Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra měření

Rozšíření testovacího nástroje Taster

Ondřej Kobza

Květen 2018
Vedoucí práce: Ing. Jan Sobotka, Ph.D

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kobza** Jméno: **Ondřej** Osobní číslo: **457004**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozšíření testovacího nástroje Taster

Název bakalářské práce anglicky:

Taster Testing Tool Extension

Pokyny pro vypracování:

1. Seznamte se softwarovým nástrojem Taster.
2. Definujte podmnožinu jazyka UPPAAL podporovanou nástrojem Taster.
3. Upravte použití modelovací jazyk, tak aby byl zpětně kompatibilní s nástrojem UPPAAL.
4. Rozšiřte množinu podporovaných konstrukcí použitého modelovacího jazyka o podporu ?urgent? a ?commited? stavů, C struktur a volání funkcí.
5. Upravte testovací nástroj tak, aby umožnoval náhodné časování událostí.
6. Rozšiřte syntaktický analyzátor dle aktuální podoby modelovacího jazyka.
7. Výsledky demonstrujte vhodnou ukázkou.

Seznam doporučené literatury:

[1] GRUS, Tomáš: Implementace softwarového nástroje pro generování integračních testů. Praha, 2014. Diplomová práce ČVUT.
[2] J. Zander, I. Schieferdecker, and P.J. Mosterman: Model-Based Testing for Embedded Systems. Taylor & Francis, 2011.
[3] Johan Bengtsson, and Wang Yi: Timed Automata: Semantics, Algorithms and Tools. Lecture Notes in Computer Science. 2004.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jan Sobotka, Ph.D., katedra měření FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **08.01.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce:
do konce letního semestru 2018/2019

Ing. Jan Sobotka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

Poděkování / Prohlášení

Děkuji vedoucímu své bakalářské práce Ing. Janu Sobotkovi Ph.D za odborné vedení, pomoc a rady při zpracování této práce.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt / Abstract

Cílem této bakalářské práce je rozšířit nástroj Taster. Rozšíření spočívá v implementaci náhodného časování událostí, dále pak nového intepretu modelovacího jazyka a syntaktického analyzátoru, jehož gramatika generuje modelovací jazyk. Zároveň by formát modelů pro Taster měl být co nejvíce kompatibilní s formátem modelů pro program UPPAAL, tj. modelovací jazyk Tasteru by měl být co nejvíce kompatibilní s modelovacím jazykem UPPAALu. Práce nejprve zkoumá časované automaty, dále nástroj UPPAAL a Taster, následně se zabývá implementací jednotlivých bodů zadání, tj. implementace interpretu modelovacího jazyka, implementace náhodného časování událostí a implementace podpory pro urgent a committed locations. Nakonec je funkčnost celé implementace demonstrována na ukázkovém modelu.

Klíčová slova: Taster, UPPAAL, Časované automaty, EXAM, Interpret, Gramatika, Syntaktický analyzátor

The main goal of this bachelor thesis is to extend Taster test tool. The extension consists of implementation of random timing of events, a new interpret of the modeling language and of a new syntactical analyzer, whose grammar generates that modeling language. At the same time, the Taster modeling language should be compatible as much as possible with the UPPAAL modeling language. The thesis first studies timed automata, the UPPAAL and Taster tool, then it deals with the implementation of individual entry points from the assignemnt, ie. implementation of modeling language interpret, implementation of random timing of events and implementation of support for urgent and committed locations. Finally, the functionality of the entire implementation is demonstrated on sample model.

Keywords: Taster, UPPAAL, Timed automata, EXAM, Interpret, Grammmar, Syntactical analyzer

Title translation: Taster Testing Tool Extension

Obsah /

Kapitola 1
Úvod

Cílem této bakalářské práce je rozšířit nástroj Taster o náhodné časování událostí, dále pak o nový intepret modelovacího jazyka a o syntaktický analyzátor, jehož gramatika generuje modelovací jazyk. Zároveň by měl formát modelů pro Taster být co nejvíce kompatibilní s formátem modelů pro program UPPAAL.

UPPAAL slouží k tvorbě, simulaci a testování tzv. časovaných automatů (timed automata).

Taster byl vyvinut pod vedením pana Ing. Jana Sobotky Ph.D. na Katedře měření Fakulty elektrotechnické ČVUT v Praze. Nástroj slouží k integračnímu testování automobilových řídících jednotek.

Integrační testování je následně prováděno na základě modelu systému a okolí. Tento přístup se obvykle označuje jako „Model-Based Testing“. Propojení nástroje s reálným hardwarem je řešeno v Tasteru komunikací se softwarem EXAM, který se používá pro tzv. „hardware-in-the-loop“ (HIL) testování [1].

Metoda HIL se obecně používá pro simulaci elektromechanických systémů, které jsou řízeny elektronickou řídící jednotkou [2]. Pro HIL testování podporuje Taster kromě komunikace se softwarem EXAM také komunikaci s programem VeriStand.

1.1 Použité názvosloví

Jelikož pro mnoho anglických názvů neexistuje odpovídající český ekvivalent, budu pro tyto názvy používat anglické termíny (například urgent locations atp.).

Kapitola 2
Úvod do časovaných automatů

Časovaný automat je speciální typ tzv. přechodového systému (Transition system), který je rozšířený o časová omezení (tzv. Transition system with timing constraints) [3], [4].

2.1 Přechodový systém

Přechodový systém je uspořádaná n-tice (Q, Q_0, Σ, T), kde

- Q je množina stavů
- $Q_0 \subseteq Q$ je množina počátečních stavů
- Σ je množina symbolů (nebo událostí), zvaná abeceda
- $T \subseteq Q \times \Sigma \times Q$ je množina přechodů

Množina přechodů T pro každý stav q určuje, za jakých podmínek a do kterých stavů lze ze stavu q přejít. Například, mějme přechod (q_1, α, q_2), ten říká, že je-li přechodový systém ve stavu q_1, může při události α změnit svůj stav z q_1 do q_2.

Tyto přechody lze také vyjádřit tzv. přechodovou funkcí δ, což je zobrazení:

$$\delta : Q \times \Sigma \to Q$$

2.1.1 Komplexní přechodový systém

Komplexní přechodový systém vznikne spojením více přechodových systémů. Mějme přechodové systémy:

$$S_1 = (Q_1, Q_{1,0}, \Sigma_1, T_1), S_2 = (Q_2, Q_{2,0}, \Sigma_2, T_2), ..., S_n = (Q_n, Q_{n,0}, \Sigma_n, T_n)$$

Výsledný komplexní přechodový systém vzniklý pomocí těchto dílčích přechodových systémů bude uspořádaná n-tice:

$$(Q_1 \times Q_2 \times ... \times Q_n, \Sigma_1 \cup \Sigma_2 \cup ... \cup \Sigma_n, T_1 \cup T_2 \cup ... \cup T_n)$$

Symboly patřící m přechodovým systémům, kde $m > 1$, slouží pro synchronizaci těchto přechodových systémů (tzv. synchronizace hran).

2.2 Přechodový systém s časovými omezeními

Časovaný přechodový systém je n-tice $(Q, Q_0, \Sigma, X, I, \Phi(X), T)$, kde:

- Q je množina všech stavů
- Q_0 je množina všech počátečních stavů
- Σ je abeceda
- X je množina hodinových proměnných

- I je zobrazení, které přiřazuje některým stavům z Q některá hodinová omezení z $\Phi(X)$. Těmto omezením se říká invarianty.
- $\Phi(X)$ je množina hodinových omezení φ (nebo-li podmínek) ve tvaru:

$$\varphi := x \leq c \quad | \quad c \leq x \quad | \quad x < c \quad | \quad c < x \quad | \quad \varphi_1 \wedge \varphi_2$$

  Proměnná x je hodinová proměnná z množiny X a c je nezáporné racionální číslo. Tato omezení se nazývají stráže (guards).
- $T \subseteq Q \times \Sigma \times 2^C \times \Phi(C)$ je množina přechodů.

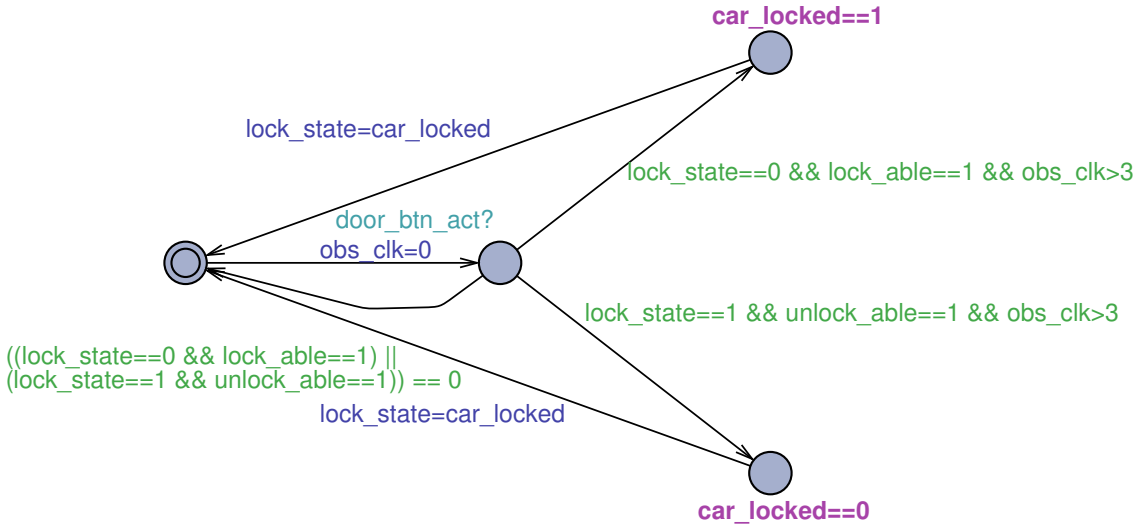
Přechod v takovém systému bude čtveřice $(q_1, \alpha, \lambda, \varphi,_2)$ reprezentující hranu ze stavu q_1 do stavu q_2 při události α za hodinových omezení φ. λ je množina prvků z množiny X které mají být resetovány.

Výše popsaný časovaný přechodový systém je časovaným automatem.

UPPAAL umožňuje vytvořit i komplexní časovaný přechodový systém, jež je dále v textu nazýván jako „model“. Jednotlivé automaty v každém modelu mohou obsahovat pouze jeden počáteční stav. Navíc, stráže a invarianty v automatech pro UPPAAL nemusí obsahovat pouze hodinová omezení.

Kapitola 3
Popis jazyka UPPAAL

Jednotlivé modely mohou obsahovat jeden nebo více časovaných automatů. Každý automat lze reprezentovat grafem, jenž je určen množinou uzlů (reprezentující jednotlivé stavy automatu) a množinou orientovaných hran (jež reprezentují možné přechody), které spojují ony uzly. Následující text pohlíží na časované automaty jako na grafovou datovou strukturu. Jak takový automat může vypadat ilustruje následující obrázek:



Obrázek 3.1. Příklad časovaného automatu

V následující podkapitole je popsáno, jak je model časovaných automatů reprezentován v nástroji UPPAAL, jenž byl vyvinut na švédské univerzitě Uppsala a dánské univerzitě Aalborg:

3.1 Nástroj UPPAAL

Každý model je v UPPAALu rozdělen na několik částí:

- **Template (šablona)** - samotný automat. Každý model může obsahovat více šablon, nebo-li automatů, které mohou být vzájemně propojené synchronizovanými hranami.
- **Global Declarations (globální deklarace)** - zde se deklarují globální proměnné a definují globální funkce. K těm pak lze přistupovat v rámci celého modelu.
- **Local declarations (lokální deklarace)** - zde se deklarují lokální proměnné a definují lokální funkce. Tyto deklarace přísluší k dané šabloně, resp. automatu.
- **System declarations (systémové deklarace)** - slouží zejména pro vytvoření instancí jednotlivých šablon.

