad.

vztahují k filtraci a jsou popsány v podkapitole o filtraci. Nápověda obsahuje také popis syntaxe filtračních možností.

Aplikace předpokládá vstupní soubory ve správném formátu, které generuje nástroj Uppaal a program `verifyta`. Pokud je během zpracování souboru nalezen neznámý prvek, je vypsána příslušná chybová zpráva a aplikace ukončena. Pro soubor XML dále aplikace předpokládá, že jeho obsah prošel ověřením syntaxe (F7 v nástroji Uppaal) a neobsahuje chyby.

3.4 Popis překladu

Následující popis se vztahuje pro spuštění, při kterém jsou specifikovány cesty ke všem třem souborům. Při spuštění se souborem JSON překlad neprobíhá. Na obrázku 3.5 je uvedeno grafické vyjádření překladu simulačního běhu. Prvním souborem, který se zpracovává, je soubor IF. Informace získané z tohoto souboru jsou dále upravovány a používány. Jak bylo uvedeno u popisu tohoto formátu, soubor obsahuje celkem šest tabulek. Pro samotný překlad nejsou důležité všechny informace, které jsou v tabulkách uvedeny, například tabulka `instructions` je při zpracování celá přeskočena. Z tabulky `layout` jsou uloženy informace o hodinách, místech a proměnných. U hodin a proměnných je uloženo jejich jméno, které je pro lokální varianty prefixováno názvem procesu, a hodnota `nr` (pořadové číslo). U míst je kromě jména uloženo číslo `index`, k němuž se odkazují další tabulky, a informace o typu místa (běžné, urgentní, `committed`). Tabulku `processes` aplikace ukládá kompletní. Z tabulky `locations` jsou také ukládány všechny informace s tím, že jsou tyto informace vkládány do objektů vzniklých při zpracování míst v tabulce `layout`. Dále je získávána a ukládána informace o pořadí místa v rámci procesu. Tato informace je důležitá

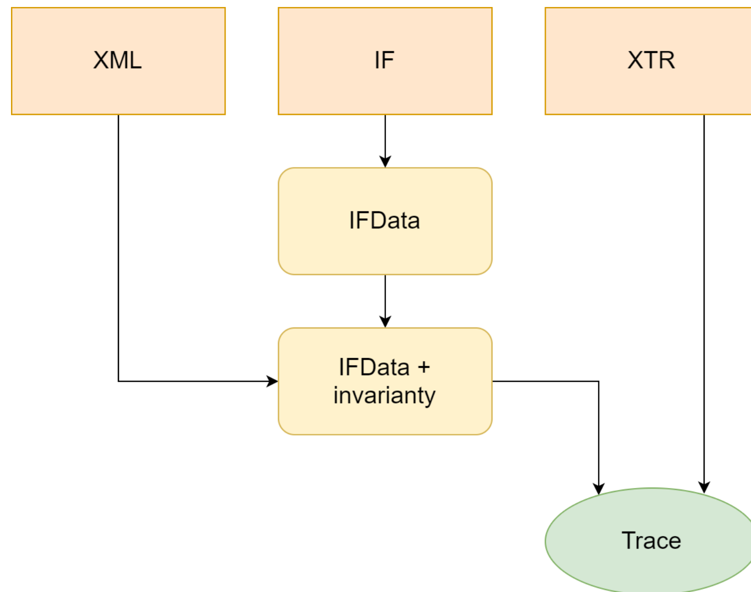


Obrázek 3.4: Zjednodušený vývojový diagram aplikace.

při úpravě získaných informací pomocí dat ze souboru XML. Na diagramu tříd (obr. 3.3) je vidět, že objekt `IFData` obsahuje kolekce pro místa z tabulek `layout` a `locations` odděleně, přičemž kolekce jsou různých datových typů. První kolekce obsahuje pouze seznam všech míst, kde klíčem je index, zatímco druhá kolekce skládá místa podle procesů. Klíčem v této kolekci je index procesu a hodnou seřazený seznam jemu příslušných míst. Při zpracování této tabulky je plněna i druhá kolekce odkazy na objekty, které jsou uloženy v první kolekci. Důvodem tohoto přístupu je ten, že při zpracování souboru XTR je v některých případech nutné hledat podle indexu a v jiných podle procesu a pořadového čísla místa. Z následující tabulky `edges` jsou uloženy všechny informace, ale jsou uloženy podle čísla procesu (podobně jako druhá kolekce pro místa). Z poslední tabulky `expressions` jsou uloženy pouze identifikace a texty výrazů.

Po zpracování a uložení všech podstatných informací pro překlad ze souboru IF, je zpracován soubor XML. Z tohoto souboru jsou pro účely aplikace potřebné pouze výrazy invariantů v místech, ve kterých jsou definovány. V tabulce `locations` v souboru IF je sice pole `invariant`, které obsahuje odkaz do tabulky `expressions`, ve které by měl být výraz uveden, program `verifyta` však obsahuje chybu a tyto výrazy nejsou kompletní. Testováním bylo zjištěno, že pro výraz např. $x < 5$ program `verifyta` v tabulce `expressions` uvede pouze x. Tato chyba je jediným důvodem, proč aplikace vyžaduje soubor XML. Pro správné přiřazení invariantů k místům, jsou ze souboru XML uložena všechna místa přidělena jednotlivých šablonám. Po zpracování všech šablon je zpracován element `system`, ve kterém se vytváří pojmenované instance těchto šablon. Podle jména procesů jsou následně upraveny všechny výrazy invariantů získané při zpracování IF souboru.

Jakmile jsou získány všechny informace ze souborů IF a XML, aplikace přechází k průchodu souboru XTR a plněný objektu typu `Trace`. Každý stav simulačního běhu by se dal



Obrázek 3.5: Ilustrace procesu překladu simulačního běhu implementovanou aplikací.

rozdělit na čtyři části, jak bylo uvedeno u popisu formátu XTR. S výjimkou počátečního stavu, ve kterém chybí část o provedených přechodech. Při překladu první části se vyhledávají informace o místech podle čísla procesu a pořadového čísla místa v rámci procesu. V sekci „zóna" překlad převádí jednotlivá omezení na výrazy formátu $x - y < d$, případně $x - y <= d$, kde x a y jsou hodiny a $d \in \mathbb{N}$. Pro proměnné překlad kromě získání jména zjišťuje, jestli se jedná o proměnnou lokální nebo globální, a podle toho ji následně přiřazuje správnému objektu v přeloženém simulačním běhu. Pro přechody v kroku simulace překlad vyhledá konkrétní hranu pro daný proces a vyplní všechny informace získané z IF souboru pro tuto hranu. V tomto případě se místa (počáteční a koncová) vyhledávají pouze podle indexu v tabulce `layout`.