Jak pro globální, tak i pro lokální deklarace se používá notace jazyka C. Nástroj Taster ověřuje správnou syntaxi pomocí syntaktického analyzátoru, který lze automaticky vygenerovat z gramatiky (generující modelovací jazyk) do kódu jazyka C# (v němž je celý Taster naprogramován) pomocí nástroje *antlr4* [5].

3.2 Jazyk UPPAALu

V rámci šablon, globálních, lokálních a systémových deklarací podporuje UPPAAL následující jazykové konstrukty: (Pozn.: pro každý jazykový konstrukt je v závorce uvedeno, zda byl před vznikem této bakalářské práce Tasterem podporován, či nikoli)

- **Bounded integer variables (nepodporováno):**

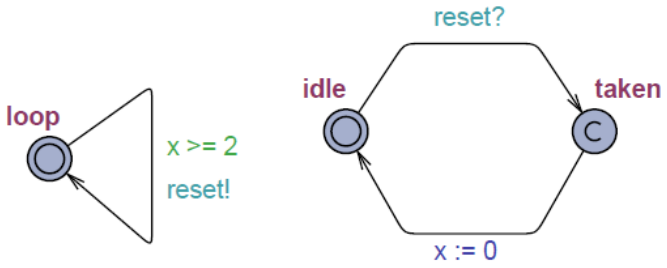
```
int[min,max] name;
```

  kde „min“ je dolní a „max“ je horní hranice rozsahu povolených hodnot. Proměnná „name“ tak může nabývat hodnot pouze v rozmezí od „min“ po „max“.
- **Binary synchronization (podporováno):** pokud nadeklarujeme proměnnou typu „chan“:

```
chan c;
```

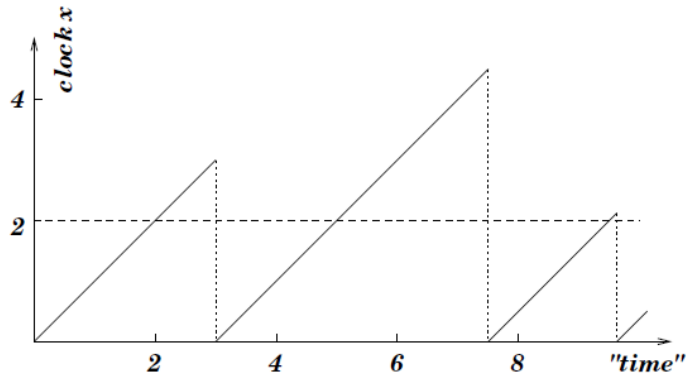
  pak hrana označena se synchronizační proměnnou ve tvaru „c!“ je synchronizována s hranou s toutéž proměnnou ve tvaru „c?“.
  Na následujícím příkladu je ilustrován princip synchronizace hran:

```
chan reset;
clock x;
```



Obrázek 3.2. Příklad modelu v UPPAALu. Vlevo je automat L1, vpravo automat Observer [6].

Uvažujme, že Observer je v uzlu „idle“. Pak čeká, než v automatu L1 bude vzata hrana se synchronizační proměnnou „reset!“. Na tuto událost zareaguje Observer a vezme hranu s proměnnou „reset?“. Proměnná „x“ reprezentuje hodiny. Následující diagram ukazuje, jak se bude proměnná „x“ měnit:



Obrázek 3.3. Chování modelu v čase [6].

- **Broadcast channels (nepodporováno):** mějme:

```
broadcast chan c;
```

  pak na jednoho odesílatele „c!“ může být více příjemců „c?“.
- **Location (podporováno):** jedná se o jednotlivé stavy v daném automatu. Jelikož se však termínem *stav* v UPPAALu myslí množina locations taková, že z každého automatu daného modelu je v množině právě jedna location (máme-li například model o třech automatech, v prvním automatu je aktuální location *l11*, ve druhém *l21* a ve třetím *l31*, tak množina $\{l11, l21, l31\}$ definuje aktuální stav modelu), budu dále v textu používat termín „uzel“.
  Uzel může mít tři atributy, *initial, urgent a comitted*, přičemž *initial location* označuje počáteční stav (resp. uzel).

  - **Invariant (podporováno):** jedná se o podmínku, která říká, že v daném uzlu musí být splněny jisté vlastnosti (například hodnota časové proměnné musí být větší/menší, než zvolená hodnota atp.). Abychom mohli být v nějakém uzlu, musí být splněny všechny jeho invarianty.
  - **Urgent locations (nepodporováno):** pro *urgent location* platí, že čas nemůže plynout, pokud je systém v nějakém takovém uzlu.
  - **Comitted locations (nepodporováno):** platí totéž, co pro *urgent location*, navíc však následující krok musí být učiněn z nějakého „committed“ uzlu.
- **Edge (podporováno):** orientovaná hrana spojující dva uzly. Na hraně se mohou vyskytovat následující výrazy:

  - **Guards (podporováno):** nebo-li stráž. Pokud má nějaká hrana stráž, může být použita pouze, je-li stráž splněna. Jedná se tedy o podmínku, jako například $x < 5$ atp.
  - **Synschronization (podporováno):** každá hrana může být označena synchronizační proměnnou ve tvaru c! nebo c?.
  - **Assignment (podporováno):** jedná se o přiřazení hodnoty do proměnné (tato hodnota může být mj. i návratovou hodnotou nějaké funkce) nebo o samostatné volání funkce.

- **Initialisers (nepodporováno):** funkcionalita umožňující přiřazení hodnoty do proměnné. Například:

```
int i=2;
int i[3]={1,2,3};
```

- **Functions (nepodporováno):** tělo funkce se v UPPAALu skládá z výrazů (expressions), případně z cyklů (for, while, do-while) a podmínek (if nebo if-else). Funkce se syntakticky definuje stejně, jako v jazyce C:

```
type name_of_function(args){expressions;}
```

Jsou deklarovány v *Declarations* (lokálních i globálních).

- **Datové typy:**
  - **int (podporováno):** integer, tedy celočíselný 32bitový datový typ.
  - **bool (podporováno):** proměnná typu bool může nabývat pouze dvou hodnot: „True“ a „False“.
  - **clock (podporováno):** definuje hodiny.
  - **chan (podporováno):** proměnná typu chan je synchronizační proměnnou.
    - **urgent chan (nepodporováno):** pokud je nějaká hrana svázána s proměnnou typu `urgent chan` a jsme ve stavu, kdy čekáme na událost (příklad: máme proměnnou *urgent chan go*. V uzlu, z nějž vede hrana označena synchronizační proměnnou *go?* čekáme na událost *go!*). Pak po příchodu této události se musí vzít ona hrana označená *go?* bez jakéhokoli časového zpoždění.
      Pozn.: Výraz „vzít hranu“ znamená udělat krok z uzlu U_1 po vybrané hraně E do uzlu U_2, který je touto hranou E spojen s uzlem U_1.
  - **const (nepodporováno):** proměnná s nezměnitelnou hodnotou. Příklad:

```
const int a=1; //Přiřadí do proměnné a hodnotu 1.
//Dále do této proměnné již žádná jiná hodnota přiřadit nepůjde.
```

  - **void (nepodporováno):** používá se pro funkce, které nevrací žádnou hodnotu.
- **Datové struktury (nepodporováno):**
  - **Arrays (nepodporováno):** pro deklaraci pole použijeme klasickou notaci jazyka C, například:

```
int array[3];
```

  - **record variables (nepodporováno):** jedná se o struktury, syntaxe je následovná: $(struct\{int \quad a; ...\}name;)$
  - **scalars ne(podporováno):** množina tzv. „scalar values“, které mohou být pouze porovnávány na rovnost. Používají se pro tzv. Symetry reduction [7].
  - **typedef (nepodporováno):** tato jazyková konstrukce umožňuje definovat nové datové typy.

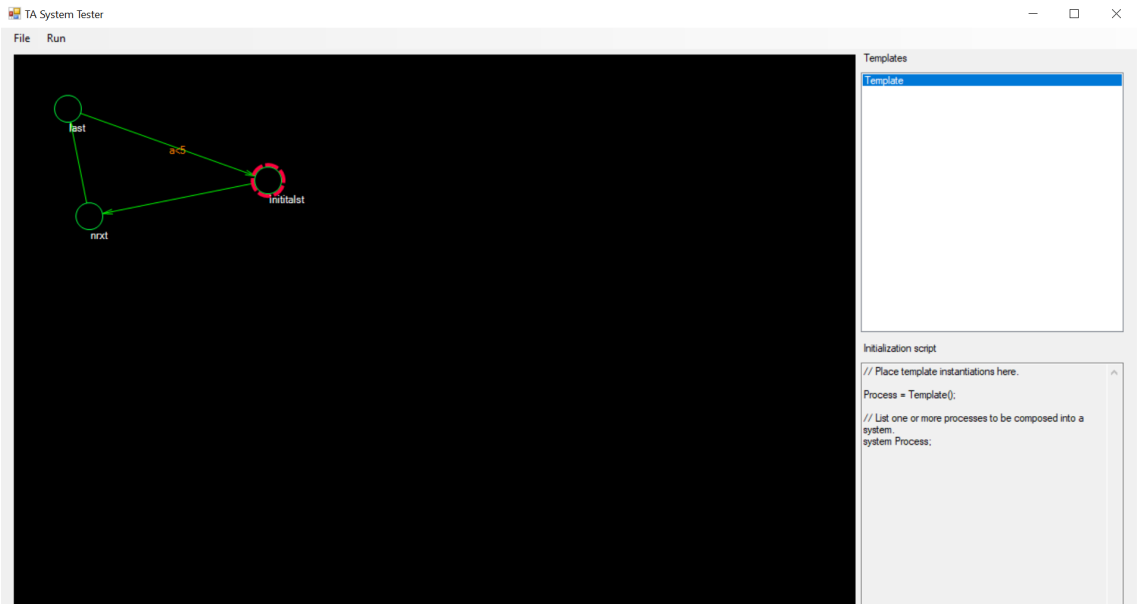
Kapitola 4
Seznámení se s nástrojem Taster

Program Taster se skládá z následujících částí:

- GUI (grafické rozhraní)
- XML parser
- Syntaktický a lexikální analyzátor
- Adaptéry
- Hodinový systém
- Systém pro reprezentaci modelu
- Logika zajišťující průchod modelu

4.0.1 GUI

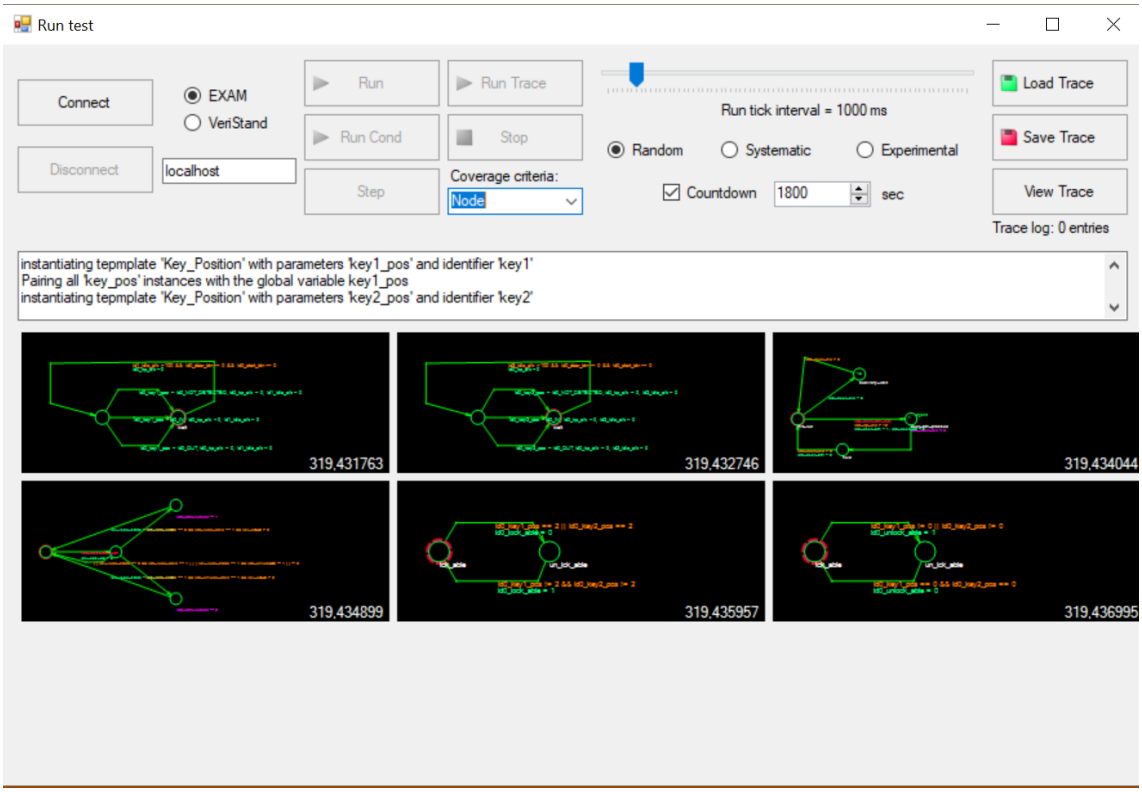
Grafické rozhraní Tasteru má dvě části. První je určena k výběru modelu popisující systém, jenž chceme testovat:



Obrázek 4.1. Úvodní část GUI nástroje Taster s načteným automatem

Druhá část slouží k samotnému testování systému, který je popsán vybraným modelem. Průběh průchodu modelem lze sledovat buď na obrázcích jednotlivých automatů nebo si lze prohlédnout celý záznam průchodu (tlačítko View Trace). Tento záznam lze uložit pro pozdější použití.

Ukázka:



Obrázek 4.2. Okno pro testování systému (reprezentovaným předem vybraným modelem)

4.1 XML parser

Knihovna *System.Xml* parsování XML značně zjednodušuje. Taster prochází jednotlivé XML elementy a postupně je zařazuje do své interní datové reprezentace. (Například pro XML objekt „transition“ vytvoří Taster nový objekt třídy „edge“ a vloží jej do objektu třídy „Template“).

4.2 Lexikální a syntaktický analyzátor

Kód lexikálního syntaktického analyzátoru je vygenerován v programovacím jazyce C# pomocí nástroje *antlr4*. Přičemž *antlr4* oba tyto analyzátory vygeneroval na základě definované gramatiky (která používá syntaxi dle *antlr4* a která generuje modelovací jazyk), jež má dvě části:

- pravidla pro Lexer
- pravidla pro Parser

4.2.1 Lexer (Lexikální analyzátor)

Lexer zajišťuje lexikální analýzu textového řetězce. Rozdělí vstupní řetězec na tzv. tokeny - lexikální jednotky (např. čísla, klíčová slova, operátory, či některé specifické znaky). Tyto tokeny jsou dále určeny ke zpracování pro Parser [8].

Příklad gramatiky pro lexer:

```
ID_NoDigitStart
: ( 'a'..'z' | 'A'..'Z' ) ('a'..'z' | 'A'..'Z' | '_' |Digit )*
;
```

4.2.2 Parser (syntaktický analyzátor)

Parser je definován bezkontextovou (CFG) gramatikou (2. typ gramatiky dle Chomského [9] dělení gramatik). Parser kontroluje, zda posloupnost symbolů tvoří gramatikou povolený výraz.

4.3 Adaptéry

Adaptéry slouží ke komunikaci s nástroji pro provádění integračních testů.

Taster obsahuje dva adaptéry

- **EXAMAdaptor:** přidává podporu pro nástroj EXAM
- **VeriStandAdaptor:** přidává podporu pro nástroj VeriStand

Exam i VeriStand jsou programy sloužící pro HIL [1] testování.

4.4 Hodinový systém

Hodinový systém Taster používá k simulaci času. Taster užívá knihovnu jazyka C# *System.Diagnostics* a hodiny reprezentuje objekt třídy *StopWatch*.

4.5 Systém pro reprezentaci modelu

V každém modelu je jeden nebo více automatů, jež jsou reprezentovány pomocí grafu. Každý automat (šablona) je reprezentován třídou Template. Základními prvky každého automatu v jeho grafové reprezentaci jsou uzly a hrany. Uzly i hrany jsou reprezentovány pomocí vlastních tříd (Node a Edge) a mezi jejich atributy patří například stráže nebo invarianty.

4.6 Algoritmus simulace modelu

Průchod je založen na pseudonáhodné strategii, kdy se nejprve vytvoří seznam hran vedoucích z aktuálního stavu, přičemž tento seznam obsahuje pouze hrany, pro něž jsou splněny jejich stráže. Následně se z tohoto seznamu náhodně vybere jedna hrana. Uzel, do kterého tato vybraná hrana vede se stane aktuálním uzlem pro daný automat, resp. šablonu. Při průchodu se pro každý nový aktuální uzel kontrolují invarianty. Pokud nějaký invariant není splněn, průchod se musí ukončit. Hlavní logiku průchodu zajišťuje metoda *public bool Step()*.

Kapitola 5
Rozšíření modelovacího jazyka

Nástroj *antlr4* slouží především k tvorbě syntaktických analyzátorů na základě gramatik. Gramatika pro *antlr4* musí být v souboru s příponou *g4*. *Antlr4* také specifikuje, jak má gramatika vypadat z hlediska syntaxe [5].
Z gramatiky dokáže nástroj *antlr4* vygenerovat kód v různých programovacích jazycích. Chceme-li například vygenerovat kód v jazyce C# z gramatiky „uppaal_C.g4“, uděláme to v příkazovém řádku následovně:

```
antlr4 -Dlanguage=CSharp uppaal_C.g4
```

Gramatiku „uppaal_C.g4“ pro modelovací jazyk (podporovaný Tasterem) jsem dostal k dispozici od svého vedoucího. Tuto gramatiku jsem rozšířil o další pravidla (dle specifikací Uppaalu [10]) a následně jsem pomocí nástroje *antlr4* vygeneroval *Lexer* a *Parser* v jazyce C#. Ona gramatika bude dále v textu nazývána jako „Uppaal_C“.

5.1 Použití Lexeru a Parseru

Mějme nějaký program v jazyce, který je generován gramatikou *Uppaal_C*. Tento program pak můžeme *Lexerem* tokenizovat a následně *Parserem* vytvořit derivační strom. Tento strom je pak možné po jednotlivých vrcholech procházet.
Kromě *Lexeru* a *Parseru* vygeneroval *antlr4* také tzv. „interface“: „uppaal_CListener“. Tento „interface“ deklaruje metody příslušné jednotlivým pravidlům pro *Parser*. Například pravidlu „Statement“ odpovídají následující dvě metody:

```
void EnterStatement([NotNull] uppaal_CParser.StatementContext context);

void ExitStatement([NotNull] uppaal_CParser.StatementContext context);
```

Argument *context* symbolizuje uzel v derivačním stromu. Obsahuje především pole *children*, jehož prvky odpovídají potomkům tohoto uzlu.

Pokud implementujeme tento „interface“ v nějaké třídě a následně projdeme derivační strom vytvořený *Parserem* (strom procházím použitím metody *ParseTreeWalker.Default.Walk(rule, tree)*, která je k dispozici v knihovně *Antlr4*), tak po navštívení nějakého uzlu z tohoto stromu se zavolá metoda příslušná pravidlu pro tento daný uzel. Například pokud navštívíme uzel příslušící pravidlu „Statement“, zavolá se metoda „void EnterStatement([NotNull] uppaal_CParser.StatementContext context)“. Průchod pokračuje do podstromů tohoto uzlu. Poté, co se projdou všechny tyto podstromy se zavolá funkce „void ExitStatement([NotNull] uppaal_CParser.StatementContext context)“.

5.2 Gramatika modelovacího jazyka

V následujících ukázkách gramatiky je použita notace podle nástroje *antlr4*.

Aby bylo možné vytvořit gramatiku pro syntaktický analyzátor, je třeba nejprve definovat jednotlivé základní elementy jazyka, tzv. tokeny. K tomu slouží pravidla pro lexikální analyzátor, viz příloha B.

Tato pravidla pak používá *Parser* ve svých vlastní pravidlech.

5.2.1 Pravidla pro *Parser*

V této části jsou uvedena nejdůležitější pravidla gramatiky *Uppaal_C*. Kompletní gramatika je dostupná v přiložených elektronických souborech a na přiloženém CD.

- **root:** kořen derivačního stromu každého programu (napsaném v jazyce, který je generován gramatikou *Uppaal_C*) odpovídá právě tomuto pravidlu:

```
root
: declarator*
| statement*
| expression
;
```

- **declarator:** toto pravidlo odpovídá deklaraci proměnné, deklaraci pole nebo definici funkce:

```
declarator
 : (typeSpecifier ID_NoDigitStart Semi
|  Int ID_NoDigitStart Assign Integer Semi
|  Bool ID_NoDigitStart Assign ('false'|'true'| Integer) Semi
|  fieldDecl
|  Int '[' Integer ',' Integer ']' ID_NoDigitStart Semi
|  Int '[' Integer ',' Integer ']' ID_NoDigitStart Assign Integer Semi
|  functionDefinition)
|  ID_NoDigitStart ID_NoDigitStart Assign initializer Semi
|  ID_NoDigitStart ID_NoDigitStart Semi
|  'struct' '{' structDeclarator '}' ID_NoDigitStart Semi
|  'struct' '{' structDeclarator '}'
             ID_NoDigitStart Assign initializer Semi
|  'typedef' 'struct' '{' structDeclarator '}' ID_NoDigitStart Semi
;
```

  - **fieldDecl:** pravidlo pro deklaraci pole:

```
fieldDecl
: typeSpecifier ID_NoDigitStart arrayDecl*
(',' ID_NoDigitStart arrayDecl*)* ';'
;
```

  - **functionDefinition:** pravidlo pro definici funkce. Definice funkce obsahuje typ funkce, její jméno, argumenty a tělo (block):

```
functionDefinition
:   typeSpecifierF ID_NoDigitStart '(' parameters ')' block;
```

- **statement:** do tohoto pravidla spadá tělo, resp. „block“ (např. tělo funkce, tělo while cyklu, atp.), dále pak „expression“ (například přiřazení do proměnné, hodnota proměnné, inkrementace proměnné atp.), while cyklus, for cyklus, if podmínka nebo výraz return:

```
statement
: block
| ';'
| expression ';'
| forLoop
| whileLoop
| ifStatement
| returnStatement
;
```

  - **block:** pravidlo pro blok kódu, nebo-li tělo nějakého výrazu (funkce, cyklu...). Blok se skládá především z elementů odpovídajících pravidlu *statement*:

```
block
: '{' declarator* statement* '}'
;
```

  - **expression:** jedno z nejdůležitějších pravidel. Zachycuje například přiřazení do proměnné, či do pole, matematické a logické výrazy nebo inkrementaci proměnné:

```
expression
: ID_NoDigitStart
| Integer
| expression '[' expression ']'
|  expression '"'
|  '(' expression ')'
|  expression '++' | '++' expression
|  expression '--' | '--' expression
|  expression '(' arguments ')'
|  expression '(' ')'
|  Unary expression
|  expression Binary expression
| expression PlusMinus expression
| expression Shift expression
| expression Relation expression
| expression Ekvivalence expression
| expression LogicalH expression
|  expression '?' expression ':' expression
|  ID_NoDigitStart '.' ID_NoDigitStart '[' expression ']'
|  ID_NoDigitStart '.' ID_NoDigitStart
| expression '(' printArg ')'
| expression Assign expression
|  ('deadlock' | 'true' | 'false')
;
```

  - **forLoop:** pravidlo, které definuje for cykly. For cyklus se skládá především z klíčového slova „for“, tří výrazů a bloku (těla):

```
forLoop
: 'for' '(' expression ';' expression ';' expression ')' block
;
```

- **whileLoop:** pravidlo, které definuje while cykly. While cyklus může, ale nemusí obsahovat blok kódu:

```
whileLoop
: 'while' '(' expression ')' ';'
| 'while' '(' expression ')' block
;
```

- **ifStatement:** pravidlo definující, jak má vypadat if podmínka:

```
ifStatement
: 'if' '(' expression ')' block 'else' block
| 'if' '(' expression ')' block
;
```

- **returnStatement:** pravidlo pro výraz „return“:

```
returnStatement
: 'return' expression ';'
;
```

5.3 Reprezentace modelovacího jazyka třídami

V této části jsem se snažil řídit *Uppaal_C* gramatikou a také jsem vycházel z programu UPPAAL. Chceme-li v UPPAALu nadeklarovat nějaké proměnné nebo nadefinovat funkce, uděláme tak v části *Declaration*.
Základní třída se tedy jmenuje „Declaration“.