3.5 Filtrace

Filtrace simulačního běhu se provádí vždy, když je specifikován filtr. Ten lze nastavit použitím argumentů `-b`, `-bf`, `-w` nebo `-wf`. Argumenty s písmenem `b` jsou tzv. „blacklist" možnosti (tedy co se má vyfiltrovat) a argumenty s písmenem `w` jsou tzv. „whitelist" (tedy co se má zachovat), přičemž kombinace „blacklist" a „whitelist" není dovolena. Varianty argumentů s písmenem `f` berou filtrační možnosti ze souboru namísto z příkazové řádky. Při použití filtrace specifikované na příkazové řádce, jsou jednotlivé filtrační možnosti odděleny znakem `;` (středník). Pokud je použit soubor, jsou v něm filtrační možnosti uvedeny na jednotlivých řádcích, s tím, že na jednom řádku je právě jedna filtrační možnost. Jak již bylo zmíněno, syntaxe filtračních možností je uvedena v nápovědě (výpis v příloze E). Následující popis tuto syntaxi prochází a dále vysvětluje.

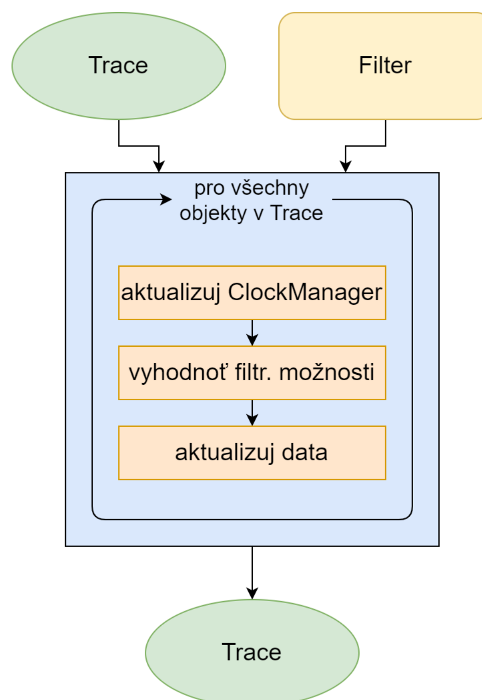
Každá filtrační možnost začíná klíčovým slovem a znakem `:` (dvojtečka). Za touto částí následují informace pro filtrování oddělené znakem `.` (tečka). Pro filtraci místa slouží klíčové slovo `location`. Informace za tímto slovem obsahují buď pouze jméno místa, nebo jméno procesu a jméno místa. Pro možnost, kde je specifikováno pouze jméno místa, je filtrace použita na všechny procesy, které obsahují místo s tímto názvem. K filtraci pře-

chodů podle hran slouží klíčové slovo `edge`. Tato varianta dále obsahuje jméno procesu a jména počátečního a koncového místa. Podobně jako u míst, je i v této variantě možnost vynechat jméno procesu. V takovém případě budou filtrovány hrany ve všech procesech. Pro filtraci podle hodnoty proměnné je možno využít klíčová slova `localVar` (pro lokální proměnné) a `globalVar` (pro globální proměnné). Informace u těchto filtračních možností specifikují jméno proměnné a interval, ve kterém se proměnná musí nacházet, aby byla vyfiltrována. U lokálních proměnných lze opět uvést jméno procesu se stejným významem jako v předchozích možnostech. Další možností filtrace je podle hodnoty hodin. V tomto případě jsou použita klíčová slova `localClock` a `globalClock`, za kterými následují stejné informace jako u proměnných, včetně možnosti specifikování jména procesu pro lokální varianty. Poslední filtrační možností je filtrace podle synchronizačních kanálů. Klíčové slovo je `chan`, za kterým následuje jméno kanálu. Opět lze nastavit i jméno procesu se stejným významem jako dříve. Navíc je možno na konec jména kanálu přidat znak `!` nebo `?`, čímž se omezí filtrace na vysílání, resp. přijímání po kanále. Filtrační možnost pro systém příkladu „bridge“, která vyfiltruje všechny stavy, ve kterých se proces s názvem Viking1 nachází ve stavu `safe`, vypadá následovně:

location:Viking1.safe

Filtrace je prováděna tak, že se prochází přeložený simulační běh, a v každém stavu či přechodech se vyhodnocují filtrační možnosti. V případě, že systém splňuje specifikovanou filtrační možnost, například hodnota proměnné je v uvedeném intervalu, celý stav či přechody jsou ze simulačního běhu odstraněny (pro typ „blacklist“). Pokud je použit filtr typu „whitelist“, je daný stav či přechody zachován je v případě, kdy jsou všechny jeho části uvedeny ve filtračních možnostech. Jedná se tedy o podstatně více omezující typ filtru. Filtrace se provádí až po překladu simulačního běhu a ne během samotného překladu, protože aplikace podporuje spuštění se souborem JSON, při kterém k opětovnému překladu nedochází. Na obrázku 3.6 je, podobně jako v části o překladu, uvedeno grafické znázornění filtrace simulačního běhu.

Protože jsou během simulace hodiny uchovávány jen v podobě výrazů omezení, bylo nutné implementovat samostatného manažera (`ClockManager`), který se stará o všechny hodiny systému. Manažer si uchovává intervaly, ve kterých se nachází hodnota daných hodin. Tyto intervaly jsou modifikovány při průchodu přeloženým simulačním během výrazy invariantů, stráží a aktualizací. Protože k zjištění, že výrazy invariantů program `verifyta` nevytváří správně, došlo až v pozdější části implementace, a protože zpracování deklaračních částí v souboru XML je velice náročné, filtrace podle hodin je velice omezená. Výchozí interval všech hodin na začátku překladu je nastaven na $\langle 0; \infty)$. Ve výrazech invariantů, stráží a aktualizací jsou dovoleny pouze proměnné, argumenty, číselné hodnoty a další hodiny. Výrazy, které obsahují například volání funkcí či konstanty, nejsou podporovány. Při aktualizacích dále nejsou podporovány ostatní výraz než přiřazení, například $time+ = 5$ nebo $time = timeA - timeB$. Aplikace však podporuje složené výrazy ($timeA < 5$ && $timeB > 10$). Dále je manažer implementován tak, aby byl co možná nejméně omezující. Vyskytuje-li se ve výrazu například porovnání dvou hodin operátorem `==`, je oběma hodinám přiřazen největší možný interval, ve kterém se hodnota může vyskytovat. Pokud je zjištěno, že hodnota hodin může být v intervalu specifikovaném filtrační možností, je daný stav či přechody vyfiltrován.



Obrázek 3.6: Ilustrace procesu filtrace simulačního běhu implementovanou aplikací.

3.6 Problémy a jejich řešení

První implementace vyžadovala pouze soubory IF a XTR, podobně jako program `tracer` z knihovny UTAP. Jak již ale bylo zmíněno, program `verifyta` nesprávně tvoří výraz pro invarianty. Nejjednodušším řešením tohoto problému by bylo nepoužívat tyto invarianty, ale vzhledem k tomu, že filtrace podle hodnoty hodin, je důležitou součástí, toto řešení nebylo přijatelné. Výrazy invariantů jsou správně uloženy v souboru XML, a tak návrh druhé verze aplikace spočíval v nevyžadování souboru IF, a tím i odebrání závislosti na programu `verifyta`, a místo něj vyžadovat soubor XML. Obsah souboru IF je přeci jen tvořen ze souboru XML, a tak lze určitě vytvořit jeho obsah vlastním zpracováním XML a vybráním podstatných částí. Implementovat tento návrh se ukázalo velice náročné. Zejména proto, že obsah části deklarací je v syntaxi podobné jazyku C, a není v XML souboru nijak strukturován. Pro správné získání všech objektů by bylo nutné napsat kompletní syntaktický analyzátor, což z časového důvodu nebylo možné. Finální verze, ta která byla představena v této kapitole, tedy požaduje všechny tři soubory a ze souboru XML jsou vybrány pouze správné výrazy invariantů.

Další problém, který byl nalezen, souvisí s testováním, které je popsáno v následující kapitole. Bylo zjištěno, že program `trace` nepodporuje názvy míst, které mají více jak 31 znaků. Jeden ze složitějších systémů, který byl testován, však obsahoval název místa o pár znaků delší než 31 znaků, a tak porovnání výstupu při ověřování správnosti překladu vrátilo negativní výsledek. Knihovna UTAP je distribuována s kódy ke všem programům a je nutné program před použitím přeložit. Řešením tohoto problému tedy bylo prosté navýšení počtu znaků, který se při získávání jména místa načítají, a znovu přeložení programu.

U popisu filtrace bylo zmíněno, že filtrace podle hodnoty hodin je značně omezené. Následující část této podkapitoly se blíže zabývá problémem získání konkrétní hodnoty

daných hodin. Dle teorie časovaných automatů má množina omezení hodin $B(C)$ prvky tvaru $x \bowtie c$ nebo $x - y \bowtie c$, kde $x, y \in C$ (hodiny), $c \in \mathbb{N}$ a $\bowtie \in \{<, \leq, =, \geq, >\}$, jak bylo uvedeno v úvodu. Simulační běhy nástroje Uppaal obsahují omezení hodin pouze ve formě $x - y < d$ nebo $x - y \leq d$, kde x a y jsou hodiny a $d \in \mathbb{N}$, jak bylo uvedeno v části o popisu XTR souboru. Toto je možné jen díky tomu, že všechny simulační běhy obsahují implicitní hodiny $t(0)$, které nelze modifikovat, a které pořád běží. Před napsáním manažera starajícího se o intervaly jednotlivých hodiny bylo zjišťováno, jestli není konkrétní hodnota hodin někde v simulačním běhu uložena. Za tímto účelem byl vložen dotaz na oficiální fórum nástroje Uppaal. Tento dotaz lze nalézt na adrese https://groups.google.com/d/msg/uppaal/1Z4TmP0IAj8/N5i6DuDEBQAJ. Odpověď je vyčerpávající a v podstatě potvrzuje předpoklad, že konkrétní hodnota hodin získat nelze. Hodiny jsou v simulačním běhu uloženy pouze jako výrazy omezení. Například, pro výše zmíněný příklad „bridge“, jedno z omezení má tvar $t(0) - time \leq 0$. Z tohoto omezení však nelze říct nic konkrétního o hodnotě hodin *time*. Jedinou možností, jak implementovat filtraci podle hodin, je tedy zabývat se intervaly, ve kterých se hodnota daných hodin může nacházet. Z teorie časovaných automatů vyplývá, že interval každých hodin je v počáteční konfiguraci $\langle 0; \infty)$, a že hranice intervalu lze měnit použitím invariantů a resetováním hodin. Nástroj Uppaal však používá rozšíření nad časovanými automaty, díky kterým je získání správného intervalu podstatně složitější. Například lze nastavit výchozí hodnotu hodin použitím konstant nebo lze během simulace hodnotu hodin měnit různými výrazy, včetně přiřazení výsledku definované funkce. Interval lze měnit pomocí invariantů v místech, případně strážemi a aktualizacemi na hranách. Ideální řešení by na začátku (v počáteční konfiguraci) získalo konkrétní hodnotu hodin a s pomocí invariantů definovalo počáteční interval. Následně by v každém stavu a přechodu simulačního běhu probíhalo vyhodnocování výrazů invariantů, stráží a aktualizací a intervaly všech hodin by se správně aktualizovaly. Implementace tohoto ideálního řešení je však velice náročná. Musela by mimo jiné obsahovat syntaktický analyzátor deklaračních částí, pro získání hodnot i zřetězených konstant, a interpret uživatelem specifikovaných funkcí pro získání hodnot výrazů, ve kterých se objevuje volání funkce. Z tohoto důvodu je filtrace podle hodnoty hodin v implementované aplikaci velice omezená. Manažer hodin podporuje jen některé typy výrazů, jak bylo uvedeno v části o filtraci. Počáteční interval je nastaven na implicitní z teorie časovaných automatů, a je modifikován podporovanými typy výrazů v invariantech, strážích a aktualizacích. Samotné výrazy uvedené v sekci „zóna“ nejsou při aktualizaci intervalů používány. Pokud jsou některé hodiny v rámci stavu či přechodu simulace použity ve více výrazech (např. globální hodiny jsou použity v invariantech dvou stavů, ve kterých procesy mohou být současně), je aktualizace těchto hodin odložena na co nejpozději, přičemž je opět vytvořen co možná největší interval z daných omezení.

Kapitola 4

Zhodnocení navržené reprezentace a překladu

Tato kapitola se blíže zabývá hodnocení navržené uživatelsky přívětivé reprezentace a navržené aplikace pro překlad simulačních běhů nástroje Uppaal do navržené reprezentace. Jak již bylo zmíněno v úvodu, hodnocení uživatelské přívětivosti je často velice subjektivní. Z tohoto důvodu byl pro zhodnocení navržené reprezentace vytvořen dotazník. Stručný popis s výsledky je uveden v následující podkapitole.

Další podkapitoly se zabývají zhodnocením implementované aplikace. V podkapitole o testování je popsáno, jak automatické testování s porovnáním výstupu s programem `tracer`, tak ruční testování, které bylo třeba provést u testování filtračních možností. Poslední podkapitola je věnována odhadu složitosti aplikace. Zkoumána byla doba běhu aplikace a celkový počet alokované paměti.

4.1 Dotazník

Vytvořený dotazník na začátku obsahoval krátký úvod k nástroji Uppaal, za kterým následoval podrobnější popis příkladu „bridge“, který byl uveden dříve. Poté byl vysvětlen výstup simulačních běhů v podobě XTR souboru a přidán byl i krátký popis souboru IF. Následoval popis navržené uživatelsky přívětivé reprezentace, na kterou byl dotazník navržen. Za tímto úvodem do problematiky byl uveden první soubor čtyř otázek. Tyto otázky jsou spolu s výsledky uvedeny v následující části.