Pozn.: V následujících částech této sekce se slovem „výraz“ v této myslí vše, co přijímají pravidla „statement“ a „expression“ gramatiky *Uppaal_C*.

5.3.1 Declaration

Třída Declaration se skládá z následujích atributů:

- **proměnné:** jsou reprezentovány slovníkem. Klíčem je vždy jméno proměnné a hodnotou je instance třídy *Variable*. Atributy třídy *Variable* uchovávají informaci o typu proměnné a její aktuální hodnotě.
- **bounded integers:** jedná se o spojový seznam (*LinkedList*), kde jednotlivé elementy jsou instance třídy „BoundedInteger“. Atributy třídy „BoundedInteger“ říkají, jaká je aktuální hodnota proměnné tohoto typu, jaké jsou její meze a především, jaké je její jméno.
- **arrays:** interpret podporuje pouze jednodimenzionální pole, která jsou reprezentována třídou „Array“. Atribut *arrays* je ve třídě Declaration dán spojovým seznamem.
- **initialValues:** slovník, kde klíčem je vždy jméno proměnné a hodnotou instance třídy *Variable*, kde hodnota atributu „value“ v této instanci (třída *Variable* obsahuje atributy *value* a *type*) je hodnota proměnné, se kterou byla tato proměnná deklarována. Pokud chceme resetovat systém, vrátíme všechny proměnné právě na příslušné hodnoty v tomto slovníku.
- **functions:** spojový seznam, jehož prvky jsou instance třídy „Function“.
- **structures:** spojový seznam pro objekty třídy „Structure“. Jejich význam je blíže vysvětlen v 5.3.19.

- **types:** seznam jednotlivých nadefinovaných typů (pomocí konstruktu „typedef“). Proměnné tak mohou mít právě některý z těchto typů (kromě standardních $int, bool, clock, chan$). V Tasteru je proměnná nově nadefinovaného typu instancí třídy „Structure“.

5.3.2 Function

Mezi atributy třídy Function patří:

- **declaration:** odkaz na objekt třídy *Declaration*, v němž se daná instance třídy *Function* nachází.
- **name:** jméno funkce. (*string*)
- **Type:** udává návratový typ funkce. (*string*)
- **Body:** objekt třídy *Body* reprezentující tělo funkce.
- **Args:** atribut typu $Dictionary < string, VariableType >$. Symbolizuje argumenty, se kterými se funkce volá. Každý takový argument je definován svým jménem a typem.
- **arrays:** slovník, kde klíčem je jméno pole a hodnotou je instance třídy *Array*.
- **type[]:** pole integerů, kde každý prvek v poli může nabývat pouze čtyř hodnot: 0,1,2,3. Podle toho se určí, zda je daný argument, který ona funkce přijímá obyčejná hodnota, reference na proměnnou, reference na pole nebo reference na uživatelský datový typ (definovaný jako struktura nebo pomocí konstruktu *typedef*). Následně se dá při volání funkce detekovat, zda jsou všechny obdržené argumenty typově v pořádku.

Dále obsahuje třída Function atributy označené jako *private*, jsou to:

- **arguments:** atribut, jenž je objektem struktury *Arguments*, definované přímo ve třídě *Function*:

```
 [Serializable]
private struct Arguments
{
public Dictionary<string, Variable> arg;
public Dictionary<string, BoundedInteger> boundedInteger;
public Dictionary<string, Array> arrayArg;
public List<Structure> structs;
}
```

Z kódu je patrno, že tento atribut symbolizuje argumenty, které funkce přijímá, přičemž těmi argumenty může být jednoduchá proměnná, vázaná proměnná, pole, či struktura.

- **copyStructNames:** jedná se o seznam řetězců. Tento atribut je používán v metodě *private void Prepare()*, kde (v případě, že mezi argumenty je struktura) se jméno struktury (jež je v argumentu volání funkce) změní na jméno, se kterým pak funkce pracuje. Například, je-li funkce definována jako:

```
void function(MyStructure str){
some code;
}
```

a my zavoláme tuto funkci a jako argument jí předáme proměnnou *myStr* (která je například definována v globálních deklaracích), tak metoda *Prepare()* ve třídě Function nahradí jméno *myStr* za *str* a řetězec *myStr* si uloží do seznamu *copyStructNames*.

Po vykonání těla funkce se tato struktura přejmenuje zpět na své původní jméno. Podobný princip se používá i u ostatních ukazatelů, jako ukazatelů na pole, či proměnnou. Veškeré zpracování argumentů funkcí zajišťuje metoda *private void Prepare()*.

Funkci (objekt třídy funkce) lze „spustit“ metodou *Execute*(). Tato metoda zavolá nejprve metodu *Prepare*() a následně metodu *Evaluate*() na objektu *Body*. Nakonec zavolá metodu *private void updateSrructs()* která obnoví původní jména přejmenovaných struktur.

5.3.3 VariableType

Výčtový typ, který udává jakého typu proměnná, či funkce je.

```
   public enum VariableType
{
Integer,
Boolean,
Channel,
Clock
}
```

5.3.4 Body

Tato třída symbolizuje tělo funkce nebo také tělo for/while cyklu, či if podmínky. Jejími atributy jsou především lokální proměnné a spojový seznam výrazů (*LinkedList* < *Expression* >):

- **declaration:** odkaz na objekt třídy *Declaration*, k němuž daná instance třídy Body patří.
- **bodyVars:** lokální proměnné. (Lokální proměnná může být deklarována v těle funkce, přičemž nejdříve musí být ve funkci deklarace a až poté případně nějaké výrazy). Tento atribut je tedy relevantní pouze, jedná-li se o tělo funkce.
- **arrays:** lokální pole.
- **boundedInteger:** lokální vázané proměnné.
- **expressions:** výrazy, které toto „tělo“ obsahuje.

Tato třída obsahuje metodu „Evaluate()“, která prochází jednotlivé elementy atributu *expressions*, jež se postupně zpracovávají.

5.3.5 Expression

Mezi atributy této třídy patří, stejně jako u třídy *Body*, objekt třídy *Declaration*. Dále pak atribut „type“, který říká, jakého typu objekt třídy *Expression* je. Od této třídy totiž dědí několik dalších tříd, dle atributu *type* se může případně instance třídy *Expression* přetypovat na instanci některé ze tříd, které od *Expression* dědí. Atribut *type* je typu „ExpressionType“, což je výčtový typ implementován následovně:

```
public enum ExpressionType
{
Assign, MathExpr, BoolExpr, If, While, For, Ret, Cont, Break,
Ternary, FunctionCall, Val, Argument, BoolVal, Name, ArrayAssign,
ArrayVal, Null, StructVal, StructAssign, StructArrayVal,
StructArrayAssign, TestPrint
}
```

K většině hodnot tohoto výčtového typu existuje příslušná třída, dědící od *Expression*.

Expression má jedinou metodu *Evaluate()*, která podle atributu *type* přetypuje právě aktuální instanci na typ příslušné třídy dědící od *Expression* a zavolá metodu *Evaluate()*, jež je definována v oné dědící třídě. Například, pokud *typ* odpovídá hodnotě *ExpressionType.MathExpr*, metoda *Evaluate()* (třídy *Expression*) přetypuje aktuální instanci a zavolá stejnojmennou metodu *Evaluate()*:

```
((MathExpr)this).Evaluate();
```

Hodnotu, kterou příslušná metoda *Evaluate()* vrátí, vrátí také metoda *Evaluate()* třídy *Expression*.

Od třídy Expression dědí mnoho dalších tříd:

- Assign
- ArrayAssign
- MathExpr
- BoolExpr
- IfCondition
- WhileLoop
- ForLoop
- Ret
- FuntionCall
- Val
- ArrayVal
- StructVal
- StructAssign
- StructArrayVal
- StructArrayAssign

5.3.6 Assign

Kromě již zděděných atributů obsahuje atribut „string nameOfVar“, který říká, do jaké proměnné se bude přiřazovat. Dalším atributem je atribut „Expression expr“, což symbolizuje výraz, jehož hodnota se přiřadí do proměnné. Tato třída tedy symbolizuje přiřazení, kdy se do nějaké proměnné *nameOfVar* přiřadí výraz *expr*. Za tímhle účelem přepisuje třída *Assign* metodu *Evaluate()* tak, že metoda nejdříve zkusí najít příslušnou proměnnou a v případě úspěchu do ní přiřadí výslednou hodnotu výrazu *expr*:

```
variable.value=expr.Evaluate(); //variable is of the type Variable
```

V případě, že proměnná nebyla nalezena nebo proměnná je typu *BoundedInteger* a hodnota, která se má přiřadit se nevejde do jejích mezí, je vyhozena výjimka.

5.3.7 ArrayAssign

Třída, která má podobnou funkci jako třída *Assign* s tím rozdílem, že se přiřazuje hodnota do pole na nějaký index. Jejími atributy tedy jsou:

- **name:** textový řetězec, udávající jméno pole.
- **index:** atribut typu *Expression*, který určuje, na jaké místo v poli se má hodnota zapsat.
- **expr:** výraz, jehož hodnota se má zapsat do pole.

Tato třída má opět jednu jedinou metodu *Evaluate()*, která přepisuje stejnojmennou metodu rodičovské třídy *Expression*.

Její funkce je následovná:

- Najde příslušné pole (mezi lokálními, či globálními poli). V případě neúspěchu vyhodí výjimku.
- zapíše hodnotu do pole:

```
array.array[index.Evaluate()].value=expr.Evaluate();
```

5.3.8 MathExpr

Třída, která symbolizuje matematický výraz. Matematickým výrazem se rozumí takový výraz, který se skládá ze dvou výrazů spojených nějakým operátorem. Tento operátor je jedním z elementů spojového seznamu „mathSymbols“, který je definován ve třídě „Lists“:

```
public static string[] mathSymols =
 { "+", "-", "*", "/", "++", "--", "%", "<<", ">>" };
```

Třída *MathExpr* má tři atributy:

- **LeftExpression:** symbolizuje výraz na levé straně matematického operátoru.
- **Operator:** matematický operátor (string).
- **RightExpression:** symbolizuje výraz na pravé straně operátoru.