1. Přijde vám zvolení formátu JSON vhodné?
   Z možností:

   - naprosto nevhodné
   - nevhodné
   - nevím
   - vhodné
   - JSON je nejlepší volba

   66.7% zvolilo možnost „JSON je nejlepší volba“ a zbylých 33.3% zvolilo možnost „vhodné“.

2. Kdyby jste vy volili formát uživatelsky přívětivé reprezentace, jaký by jste zvolili?
   Odpověď na tuto otázku byla textová. Všichni účastnici se ale shodli na tom, že formát JSON je velmi vhodný.

3. Jak hodnotíte rozhodnutí oddělit objekty pro stavy a přechody?
   Z možností:

   - velmi špatný nápad
   - špatný nápad
   - nevím
   - dobrý nápad
   - výborný nápad

   100% zvolilo možnost „dobrý nápad“.

4. Jak moc vám přijde navržená reprezentace uživatelsky přívětivá?
   Odpověď na tuto otázku bylo číslo v rozsahu 0 až 5, kde 0 znamená žádná přívětivost a 5 naprosto přívětivé. 66,7% hodnocení obdrželo číslo 4 a zbylých 33,3% připadlo číslu 5.

Po tomto souboru otázek byl představen program `tracer` a uveden výstup z tohoto programu. Za tímto popisem byla uvedena poslední otázka, jejíž znění a výsledek je uveden v následující části.

5. Přijde vám navržená reprezentace lepší než výstup programu `tracer`?
   Z možností:

   - výstup programu `tracer` je lepší
   - tak nastejno
   - navržená reprezentace je lepší

   100% zvolilo možnost „navržená reprezentace je lepší“.

4.2 Testování

Testování bylo prováděno ve dvou částech, jak bylo nastíněno v úvodu kapitoly. V první části se jednalo o automatizované testování, které porovnávalo výstup implementované aplikace s výstupem programu `tracer`. Protože však ve výstupu programu `tracer` nejsou všechny informace, které obsahuje navržená reprezentace, bylo třeba přeložený běh upravit a změnit informace, které výstup programu `tracer` neobsahuje, na výchozí hodnoty. Podobná úprava byla provedena u výstupu programu `tracer`, přičemž zde dané informace byly přidány a vyplněny výchozími hodnotami. Další úprava výstupu programu `tracer` spočívala v přeuspořádání jednotlivých částí stavů a přechodů do formátu JSON pro lepší porovnání. Tyto úpravy byly provedeny v pomocné aplikaci, která byla také napsána v jazyce C#.

Automatické testování bylo řízeno skriptem, napsaném v jazyce Python. Tento skript má na začátku v globálních proměnných uvedené cesty ke všem programům, které potřebuje. Konkrétně se jedná o cesty k programu `verifyta`, programu `tracer`, implementované aplikaci `TraceConverter` a pomocné aplikaci pro ověření správnosti výstupu. Skript je umístěn

do složky, ve které se nachází testované systémy spolu s uloženými simulačními běhy. Před samotným ověřováním, skript prochází obsah složky a získává jména všech systémů, přičemž jména XML a XTR souborů musí být shodná, aby byl simulační běh přiřazen danému systému. Pokud je ve složce systém, pro který je uveden buď pouze XML soubor, nebo pouze XTR soubor, je vypsána upozorňující hláška. Po získání jmen všech systémů jsou odstraněny všechny složky, které nesou daná jména (pro případ opětovného testování). Následně se pro každé jméno vytvoří nová složka, do které se ukládají jednotlivé soubory vzniklé při spouštění programů. Nejdříve se vytvoří soubor IF pomocí programu `verifyta`, s názvem jména procesu, a následně požadovaný výstup pomocí programu `tracer`, který je uložen do souboru „tracer.out“. Poté se spustí překlad pomocí implementované aplikace s použitím parametru `-o`, kde výstupní soubor má opět jméno systému a příponu „.json“. Požadovaný výstup v souboru „tracer.out“ a získaný výstup z překladu implementovanou aplikací je předán pomocné aplikaci. Ta po provedení potřebného přeskládání a nastavení výchozích hodnot vypisuje, že se simulační běhy shodují, pokud byl překlad proveden správně. Pokud však při překladu došlo k chybě, aplikace vypisuje informaci o této skutečnosti a navíc vypisuje JSON reprezentaci simulačního běhu přeloženého program `tracer`. Skript kontroluje výstup pomocné aplikace a v případě, že se nejedná o pozitivní výsledek, uloží výstup do souboru „tracerJSON.json“. Ten je možno, například ve webové službě, porovnat s výstupem implementované aplikace a zjistit, kde přesně došlo k chybě.

Mezi testovanými systémy byly jak jednoduché systémy, které obsahují pouze jeden proces a pár míst, tak rozsáhlé systémy s mnoha procesy a dohromady stovkami míst. Dohromady bylo automatické testování prováděno nad sadou 18 systémů, přičemž dané systémy byly tvořeny a získávány za účelem pokrýt všechny možnosti, které mohou při simulaci nastat. Testován byl i velmi složitý systém s názvem „adpcm“, který je uveden v části o složitosti, při jejímž zkoumání byl také použit. Délka simulačních běhů byla volena tak, aby pro různě složité systémy zahrnovala jak krátké, tak dlouhé běhy. Kompletní sadu testů lze nalézt ve složce „Tests“ na přiloženém datovém médiu. V této složce se nachází i testovací skript, ve kterém ale nejsou vyplněny cesty k jednotlivým programům. Pomocná aplikace není součástí výsledné aplikace, je však také uložena na datovém médiu ve složce `src` pod názvem `TraceChecker`. Pro překlad pomocné aplikace ji stačí přidat do „solution“ projektu a nastavit závislost na `TraceConverterLibrary`.

Automatické testování odhalilo mimo chyb aplikace i některé chyby, které v režii aplikace nejsou. Například při ověřování zmíněného systému „adpcm“, bylo odhaleno omezení programu `tracer` související s maximální délkou jména místa, které bylo uvedeno dříve.

Druhou částí bylo ruční testování, které se zabývalo především ověřováním správnosti filtrace. Při tomto testování docházelo k systematickému spouštění implementované aplikace a zkoumání výstupu, zda obsahuje očekávanou množinu stavů a přechodů simulačního běhu. Další ruční testy se vztahovaly na ty části překladu, které nebylo možné zkoumat automatickým testováním, protože program `tracer` tyto informace neuvádí. U těchto částí, například výraz invariantu, byl namodelovaný systém a simulační běh měněn, a po každé změně byl zkoumán výstup aplikace, zda je správný. Celkový počet těchto testů nebyl monitorován, byly však ověřeny všechny možnosti filtrace (jak jednotlivě, tak v kombinaci s dalšími filtračními možnostmi) a všechny části, které nejsou uvedeny ve výstupu programu `tracer` a nebylo je tedy možno otestovat automaticky.