Metoda *Evaluate*() provede dle operátoru příslušný výpočet a vrátí výsledek tohoto výpočtu (int). Například, je-li atribut *operator* „+“:

```
case "+":
     return LeftExpr.Evaluate() + RightExpr.Evaluate();
```

5.3.9 BoolExpr

Stejně jako třída *MathExpr* má tři atributy:

- LeftExpr
- Operator
- RightExpr

Atribut *operator* však tentokrát smí býti pouze elementem, který je obsažen v poli „boolSymbols“ definovaném ve třídě *Lists*:

```
public static string[] boolSymbols =
{ "<", ">", "|", "||", "&", "&&", "==", "<=", ">=", "!="};
```

Metoda *Evaluate()* funguje podobně jako ve třídě *MathExpr*, s tím rozdílem, že se očekává jiný operátor. Například pro operátor „==“:

```
case "==":
     return LeftExpr.Evaluate() == RightExpr.Evaluate();
```

5.3.10 IfCondition

Třída, jejíž dva atributy jsou typu *Body* a symbolizují tělo, které buď přísluší situaci, kdy je podmínka splněna a nebo situaci, kdy podmínka splněna není:

- IfTrueBody
- ElseBody

Posledním atributem je:

- condition: typ *Expression*, symbolizuje onu podmínku.

Metoda *Evaluate()* podle podmínky zavolá metodu *IfTrueBody.Evaluate()* nebo (pokud *Elsebody* není null) *ElseBody.Evaluate()*.

5.3.11 WhileLoop

Obsahuje atribut „condition“ typu *Expression* a „body“ typu *Body*. Metoda *Evaluate()* vypadá následovně:

```
 public new void Evaluate()
{
while (((BoolExpr)Expr).Evaluate())
{
body.Execute();
}
}
```

5.3.12 ForLoop

Obdobně jako třída *WhileLoop* obsahuje atribut *body*. Dále však obsahuje další dva atributy:

- **Left:** atribut typu *Expression*. Reprezentuje počáteční přiřazení do proměnné.
- **Middle:** ekvivalent atributu *condition* ve třídě *WhileLoop*.
- **Right:** atribut typu *Expression*. Slouží ke změně hodnoty řídící proměnné.

Metoda *Evaluate()*:

```
   public new void Evaluate()
{
if(Left!=null)
Left.Evaluate();
while (((BoolExpr)Middle).Evaluate())
{
body.Execute();
Right.Evaluate();
}
}
```

5.3.13 Ret

Třída, která má pouze jediný atribut typu *Expression expr*. Pokud platí *expr==null*, pak metoda *Evaluate()* vrátí hodnotu *null*. V opačném případě vrátí hodnotu *expr.Evaluate()*.

5.3.14 FunctionCall

Reprezentuje výraz volání funkce. Má tyto atributy:

- **string FunctionName:** jméno funkce, která se má volat
- **Expression[] args:** argumenty, které se funkci mají předat.

Metoda *Evaluate()* zpracuje pole *args*, najde příslušnou funkci „f“ a zavolá na ni metodu *Execute()*. Té předává tři paramtery, které jsou ve třídě *FunctionCall* deklarovány následovně:

```
private Dictionary<string, Variable> valArgument;
private Dictionary<string, Array> arrayArg;
private Dictionary<string, BoundedInteger> boundedIntArg;
private List<Structure> structs = new List<Structure>();
```

Tedy hodnoty, pole, struktury a bounded integers se předávají separátně.
Ve voláních funkcí jsou podporávany i reference na proměnnou.

5.3.15 Val

Val je třída reprezentující hodnotu. Hodnotou se myslí hodnota proměnné nebo celé číslo „integer“. Například, máme-li výraz:

```
variable+4;
```

potom oba operandy (*variable* i 4) jsou reprezentovány touto třídou *Val*.

5.3.16 ArrayVal

ArrayVal je třída, jež reprezentuje hodnotu v poli. Jejími atributy tak jsou jméno pole a index (typ *Expression*).

Metoda *Evaluate*() vyhledá dané pole a vrátí hodnotu z příslušného indexu.

5.3.17 BoundedInteger

Jedná se o třídu reprezentující typ „Bounded integer“ definovaný gramatikou UPPAALu:

```
    public class BoundedInteger
{
public int lowerBound;
public int upperBound;
public int? value;
public string name;
}
```

5.3.18 Array

Třída pro reprezentaci polí. Její atributy reprezentují jméno, hodnotu a samotné pole. Reprezentace jednodimenzionálního pole délky 2, kde například *pole[]=2, pole[1]=3* by vypadala takto:

```
Array pole = new Array{name="pole"};
pole.array=new Array[2];
pole.array[0].value=2; pole.array[1].value=3;
```

5.3.19 Lists

Jedná se o třídu, ve které se nacházejí statické proměnné. Zejména pak reference na globální deklarace, aktuální lokální proměnné atp.

5.3.20 Structure

Structure je třída, která reprezentuje struktury. Podobně jako deklarace, struktury mohou obsahovat proměnné, pole a vázané proměnné. Mezi atributy této třídy tedy patří slovník reprezentující proměnné (kde klíčem je vždy jméno proměnné), dále spojový seznam pro vázané proměnné a spojový seznam pro pole. Navíc však třída Structure obsahuje atribut *name*, jenž určuje jméno proměnné, kterážto je „instancí“ této struktury. Předposledním atributem je *typeName* určující jméno struktury a posledním atributem je *types*. Jedná se o list „integerů“, které určují pořadí, v jakém jednotlivé proměnné, pole atp. byly nadeklarovány (důležité pro inicializaci).
Při deklaraci struktury je možné použít tzv. *initializer*:

```
struct{
int value;
int array[3];
}MyStruct = {2,{2,3,4}};
```

Hodnota *Mystruct.value* bude *2*, pole *MyStruct.array* bude obsahovat prvky *2,3,4* (a to přesně v tomto pořadí).
Pokud chceme v UPPAALu definovat nový typ, uděláme to pomocí konstrukce *typedef*. Syntaxe je stejná, jako v jazyce C. Například:

```
typedef struct{
int value;
int array[3];
}MyStruct;
MyStruct mystruct1 = {2,{2,3,4}};
MyStruct mystruct2  = {3,{7,3,4}};
```

Narozdíl od obyčejných struktur, při použití konstruktu *typedef* je možné vytvořit více proměnných jednoho nově nadeklarovaného datového typu.

5.3.21 TypeDef

Tato třída reprezentuje konstrukt *typedef* a obsahuje stejné atributy jako třída *Structure*. Navíc však obsahuje metodu *public Structure Instantiate(string name)*, která umožňuje vytvořit novou proměnnou, jejíž typ je symbolizován právě touto třídou $TypeDef$. Metoda *Instantiate()* vrátí instanci třídy *Structure*.

5.3.22 StructVal

Obdoba třídy Val, $StructVal$ však reprezentuje hodnotu proměnné ve struktuře. Oproti třídě Val se liší především jinou metodou *Evaluate()*, která se nejdříve snaží najít příslušnou strukturu (instanci třídy *Structure*) (za pomocí statické metody *Lists.FindStructure(string nameOfStructure)*), ve které najde příslušnou proměnnou. Navíc tato třída obsahuje atribut pro jméno struktury (což se týká také všech následujících tříd dědících od třídy *Expression*).

5.3.23 StructArrayVal

Podobně jako v předešlém případě, je tato třída inspirována třídou $ArrayVal$, opět se však liší metodou *Evaluate()*, jež vyhledá nejdříve danou strukturu (instanci třídy *Structure*) a následně pole v ní.

5.3.24 StructAssign

Účel této třídy je podobný jako u třídy *Assign*, rozdílem však je, že ve výrazu, jenž je instancí třídy *StructAssign*, se přiřazuje do proměnné ve struktuře (instanci třídy *Structure*). Oproti třídě *Assign* se tedy opět liší v odlišné implementaci metody *Evaluate()* a atributem *string nameOfStruct*.

5.3.25 StructArrayAssign

Jedná se o obdobu třídy *ArrayAssign*, v tomto případě se však dané pole nachází uvnitř dané struktury. Opět se oproti třídě *ArrayAssign* třída *StructArrayAssign* liší rozdílnou implementací metody *Evaluate()*.

5.4 Vytvoření objektové reprezentace programu napsaném v modelovacím jazyce

Nejprve se při průchodu derivačním stromem v metodách třídy *Uppaal_CListener* : *Iuppaal_CListener* sestavuje strom „objTree“ typu „ObjectTree“.

Třída *ObjTree* je naprogramována následovně:

```
 class ObjectTree
{
public Object obj;
public ObjType type;
public ArrayList next;
public bool visited = false;
public ObjectTree()
{
this.next = new ArrayList();
}
}
```

Každý uzel stromu typu *ObjectTree* má 0...*N* následovníků a dále má atribut třídy „Object“ (což je třída, od které dědí všechny třídy v jazyce C#). Každý uzel v tomto stromě reprezentuje nějaký element programu a atribut „obj“ je právě tímto elementem. Dalším atributem je *ObjType type*, což je výčtový typ, užitečný při přetypování atributu *obj* (určuje, na jaký typ se má přetypovat).
ObjType může nabývat těchto hodnot:

```
  public enum ObjType
{
Expression,
Body,
Function,
ForLoop,
WhileLoop,
IfCondition,
Declaration,
Return
}
```

Jak můžeme vidět, tento výčtový typ připomíná jednotlivé elementy pravidla *statement* v gramatice *Uppaal_C*.

Atribut *obj* třídy *ObjectTree* může být tedy instancí následujících tříd:

- Expression
- Body
- Function
- Expression:ForLoop
- Expression:WhileLoop
- Expression:Ifcondition
- Declaration
- Expression:Ret

Strom typu *ObjectTree* tak reprezentuje pravidlo *statement*.

5.4.1 Sestavení stromu objTree

Pokusím se celý proces demonstrovat na následujícím příkladu:

Mějme fragment programu:

```
while(i>1){
 i=i-1;
}
```

Tento fragment nejprve tokenizujeme Lexerem a následně „rozparsujeme“ novým *Parserem*. Nakonec začneme procházet jeho derivační strom. Při tomto průchodu se zavolá metoda třídy $Uppaal\_CListenerImpl$: $EnterWhileLoop$:

Tato metoda vytvoří ve stromu *objTree* nový uzel, jehož atribut *type* bude *ObjectType.WhileLoop* a atribut *obj* bude instancí třídy *WhileLoop*:

```
public void EnterWhileLoop(
    uppaal_CParser.WhileLoopContext context
        ){
    ReturnRelevantNonVisitedObjNode(objTree).next.Add(
    new ObjectTree {
            obj=new WhileLoopd{Declaration=declaration},
            type=ObjType.WhileLoop
         }
     );
}
```

Funkce *ReturnRelevantNonVisitedObjNode(ObjectTree objTree)* prohledává *objTree* do hloubky a vrátí nejhlubší uzel nejlevějšího podstromu, jehož atribut *visited* je *false*. (Nechť *objTree* je uzel a jeho atribut trees není *null* a má N prvků, kde $N > 2$. Pokud první element (tedy nejlevější element) *ArrayListu* trees má atribut *visited* na hodnotě *false*, pak se rekurzivně zavolá funkce *ReturnRelevantNonVisitedObjNode(child)* a vrátí se uzel, který vrátilo toto rekurzivní volání. V opačném případě se totéž zkouší (zleva) pro další následníky daného uzlu. Pokud pro žádný z potomků aktuálního uzlu neplatí, že atribut *visited* je *false*, pak funkce vrátí tento aktuální uzel).
Uzlu, který vrátí funkce *ReturnRelevantNonVisitedObjNode(ObjectTree objTree)* se přidá nový potomek, jehož atribut *obj* je nová instance třídy *WhileLoop*.