Při ručním testování byla odhalena chyba programu `verifyta` související s výrazem invariantu, která byla také popsána dříve.

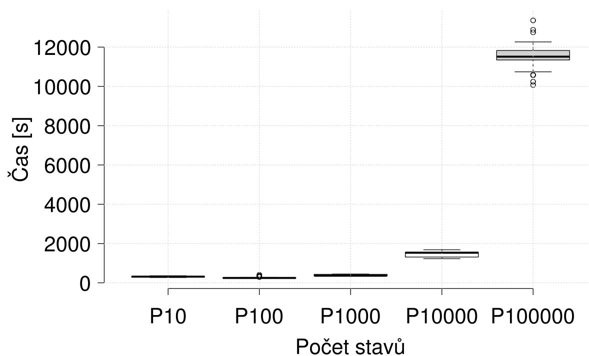
Finální verze aplikace vytváří správné výstupy pro všechny automatické testy. U ručního testování je nemožné s jistotou říct, že je aplikace bezchybná. Pro provedené testy však výstup vždy odpovídal předpokladu.

4.3 Odhad složitosti

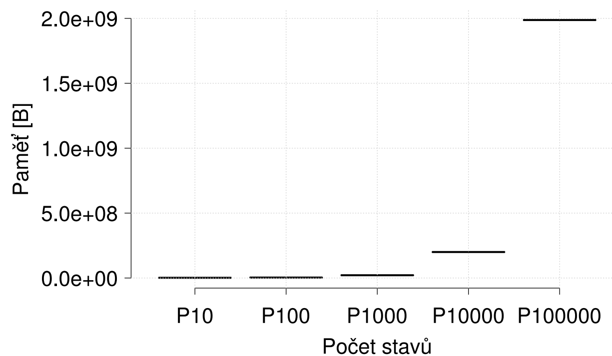
Pro odhad složitosti byl vytvořen skript, který provedl 30 oddělených běhů aplikace pro danou trojici souborů XML, IF a XTR. Složitost aplikace byla zkoumána na středně rozsáhlém příkladu „train-gata-stat“, který lze nalézt v demonstračních příkladech distribuce nové verze nástroje UPPAAL, a na složitém příkladu „adpcm“ ze souboru UPPAAL modelů pro METAMOC (dostupné na http://metamoc.dk/).

Testování bylo prováděno na počítači s procesorem `Intel Core i7-8550U` a s 16GB operační paměti. Odhad složitosti byl prováděn vůči délce simulačního běhu. Jednotlivé varianty XTR souborů měly 10, 100, 1000, 10000 a 100000 stavů v simulačním běhu. Výsledky pro 100000 stavů u příkladu „adpcm“ nebylo možno získat z důvodu nedostatku paměti.

Následující obrázky obsahují krabicové grafy, na kterých jsou znázorněny doby běhů a velikosti alokované paměti. Obrázky 4.1 a 4.2 se vztahují k příkladu „traing-gate-stat“ a obrázky 4.3 a 4.4 k příkladu „adpcm“. Značení na ose x znamená počet stavů (např. P100 = 100 stavů v simulačním běhu). Uvedené hodnoty na ose y jsou pro čas v sekundách a pro paměť v bytech.



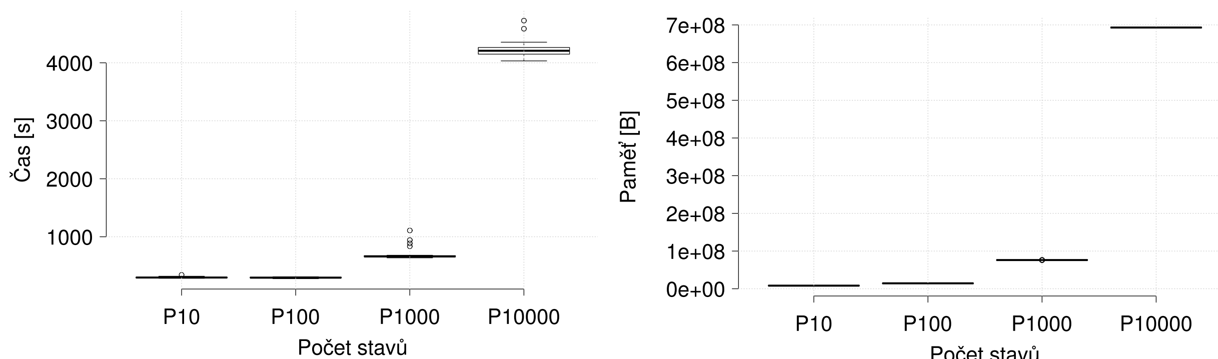
Obrázek 4.1: Krabicový graf pro doby běhu aplikace nad příkladem „train-gate-stat“.



Obrázek 4.2: Krabicový graf pro alokovanou paměť aplikace nad příkladem „train-gate-stat“.

Z naměřených dat a uvedených krabicových grafů bylo zjištěno, že časová i prostorová složitost roste lineárně s počtem stavů v simulaci. Grafy sice lineární křivku nepřipomínají, to je ale dáno tím, že osa x roste s mocninou desíti.

Při implementaci aplikace bylo na složitost myšleno a byly prováděny úpravy, za účelem snížit především paměťovou složitost. Jedna z těchto úprav byla zmíněna dříve při popisu překladu. Uložení míst pomocí dvou různých objektů, které se liší tím, jaké klíče je potřeba použít k získání konkrétní informace o místě, výrazně snížilo paměťovou složitost, protože nebylo třeba kolekce složitě prohledávat. Tato úprava umožnila otestovat „traing-gate-stat“ se 100000 stavovým simulačním během.



Obrázek 4.3: Krabicový graf pro doby běhu aplikace nad příkladem „adpcm“.

Obrázek 4.4: Krabicový graf pro alokovanou paměť aplikace nad příkladem „adpcm“.

4.4 Návrhy pro pokračování v projektu

Tato podkapitola představuje některé návrhy pro pokračování v tomto projektu. Jsou zmíněny některé úpravy implementované aplikace, které by šlo provézt, i nové směry, kterými by se dalo vydat, a které souvisí s tímto projektem.