Následně se zavolají metody

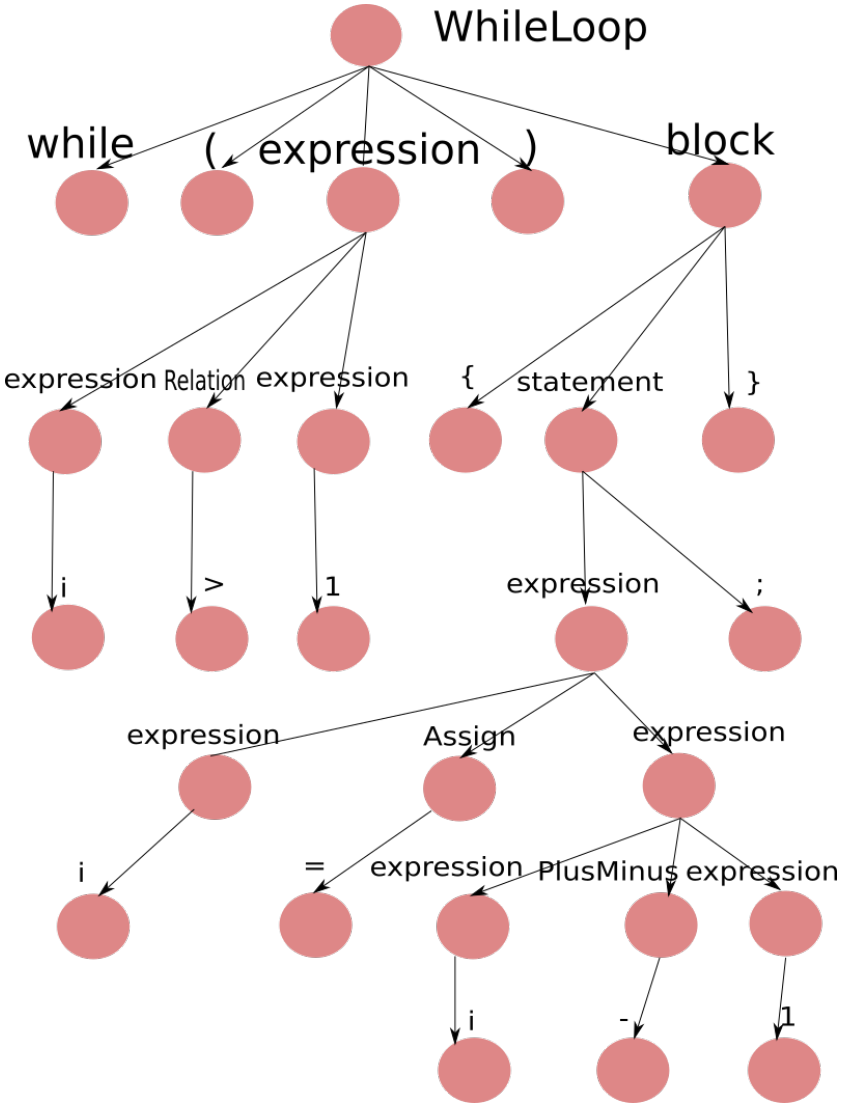
```
public void EnterExpression(uppaal_CParser.ExpressionContext context)
```

```
public void ExitExpression(uppaal_CParser.ExpressionContext context)
```

které vytvoří nový podstrom typu *ObjectTree* (jelikož je zpracování pravidla *expression* poměrně komplikované, používám k jeho zpracování speciální strom „ExpressionTree tree“, viz níže).

Argument *context* symbolizuje výraz $i > 1$. Jelikož „i“ a „1“ jsou také výrazy spadající pod pravidlo expression, zavolá se pro ně znovu dvojice metod *EnterExpression* a *ExitExpression* s argumentem, který odpovídá vždy příslušnému výrazu.
Tedy pro každé pravidlo z gramatiky se zavolá příslušná metoda *Enter∗*, projde se celý podstrom příslušného uzlu v derivačním stromu a následně se zavolá metoda *Exit∗*. Každá funkce *Exit** zpracuje případné potomky uzlu tak, že jejich atributy *obj* přiřadí atributům svého vlastního atributu *obj*.

Derivační podstrom, kde kořenem je uzel odpovídající pravidlu *WhileLoop*, pro náš příklad vypadá následovně:



Obrázek 5.1. Derivační strom pro while-cyklus z ilustračního příkladu.

5.4.2 Sestavení stromu ExpressionTree tree

Metoda *EnterExpression* vytvoří buď nový strom (pokud zatím neexistuje) a nebo vytvoří nový uzel. Tento uzel se stane potomkem uzlu, jenž vrátí metoda *ReturnRelevantNonVisitedNode(ExpressionTree tree)*, která má stejnou funkci, jako metoda *ReturnRe-*

levantNonVisitedObjNode(ObjectTree objTree) zmíněná v předchozí podsekci (jen pracuje s jiným typem stromů).
Třída *ExpressionTree* se skládá z následujících atributů:

- ExpressionType type
- VariableType ValType
- string name
- string primitiveElement
- LinkedList<ExpressionTree> trees

Atribut *type* určuje, jakého typu daný výraz je. Jedná-li se například o typ *ExpressionType.Assign*, bude se očekávat, že atribut *trees* má dvě položky:

- 1. položka bude ekvivalentem pro proměnnou, do které se má přiřazovat.
- 2. položka bude výraz, jehož hodnota se má do oné proměnné přiřadit.

Atribut *visited* má stejnou funkci jako v případě třídy *ObjectTree*. Atributy *name* a *primitiveElement* jsou užitečné v případě, kdy výrazem je hodnota nebo proměnná. V případě hodnoty bude ona hodnota uložena do atributu *primitiveElement*, v případě proměnné bude její jméno uloženo do atributu *name*.
Metoda *ExitExpression* přiřadí do atributů příslušného uzlu hodnoty (podle toho, o jaký typ výrazu jde). V případě, že je celý strom kompletní (tj. metoda *ReturnRelevantNonVisitedNode(ExpressionTree tree)* vrátí kořen stromu), zavolá se:

```
AddExpressionToObject(BuildExpressionFromTree(tree));
tree = null;
```

kde metoda *BuildExpressionFromTree(ExpressionTree tree)* vytvoří ze stromu *ExpressionTree* instanci některé třídy dědící od třídy Expression. (Například instanci třídy Expression:Assign).
Metoda *AddExpressionToObject(Expression expr)* přidá nový objekt do stromu *ObjectTree objTree*.

5.5 Urgent a Comitted locations

Tento speciální typ uzlů je popsán v kapitole 3.2. Přidání podpory pro urgent a committed locations mělo dvě části:

- **grafická podpora:** uzly typu urgent jsou označeny velkým písmenem „U“, zatímco uzly typu committed jsou označeny velkým písmenem „C“.
- **funkční podpora:** tj. pokud je některý z uzlů v aktuálním stavu urgent, zastaví se inkrementace času. Pokud je některý z uzlů v aktuálním stavu typu committed, pak další krok bude udělán právě z tohoto committed uzlu, mezitím se zastaví inkrementace času. (Poté, co již žádný uzel v aktuálním stavu není ani urgent ani committed, se inkrementace času obnoví).

Následující sekce popisují implementaci funkční podpory:

5.5.1 Urgent uzly

Pokud je některý z aktuálních uzlů typu urgent, nastaví se atribut *bool clock_running* na *false*, čímž se zastaví inkrementace času. Pokud žádný z aktuálních uzlů již není typu urgent (nebo committed), přenastaví se tento atribut zpět na hodnotu *true*.

Inkrementace času se děje především v metodě *Bw_DoWork(object sender, DoWorkEventArgs e)* třídy *RunForm*.
Ona metoda neustále volá metodu Step() dokud:

- 1. metoda Step() nevrátí hodnotu *false* (to se stane, pokud je některý invariant na aktuálním uzlu nesplněn).
- 2. stiskne se tlačítko stop.

Čas se v této metodě inkrementuje následovně:

```
if (!_runtime.stopClock)
{
while (Heartbeat.MonotoneTime < time + iteration * _interval)
Thread.Sleep(0);
}
```

Atribut *_runtime* je instance třídy *TARuntime*, proměnná *iteration* je definována v metodě *Bw_DoWork()* a je nastavena na hodnotu 1, při každém volání metody *Step()* se inkrementuje o jedničku. Atribut *_interval* je typu double a jeho hodnota je padesát tisícin. Proměnná *time* určije začátek běhu metody *Bw_DoWork()*.

5.5.2 Committed uzly

Pokud je alespoň jeden z uzlů v aktuálním stavu committed, pak stejně jako pro uzly typu urgent: dočasně se zastaví inkrementace času. Navíc se z tohoto committed uzlu metoda *Step()* pokusí dostat pryč. V případě, že to kvůli strážím na hranách nelze, pak vrátí metoda *Step()* hodnotu *false* a běh se ukončí.

Tato situace by však měla nastat pouze zřídka, neb metoda *Step()* kontroluje, zda cílový uzel na zvolené hraně je committed a jestli bude možné z tohoto committed uzlu udělat krok do nějakého dalšího uzlu.

Kapitola 6
Rozšíření algoritmu pro procházení modelů

6.1 Komunikace s programem EXAM

Pro komunikaci s EXAMem využívá Taster adaptér definovaný ve třídě *EXAMAdaptor*. Tato třída obsahuje mj. metodu *Execute()*, která pošle EXAMu string se jménem funkce (a jejími argumenty), kterou chceme, aby EXAM spustil.

6.1.1 Nástroj EXAM

Jedná se o program sloužící pro testování, zejména HIL [1] testování. Testovací funkce je možné navrhnout za použití grafického programování nebo je možné je naprogramovat přímo ve skriptovacím jazyce *Python*. Tyto testovací funkce může Taster zavolat díky metodě *Execute()* definované ve třídě *EXAMAdaptor*. Volání těchto testovacích funkcí není kompatibilní s nástrojem Uppaal, řešení tohoto problému popisuje následující podsekce:

6.1.2 Volání funkcí EXAMu

Nejprve bylo potřeba zajistit kompatibilitu s UPPAALem, neb ten žádné volání funkcí EXAMu nezná. Vyřešení kompatibility:

Chceme-li do modelu přidat volání funkce EXAMu, uděláme to následujícím konstruktem:

```
Execute(/*Something*/)
```

Například:

```
Execute(/*TrunkState.CheckPosition("Close"),
TrunkState.CheckPosition("Open") */)
```

Textový řetězec, který je uvnitř blokového komentáře bude argumentem *string expr* metody *Execute(string expr)* ve třídě TARuntime. Ona metoda její vstupní argument zpracuje (například vyhodnotí některé výrazy) a pošle metodě *Execute* v příslušném adaptéru.

Nadefinujeme-li v globálních deklaracích funkci „Execute“, pak se model stane kompatibilním s nástrojem UPPAAL.

6.1.3 Metoda Execute(string expr) třídy TARuntime

Tato metoda nejprve zpracuje svůj argument *expr* tak, že:

pokud je v řetezci *expr* volání „Execute(/*something*/)“, pak nejprve pošlou EXAMu argumenty tohoto vnořeného volání a místo „Execute(/*something*/)“ dosadí hodnotu, kterou EXAM vrátil. Pokud je v řetězci *expr* volání „interpret(something)“, pak se nejprve zavolá syntaktický analyzátor, který výraz „something“ převede na instanci třídy *Expression* a následně se tento výraz vyhodnotí. Metoda *Execute(string*

expr) pak nahradí část „interpret(something)“ hodnotou, na kterou se výraz „something“ vyhodnotil.

Po zpracování textového řetězce *expr* se tento řetězec pošle do EXAMu.

6.2 Náhodné časování událostí

V reálném světě se některé události dějí s časovým zpožděním, které je pokaždé jiné. Například: řidič chce nastartovat svůj automobil (poté, co uvedl do chodu elektroniku v onom automobilu, tj. klíč v zapalování je v první poloze), někdy to však udělá téměř hned od svého nástupu do automobilu a někdy až po nějaké době. Toto nedeterministické chování lze simulovat pomocí náhodného časování událostí.

Z tohoto důvodu jsem v Tasteru implementoval pro náhodné časování událostí podporu. Každou hranu v modelu je možné označit, aby případný krok přes tuto hranu byl učiněn až za nějaký náhodný čas. Maximální hodnota tohoto času je definována buď v globálních deklaracích (v proměnné $maxRandomTime$) nebo je definována explicitně pro tuto hranu.

Chceme-li, aby hrana byla použita s náhodným časovým zpožděním, přidáme na hranu do komentáře značku:

```
random()
```

V tomto případě se krok přes tuto hranu zpozdí o náhodný čas Δt, který patří do intervalu $\langle 0, maxRndomTime \rangle$, kde proměnná $maxRndomTime$ je definována v globálních deklaracích.

Alternativně je možné přidat na hranu do komentáře značku:

```
random(number)
```

kde *number* je nějaké celé číslo. Zpoždění Δt pak bude v intervalu $\langle 0, number \rangle$.

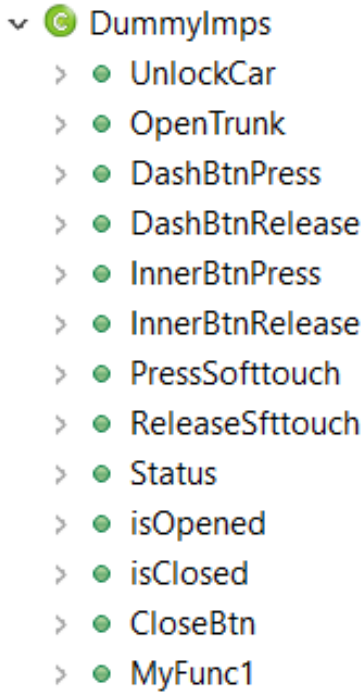
V případě, že některý z uzlů v aktuálním stavu je typu urgent nebo committed, pak se vybraná hrana vezme ihned bez ohledu na to, zda je označena značkou *random()* nebo *random(number)*. Důvod je ten, že pokud je jeden z uzlů typu urgent nebo committed, zastaví se běh času. Program by tak čekal na uběhnutí času Δt, který by ale nikdy neuběhl.

Kapitola 7
Testování

Cílem bylo otestovat novou funkcionalitu na modelech pokrývajících co nejvíce možných situací a opravit případné chyby.

7.1 Testování pomocí EXAMu

Od svého vedoucího jsem dostal šablonu s již předpřipravenými testovacími funkcemi:



Obrázek 7.1. Funkce definované v EXAMu (Funkci MyFunc1 jsem již definoval já - přijímá tři argumenty a vrací hodnotu typu int)

Následně jsem ve svém testovacím modelu vytvořil automat, který testuje volání funkcí EXAMu. Příkladem může být volání funkce (definované v testovací šabloně v EXAMu) *MyFunc1*:

```
Execute(/*Trunk.MyFunc1(Execute(Trunk.Status("up")),
          interpret((5+3)/4),
          "example")*/)
```

7.2 Testování interpretu

Zvolil jsem testování za pomocí svého vlastního testovacího rozhraní.

Rozšířil jsem gramatiku o možnost volání funkce *print*, přičemž syntaxe je následovná:

```
print(string -- variable);
```

Příkladem může být například:

```
print(result_should_be_2_result_is -- expr1);
```

Pokud Taster zachytí volání *print*, vypíše na standardní výstup řetězec „string“ a dále vypíše hodnotu zadané proměnné „variable“. Pro pole existuje podobná funkce *printAr*, která hodnoty v poli vypíše na standardní výstup pod sebe, každou na vlastní řádek.

Z důvodu přehlednosti platí, že se tyto testovací funkce mohou volat pouze na konci nějakého bloku kódu (jenž je v Tasteru reprezentován třídou Body). Všechny výrazy za nějakým voláním *print* nebo *printAr* jsou ignorovány (výjimku tvoří právě výrazy *print* a *printAr*).

Ani výraz *print* ani *printAr* nejsou kompatibilní s UPPAALem, proto není vhodné, aby byly neuspořádaně na různých místech v deklaracích.

V Tasteru se o volání těchto dvou testovacích funkcí stará třída *TestingPrint*, která dědí od třídy *Expression*.

7.2.1 TestingPrint:Expression

Jejími atributy jsou:

- string str: řetězec, jenž je argumentem funkce *print/printAr*
- string nameOfVar: jméno proměnné, jejíž hodnota se má vytisknout

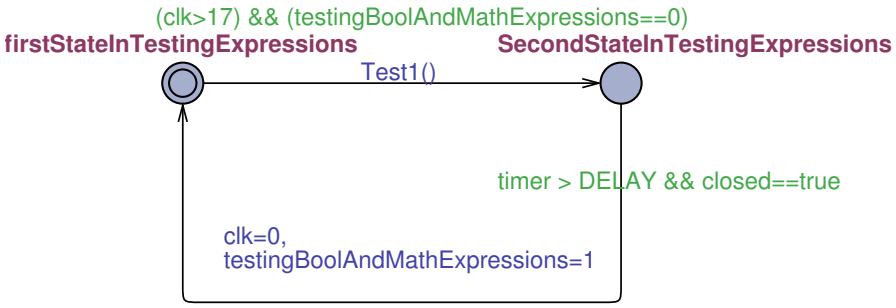
Metoda *public void Evaluate()* nalezne příslušnou proměnnou/pole (a to i uvnitř případné struktury) a spolu s řetězcem *str* vytiskne na standardní výstup její hodnotu.

7.2.2 Testování výrazů

Dalším automatem v mém testovacím modelu je jednoduchý automat, který má ve svých lokálních deklaracích definovanou metodu *Test1()*, ve které se nejprve přiřadí výrazy do proměnných a následně se volají testovací funkce *print/printAr*. Testují se následující výrazy:

```
expr1= 4+4+4+12/3-((1+1)*8)+n1*n2-number1;
expr2 = (bool1 && true) || (bool1+bi3*bi1)/(44*6);
expr3=(bool1 && true) && (bool1+bi3*bi1)/(44*6);
expr4 = bi3 && true && bool1 || (bool1 && true) || (bool1 || true);
expr5=bi1*(1+1+6*3+8/2+bool1*1000-50-50)+45/5+4*3+6;
expr6= (bi3 && true && bool1 || (bool1 && true) || (bool1 || true))
       == false;
}
```

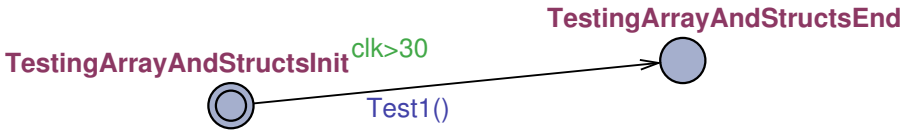
Testovací automat je ilustrován na následujícím obrázku:



Obrázek 7.2. Automat pro testování výrazů.

7.2.3 Testování polí, ukazatelů, struktur a funkcí

Poslední automat je velice jednoduchý, důležitá je však funkce $Test1()$, která se volá při průchodu tímto automatem:



Obrázek 7.3. Automat pro testování funkcí.

Funkce $Test1()$ je definována v lokálních deklaracích. Testuje použití struktur, polí, funkcí a vlastních datových typů, jež jsou opět definovány v globálních deklaracích (viz příloha C).

Tento test odhalil vícero chyb:

- Předávání struktur jako argumentů funkce - struktura nebyla přejmenována a funkce tudíž chtěla pracovat s jí neznámou strukturou
- Při volání funkce v nějaké jiné funkci se neuložily lokální proměnné té funkce, z níž se volalo. To vedlo k přepsání všech proměnných, přičemž původní proměnné se vytratily a ta funkce k nim dále neměla přístup.
- Selhalo předání pole jako argumentu funkce.
- Nedostatečné hledání struktur

Všechny odhalené chyby jsem opravil.

7.2.4 Testování uzlů typu urgent a committed

Pro toto testování jsem použil následující model:

Obrázek 7.4. Automat pro testování urgent a comitted locations. Uzly, které jsou označeny velkým písmenem „U“ jsou urgent, uzly označené velkým „C“ jsou committed

Tento test objevil chybu při práci se slovníkem - jednalo se o chybnou syntaxi. Po opravě této chyby již vše fugovalo.

Kapitola 8
Praktická ukázka

Cílem této části mé bakalářské práce bylo vytvořit model nějaké reálné situace pro demonstraci nové funkcionality Tasteru.

V kapitole 6 jsem se zaměřil čistě na testování nově implementované funkcionality Tasteru, zatímco tato sedmá kapitola je zaměřena na praktické využití výsledků mé bakalářské práce.

8.1 Model

V nástroji UPPAAL jsem vytvořil model, jenž má tři šablony, resp. automaty:

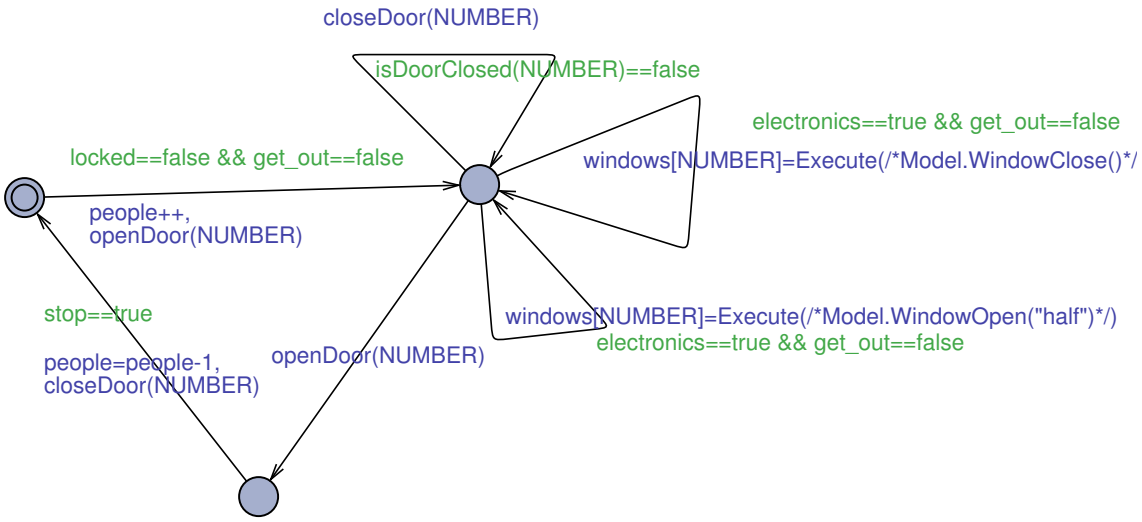
- Driver
- Car
- Passenger

Cílem modelu je zjednodušeně simulovat, jak řidič užívá automobil, přičemž součástí modelu mohou být i spolujezdci.

8.1.1 Passenger

Předpokládá se, že pasažérů bude 0...4, přičemž každý pasažér má své identifikační číslo, které se předává jako argument oné šablony při vytváření její instance.

Následující obrázek zobrazuje automat *Passenger*:

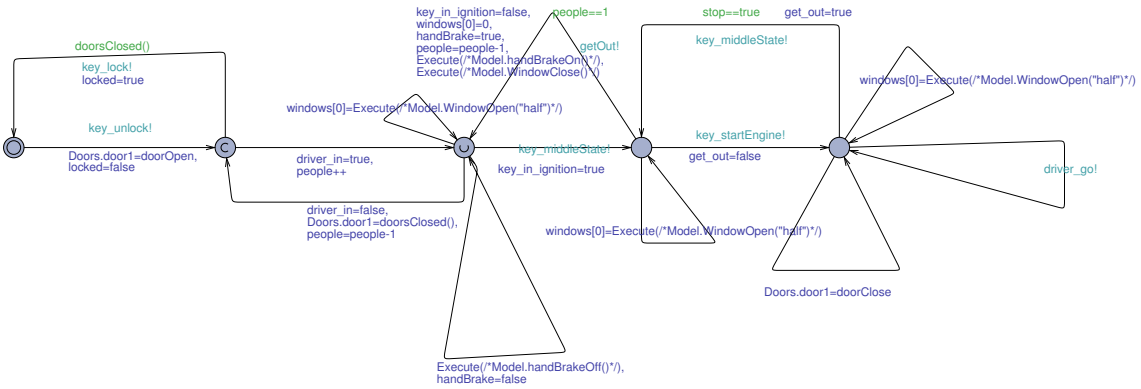


Obrázek 8.1. Automat pro testování funkcí.