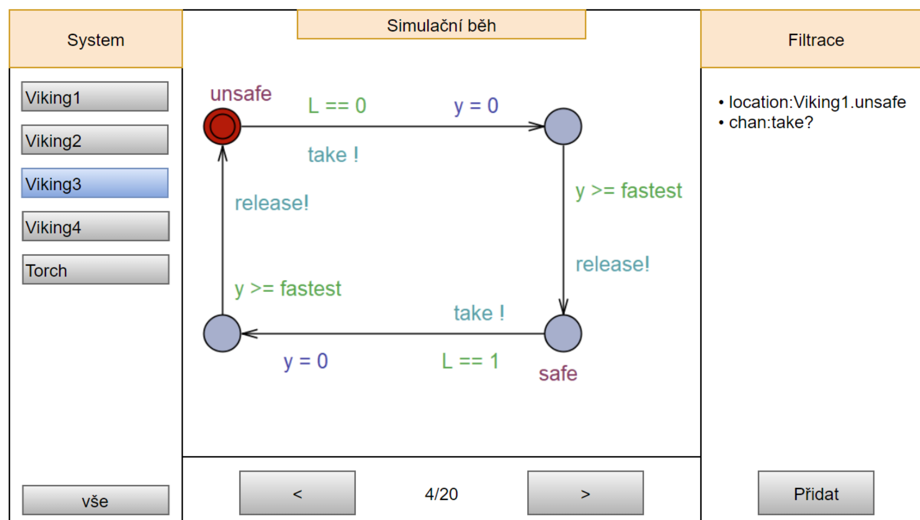
První nápad pro pokračování v projektu souvisí s dříve zmíněným problémem. Implementovaná aplikace vyžaduje na vstupu tři soubory a je závislá na programu `verifyta`. Obsah souboru IF však lze získat zpracováním souboru XML. Je však nutné implementovat syntaktický analyzátor pro deklarační části. Tento směr vývoje by mohl být jedním z pokračování v tomto projektu.

Dalším pokračováním by mohla být práce, která se zabývá složitostí. Implementovaná aplikace je paměťově velmi náročná pro dlouhé simulační běhy. V případě, že je specifikován nějaký filtr, by šla paměťová složitost, pro verzi spuštění, při které dochází k překladu, snížit například tím, že by se filtrace prováděla zároveň s překladem. Další možností, jak zlepšit paměťovou složitost, by mohlo být nějaké chytré řešení ukládání objektů ze souboru IF, které jsou často používány při překladu simulačního běhu.

Filtrace simulačních běhů by mohla být další oblastí pro pokračování v projektu. Dalo by se navrhnout složitější filtrování, možná i s možností kombinace „blacklist“ a „whitelist“ filtrů. Dále filtrace podle hodnoty hodin je v implementované aplikaci velmi omezená, jak bylo uvedeno dříve. Tato možnost by se dala dále rozvést. Například úpravou některých částí manažera hodin by bylo možné podporovat vetší rozsah typů výrazů.

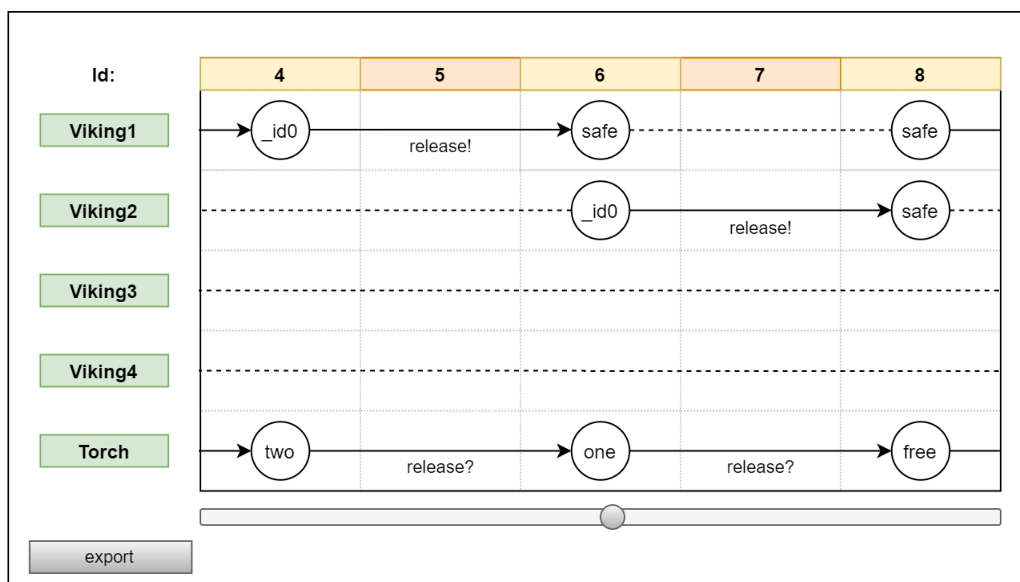
Dalším, poměrně jednoduchým, rozšířením aplikace, by mohlo být přidání výběru výstupního formátu. Například XML či YAML by mohly být další možnosti k implementované variantě JSON.

Poslední návrh, který je v této podkapitole uveden, na rozdíl od ostatních, nesouvisí s úpravou implementované aplikace. Pokračováním v projektu by mohla být nová aplikace, která využívá vytvořenou implementaci v podobě knihovny. Tato aplikace by již nebyla konzolová, ale obsahovala by grafické uživatelské rozhraní, které by zahrnovalo i vizuální zobrazení simulačních běhů. Filtrace by se v této aplikaci dala provádět i za běhu (při krokování simulace). Znázornění návrhu grafického uživatelského rozhraní takové aplikace



Obrázek 4.5: Navržené GUI pro potenciální pokračování v projektu.

je uvedeno na obrázku 4.5. V levé části aplikace by šlo specifikovat zobrazení jednotlivých časovaných automatů procesů v systému, případně volbou „vše“ zobrazení všech. Největší prostor okna by zabírala prostřední část, ve které by bylo grafické znázornění simulačního běhu, pod kterou by bylo ovládání simulace. V pravé části by pak šlo specifikovat filtrační možnosti a simulační běh by se, spolu s popiskem celkového počtu stavů, za běhu upravil. Další funkcionalitou této aplikace by mohl být statický průchod simulačním během v podobě diagramu. Návrh pro tuto obrazovku aplikace je uveden na obrázku 4.6.



Obrázek 4.6: Návrh GUI pro statický průchod simulačním během v podobě diagramu.

Diagramem by se dalo pomocí dolního posuvníku posunovat v čase, přičemž nahoře uvedená identifikační čísla souhlasí s těmi použitými v JSON reprezentaci. Pro větší počet procesů v simulovaném systému, by mohl přibýt i vertikální posuvník, který by zlepšil přehlednost. Na obrázku je použita čárkovaná čára u všech stavů a přechodů systému,

ve kterých se pro daný proces nic nezměnilo. Toto by mohlo být součástí nastavení, kde druhou možností by bylo zopakovat předchozí stav a přechod by buď nebyl vyplněn, nebo by byla opět použita čárkovaná čára. Diagramem reprezentovaný simulační běh by dále šlo exportovat do obrázku, nebo do grafu popsaného syntaxí některého jazyka pro definování diagramu (například Graphviz).

Kapitola 5

Závěr

Tato diplomová práce se zabývala návrhem uživatelsky přívětivé reprezentace simulačních běhů nástroje Uppaal. V úvodu byly uvedeny souvislosti s tímto problémem, mezi které patří zejména časované automaty, které jsou k modelování a simulaci v nástroji použity, temporální logiky, které nástroj používá v ověřování specifikovaných vlastností a ověřování modelů („model checking“), který byl zasazen do vývojového cyklu produktu.

Další část práce podrobně popisuje nástroj Uppaal a je zde uvedena i dostupná programová podpora pro práci se soubory, které nástroj vytvoří. Je popsán návrh systému, který se provádí v záložce „Editor“, simulace a verifikace (záložky „Simulator“ a „Verifier“). U návrhu systému je také popsán a vysvětlen demonstrační příklad s názvem „bridge“, který je součástí distribuce nástroje Uppaal, a na který je v rámci textu vícekrát odkazováno. V této části jsou dále zmíněny i nástroje, které s Uppaal souvisí, mezi které patří například: Stratego, Cora nebo Tron. Za popisem samotné aplikace Uppaal jsou uvedeny jednotlivé soubory, se kterými nástroj pracuje. Konkrétně soubor XML, ve kterém je uložený namodelovaný systém, soubor XTR, ve kterém nástroj ukládá simulační běhy, a soubor IF, který slouží jako pomocný soubor pro překlad, ve kterém jsou uvedeny tabulky objektů modelu, na které se v souboru XTR odkazuje.

Kapitola o návrhu a implementaci podrobně popisuje navrženou uživatelsky přívětivou reprezentaci simulačních běhů nástroje Uppaal, která je formátována jako JSON řetězec. Jsou zde uvedeny všechny použité prostředky spolu s návodem, jak nainstalovat překladač pro jazyk C#, který byl použit při implementaci konzolové aplikace. Za popisem samotného návrhu aplikace, je uveden podrobný popis překladu simulačních běhů do navržené reprezentace a popis filtrace, spolu s popisem všech filtračních možností. Na konci kapitoly o návrhu jsou uvedeny i některé problémy, na které se při tvorbě práce narazilo, a je nastíněno jejich řešení.

Zhodnocení navržené reprezentace bylo prováděno formou dotazníku, který je spolu s výsledky uveden v další kapitole. V této kapitole je také popsáno testování a je odhadnuta složitost aplikace. Zkoumána byla délka běhu aplikace a celkový počet alokované paměti při překladu.

V poslední části textu jsou uvedeny některé návrhy, jak by se dalo v projektu pokračovat. Mezi ně patří například vylepšení složitostí, či nová aplikace s grafickým uživatelským rozhraním, která využívá implementaci v podobě knihovny.

Literatura

[1] Baier, C. a Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X.

[2] David, A., Larsen, K. G., Legay, A., Mikučionis, M. a Poulsen, D. B. *Uppaal SMC Tutorial* [online]. Denmark and France: Department of Computer Science, Aalborg University, Denmark and INRIA/IRISA Rennes, France, leden 2018 [cit. 2020-05-07]. Dostupné z: https://www.it.uu.se/research/group/darts/papers/texts/uppaal-smc-tutorial.pdf.

[3] Emerson, E. A. *TEMPORAL AND MODAL LOGIC* [online]. USA: Computer Sciences Department University of Texas at Austin, březen 1995 [cit. 2020-05-07]. Dostupné z: https://profs.info.uaic.ro/~masalagiu/pub/handbook3.pdf.

[4] ESI (Embedded System Institute). *Tool uppaal2octopus* [online]. 2013 [cit. 2020-05-07]. Dostupné z: https://github.com/Wassasin/uppaal2octopus.

[5] Gerd Behrmann, A. D. a Larsen, K. G. *A Tutorial on Uppaal 4.0* [online]. Denmark: Department of Computer Science, Aalborg University, listopad 2006 [cit. 2020-05-07]. Dostupné z: https://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[6] Hessel, A. *Uppaal Cover* [online]. 2017 [cit. 2020-05-07]. Dostupné z: http://www.hessel.nu/CoVer/.

[7] Håkansson, J. a Pettersson., P. *Uppaal Port* [online]. 2019 [cit. 2020-05-07]. Dostupné z: https://www.it.uu.se/research/group/darts/uppaal/port/.

[8] Iqbal, J., Truscan, D., Vain, J. a Porres, I. *Tron2Uppaal Back Tracer Tool* [online]. Červen 2015 [cit. 2020-05-07]. Dostupné z: https://www.researchgate.net/publication/282867122_TRON2UPPAAL_Backtracer_Tool_-_From_TRON_Logs_to_UPPAAL_Traces.

[9] Křepelková, S. *Metody uživatelského hodnocení HMI v prostředí automotive. Případová studie.* Brno, CZ, 2019. Diplomová práce. Univerzita Karlova v Praze, Filozofická fakulta. Dostupné z: https://is.cuni.cz/webapps/zzp/detail/202559/.

[10] Legay, A., Delahaye, B. a Bensalem, S. *Statistical Model Checking: An Overview.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010 [cit. 2020-05-07].

[11] Norman, D. a Nielsen, J. *The Definition of User Experience (UX)* [online]. Denmark: [b.n.], listopad 2006 [cit. 2020-05-07]. Dostupné z: https://www.nngroup.com/articles/definition-user-experience/.

[12] Schmaltz, J. *Timed Computational Tree Logic (TCTL)* [online]. Nizozemsko: Eindhoven University of Technology, březen 2017 [cit. 2020-05-07]. Dostupné z: https://www.win.tue.nl/~jschmalt/teaching/2IX20/reader_software_specification_ch_11.pdf.

[13] Stigge, M. *Uppaal 4.0 : Small Tutorial* [online]. Listopad 2009 [cit. 2020-05-07]. Dostupné z: https://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf.

[14] Uppsala University a Aalborg University. *Uppaal Tron* [online]. 2012 [cit. 2020-05-07]. Dostupné z: https://people.cs.aau.dk/~marius/tron/.

[15] Uppsala University a Aalborg University. *Uppaal Cora* [online]. 2014 [cit. 2020-05-07]. Dostupné z: https://people.cs.aau.dk/~adavid/cora/index.html.

[16] Uppsala University a Aalborg University. *Uppaal Tiga* [online]. 2014 [cit. 2020-05-07]. Dostupné z: http://people.cs.aau.dk/~adavid/tiga/index.html.

[17] Uppsala University a Aalborg University. *Uppaal* [online]. 2015 [cit. 2020-05-07]. Dostupné z: http://www.uppaal.org/.

[18] Uppsala University a Aalborg University. *Uppaal Stratego* [online]. 2015 [cit. 2020-05-07]. Dostupné z: https://people.cs.aau.dk/~marius/stratego/.

[19] Uppsala University a Aalborg University. *Uppaal Timed Automata Parser Library* [online]. 2018 [cit. 2020-05-07]. Dostupné z: http://people.cs.aau.dk/~adavid/utap/.

Příloha A

Soubor XML

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC ’-//Uppaal Team//DTD Flat System 1.1//EN’
’http://www.it.uu.se/research/group/darts/uppaal/flat-1_2.dtd’>
<nta>
<declaration>
chan take, release;     // Take and release torch
int[0,1] L;     // The side the torch is on
clock time;     // Global time
</declaration>
    <template>
        <name x="32" y="16">Soldier</name>
        <parameter>const int delay</parameter>
        <declaration>clock y;</declaration>
        <location id="id0" x="288" y="80">
        </location>
        <location id="id1" x="288" y="216">
            <name x="272" y="232">safe</name>
        </location>
        <location id="id2" x="64" y="216">
           ...
        </location>
        <init ref="id3"/>
        <transition>
            <source ref="id2"/>
            <target ref="id3"/>
            <label kind="guard" x="72" y="176">y &gt;= delay</label>
            <label kind="synchronisation" x="72" y="120">release!</label>
        </transition>
        <transition>
            ...
        </transition>
    </template>
    <template>
        <name x="32" y="-16">Torch</name>
        ...
    </template>
    <system>const int fastest = 5;
const int fast    = 10;
const int slow    = 20;
const int slowest = 25;

Viking1 = Soldier(fastest);
Viking2 = Soldier(fast);
```

```
Viking3 = Soldier(slow);
Viking4 = Soldier(slowest);

system Viking1, Viking2, Viking3, Viking4, Torch;</system>
    <queries>
        <query>
            <formula>A[] not deadlock</formula>
            <comment>The system is deadlock free.</comment>
        </query>
        <query>
            ...
        </query>
        <query>
            <formula>E&lt;&gt; Viking1.safe and Viking2.safe
and Viking3.safe and Viking4.safe</formula>
            <comment>Scheduling problem reformulated as reachability property. Use
'Diagnostic Trace:Fastest' option to find fastest solution.</comment>
        </query>
    </queries>
</nta>
```

Příloha B

Formát XTR

```
3   \
0    \    poradi cisel udava cislo procesu
3     | - a cislo udava, ve kterem stavu se
3    /    proces nachazi
1   /
.
0  \
1   \
0    |
.    |
0    |
3    |
0    |
.    |
1    |
2    |
0    |
.    |
2    |
4    | - sekce "zona", ktera obsahuje podminky
0    |   nad hodinami
.    |
3    |
0    |
0    |
.    |
4    |
5    |
0    |
.    |
5    |
1   /
0  /
.
.
0    | - hodnoty promennych
.
1 3 ;  - prechody, ktere byly pouzity, aby se
4 0 ;    system dostal do tohoto stavu
.
```

Příloha C

Formát IF

```
layout
#index:const:value
#index:clock:nr:name
#index:var:min:max:init:nr:name
#index:meta:min:max:init:nr:name
#index:cost
#index:location:flags:name
#index:meta:min:max:name
0:clock:0:t(0)
1:const:1
3:var:0:1:0:0:L
4:clock:1:time
11:clock:2:Viking1.y
13:location::safe
15:location::unsafe
38:location::one
39:location:urgent:_id5
40:location::free
41:location::two
.....

instructions
#address: opcode
0: 0 1          push 1
2: 44           halt
3: 0 0          push 0
5: 44           halt
6: 0 1          push 1
8: 44           halt
9: 0 1          push 1
11: 44          halt
12: 0 0         push 0
....

processes
#index:initial:name
0:15:Viking1
1:22:Viking2
2:29:Viking3
3:36:Viking4
4:40:Torch
```

```
locations
#index:process:invariant
12:0:6
13:0:15
14:0:24
15:0:33
19:1:126
20:1:135
21:1:144
22:1:153
.....

edges
#process:source:target:guard:sync:update
0:14:15:39:56:53
0:13:14:62:75:71
0:12:13:81:98:95
0:15:12:104:117:113
1:21:22:159:176:173
1:20:21:182:195:191
1:19:20:201:218:215
1:22:19:224:237:233
2:28:29:279:296:293
.....

expressions
#address:reads:writes:text
0:::1
6:::1
39:10,11::y >= delay
47:11::y
50:10::-delay
56:::release!
62:3::L == 1
71::11:y = 0
75:::take!
81:10,11::y >= delay
89:11::y
92:10::-delay
98:::release!
.....
```

Příloha D

Struktura navržené reprezentace

```
{
    "States": {
        "0": {
            "processes": [
                {
                    "name": "<jmeno procesu>",
                    "location": {
                        "name": "<aktualni stav>",
                        "type": "[common|urgent|committed]"
                    },
                    "invariant": "<vyraz>",
                    "vars": [ // lokalni promenne
                        {
                            "name": "<jmeno promenne>",
                            "value": "<hodnota promenne>"
                        }
                        // ...
                    ]
                },
                // ...
            ],
            "vars": [ // globalni promenne
                {
                    "name": "<jmeno promenne>",
                    "value": "<hodnota promenne>"
                }
                // ...
            ],
            "clocks": [ // globalni hodiny
                "<vyraz omezeni>",
                // ...
            ]
        },
        "2": {
            // ...
        }
        // ...
    },
```

```
    "Transitions": {
        "1": [
            {
                "process": "<jmeno procesu>",
                "from": "<pocatecni stav>",
                "to": "<cilovy stav>",
                "guards": "<vyraz>",
                "sync": "<vyraz>",
                "updates": "<vyraz>"
            },
            // ...
        ],
        "3": [
            // ...
        ],
        // ...
    }
}
```

Příloha E

Nápověda aplikace

```
        TraceConverter xmlFile ifFile xtrFile [option]...

        TraceConverter jsonFile [option]...
        (needs file created with merge option 0)
OPTION
        -o file
                Redirects output to file instead of console.
        -m [0|1|2]
                How to merge states and transitions in final JSON
                * 0 - (default) first all states and then all transitions
                * 1 - init state and then (transition, state)+
                * 2 - init state and then (state, transition)+
        -I
                Output will not be intended (formated)
        -s
                Only states in output file
        -t
                Only transitions in output file
        -b item;item;...
                Blacklist specified items. Items are separated by ';'
        -bf file
                Blacklist specified items from file. Items are on lines (one item on one line)
        -w item;item;...
                Whitelist specified items. Items are separated by ';'
        -wf file
                Whitelist specified items from file. Items are on lines (one item on one line)
FILTER ITEM SYNTAX
        Location:
                location:<locationName>
                location:<processName>.<locationName>
        Edge:
                edge:<from>.<to>
                edge:<processName>.<from>.<to>
        Variable:
                local:
                localVar:<varName>.<from>.<to>
                localVar:<processName>.<varName>.<from>.<to>

                global:
                globalVar:<varName>.<from>.<to>
```

```
Clock:
        local:
        localClock:<clockName>.<from>.<to>
        localClock:<processName>.<clockName>.<from>.<to>

        global:
        globalClock:<clockName>.<from>.<to>
Channels:
        chan:<chanName>
        chan:<processName>.<chanName>
        (<chanName> without '!','?' means both options)

(location, edge, localVar, chan without 'processName' means all processes)
```