Dle modelu: spolujezdec může vstoupit do odemčeného automobilu, manipulovat s okny a dveřmi (ty může otevřít jen tehdy, když automobil není v pohybu) a vystoupit z automobilu.

8.1.2 Driver

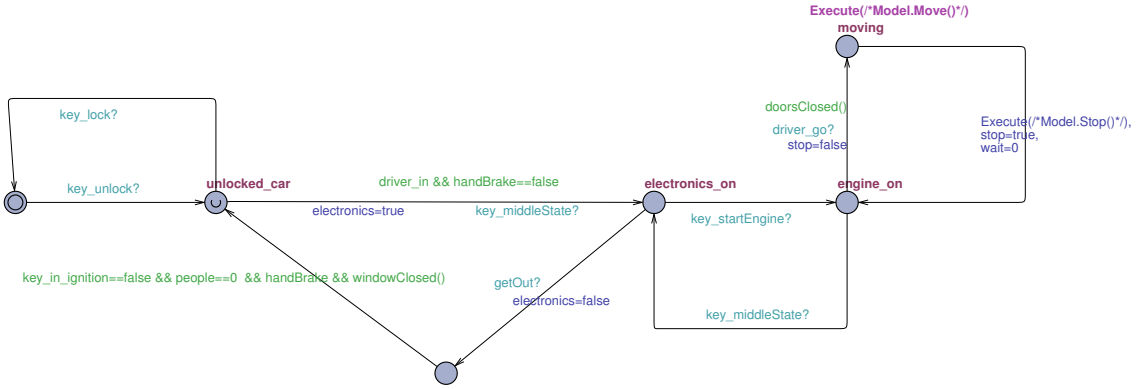
V modelu může řidič odemknout a zamknout automobil, dále může zapnout zapalování, nastartovat motor, uvést automobil do pohybu nebo (je-li automobil v pohybu) zastavit automobil. Konkrétně vše ilustruje následující automat, jenž je součástí celého modelu:



Obrázek 8.2. Automat pro testování funkcí.

8.1.3 Car

Stav automobilu v mém modelu vždy závisí na stavu řidiče. Tento automat tedy obsahuje synchronizované hrany s automatem *Driver*:



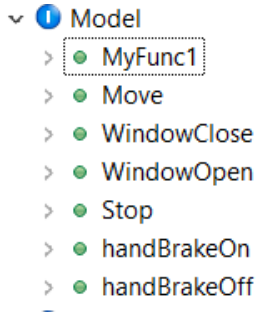
Obrázek 8.3. Automat pro testování funkcí.

8.1.4 Declarations

Každý jednotlivý automat v tomto ukázkovém modelu používá několik proměnných, funkcí, jedno pole a jednu strukturu. Vše je naprogramováno v globálních deklaracích, viz příloha D.

8.1.5 EXAM

V EXAMu jsem vytvořil rozhraní, ve kterém jsou nadeklarované funkce, jež se používají v této praktické ukázce. To ilustruje následující obrázek:



Obrázek 8.4. Automat pro testování funkcí.

Následně jsem vytvořil implementaci těchto funkcí v jazyce *Python*. Všechny tyto funkce jsou velice jednoduché, obvykle pouze vypíší na výstup informační řetězec. Cílem bylo demonstrovat funkčnost

8.1.6 Využívané funkcionality Tasteru

Model využívá následujících funkcionalit Tasteru, které byly implementovány v rámci této bakalářské práce:

- Volání funkcí EXAMu
- Interpret: struktury, pole, funkce, if-podmínky, for-cyklus, výrazy, proměnné atp.
- Urgent a Committed locations
- Náhodné časování událostí

8.1.7 Průchod modelem

Provedl jsem několik zkušebních průchodů modelem, při kterých se neprojevilo žádné neočekávané chování.

Kapitola 9
Závěr

Cílem této bakalářské práce bylo rozšířit nástroj Taster o náhodné časování událostí, nový interpret modelovacího jazyka a o podporu urgent a committed locations.

Náhodné časování událostí a podpora urgent a committed locations byla implementována především modifikací metody *Step()* třídy *TARuntime*, která má na starosti procházení modelem.

Pro nový interpret byl upraven stávající lexikální analyzátor (Lexer) a také byl výrazně rozšířen syntaktický analyzátor (Parser) včetně jeho gramatiky.

Následně jsem pro interpret vytvořil stromovou reprezentaci modelovacího jazyka, která se vytváří při průchodu derivačním stromem. Tuto stromovou reprezentaci lze taktéž chápat jako tzv. vnitřní formu.

Poslední částí mé bakalářské práce bylo implementovat zpracování argumentů pro volání funkcí EXAMu a vytvoření praktické ukázky, která demonstruje funkčnost vypracovaného řešení. Tato praktická ukázka byla demonstrována formou modelu vytvořeného prostřednictvím nástroje UPPAAL. Ukázkový model zjednodušeně reprezentuje, jak řidič a případní spolujezdci užívají automobil. Průchod tímto modelem v testovacím nástroji Taster proběhl dle mého očekávání.

Příloha A
Obsah přiloženého CD

- **Taster**: rozšířený nástroj Taster
- **Gramatika modelovacího jazyka**
- **Balíček EXAM**: obsahuje definici funkcí, jež volají testovací modely a model pro praktickou ukázku
- **Testovací modely**
- **Model reprezentující praktickou ukázku**

Příloha B
Pravidla lexikálního analyzátoru

```
Whitespace
:   [ \t]+
-> skip
;

Newline
:   (   '\r' '\n'?
|   '\n'
)
-> skip
;

BlockComment
:   '/*' .*? '*/'
-> skip
;



LineComment
:   '//' ~[\r\n]*
-> skip
;


Integer
: Digit+
;

Int : 'int';
Bool : 'bool';
Chan : 'chan';
Clock : 'clock';
Semi : ';';

fragment
Digit
: '0'..'9'
;

ID_NoDigitStart
: ( 'a'..'z' | 'A'..'Z' ) ('a'..'z' | 'A'..'Z' | '_' |Digit )*
;
```

```
Assign
: ('=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='| '|=' | '&='
| '^=' | '<<=' | '>>=');

Unary
: ('!' | 'not');

PlusMinus
: '+' | '-';

Binary
:( '*' | '/' | '%' |  '<?' | '>?' );

Shift
:('<<' | '>>' );

Testing
:('print' | 'printAr');

Relation
:( '<' | '<=' | '>=' | '>');

Ekvivalence
:('==' | '!=');

LogicalH
: ('&&' | '||' | '&' | '|' | '^' );
```

Příloha C
Testování funkcí, struktur a polí

V této příloze je ukázka testovacího programu pro interpret nástroje Taster. Testují se zejména funkce složitějšího charakteru, vlastní typy, struktury a pole.

```
clock clk;

int fibN=6;

int fibonacci(int &n){
int pole[100];
int i=2;
pole[0]=0;
pole[1]=1;
for(i;i<n;i++){
pole[i]=pole[i-2]+pole[i-1];
}
if(n>0){

n= pole[n-1];
}
return fibN;
}

int arrayToSort[8];

void fillArray(){
arrayToSort[0]=3;
arrayToSort[1]=6;
arrayToSort[2]=8;
arrayToSort[3]=0;
arrayToSort[4]=4;
arrayToSort[5]=2;
arrayToSort[6]=1;
arrayToSort[7]=7;
}


void bubleSort(int &array[8]){
int tmp;
int size=8;
int i=0;
int j=0;
for( i = 0; i < size - 1; i++){
for(j = 0; j < size - i - 1; j++){
if(array[j+1] < array[j]){
tmp = array[j + 1];
```

```
array[j + 1] = array[j];
array[j] = tmp;
}
}
}
}



typedef struct{
int value;
bool b;
int[4,8] bi;
int ar[10];
}TD;

TD t1;
TD t2={1,false,5,{0,1,2,3,98,5,6,7,8,9}};

struct{
int value;
int pole[2];
//int bi;
}ss;

typedef struct{
int a;
bool b;
}S;

S str2={2,1};
S str3;
S str4={5,6};



typedef struct{
int val;
int ar[8];
int vv;
}TDD;

TDD tt1={3,{1,2,19,4,5,17,14,11},4};
TDD tt2={1,{0,0,0,0,0,0,0,0},3};

void add(TDD t1, TDD t2){
t1.val=t1.val+t2.val;
t1.vv=t1.vv+t2.vv;
}


void Test1(){
int res;
```

```
fillArray();
bubleSort(arrayToSort);

bubleSort(tt1.ar);
res=fibonacci(fibN);
add(tt1,tt2);
printAr(TestingArrayAndStructs_array_should_be_sorted -- arrayToSort);
printAr(TestingArrayAndStructs_array_should_be_sorted -- tt1.ar);
print(TestingArrayAndStructs_result_should_be_5_is -- res);
print(TestingArrayAndStructs_first_should_be_4_is -- tt1.val);
print(TestingArrayAndStructs_second_should_be_7_is -- tt1.vv);

}
```

Příloha D
Globální deklarace pro ukázkový model

```
int NUMBER_OF_PERSONS=3;

clock wait;

chan key_lock;
chan key_unlock;
chan key_middleState;
chan key_startEngine;
chan driver_go;
chan getOut;

bool driver_in=false;
bool key_in_ignition=false;
int people=0;
bool handBrake=true;
bool stop=true;
bool locked=true;
bool electronics=false;
bool get_out=false;

int windowClose=0;
int windowOpenFull=1;
int windowOpen=2;

int windows[4];

struct{
int door1;
int door2;
int door3;
int door4;
int trunk;
}Doors={0,0,0,0,0};
int doorOpen=1;
int doorClose=0;

void initWindows(){
windows[0]=0;
windows[1]=0;
windows[2]=0;
windows[3]=0;
}

int Execute(){
```

```
return 1;
}
bool windowClosed(){
int i;
for(i=0; i<4; i++){
if(windows[i]!=windowClose){
return false;
}
}
return true;
}

bool doorsClosed(){

bool ret;
ret=Doors.door1==0 && Doors.door2==0 && Doors.door3==0 && Doors.door4==0;
return ret;
}

bool isDoorClosed(int number){
if(number==2){
return Doors.door2;}
if(number==3){
return Doors.door3;}
if(number==4){
return Doors.door4;}
return false;
}

void openDoor(int number){
if(number==2){
Doors.door2=1;}
if(number==3){
Doors.door3=1;}
if(number==4){
Doors.door4=1;}
}

void closeDoor(int number){
if(number==2){
Doors.door2=0;}
if(number==3){
Doors.door3=0;}
if(number==4){
Doors.door4=0;}
}

//
clock GLOBAL_CLK;
```

Literatura

[1] Hans-Petter Halvorsen. *Introduction to Hardware-in-the-Loop Simulation*. http://home.hit.no/~hansha/documents/lab/Lab%20Work/HIL%20Simulation/Background/Introduction%20to%20HIL%20Simulation.pdf.

[2] Ing. Jaromír Krecl. *HIL simulace jako prostředek pro testování řídících jednotek v automobilu*. http://dsp.vscht.cz/konference_matlab/matlab03/krecl.pdf.

[3] Rajeev Alur. *Timed automata, In: Computer Aided Verification*. 1999.

[4] David L.Dill Rajeev Alur. *A Theory of Timed Automata*. 1994. https://www.sciencedirect.com/science/article/pii/0304397594900108.

[5] *ANTLR 4 Documentation*. https://github.com/antlr/antlr4/blob/master/doc/index.md.

[6] David Alexandre, Amnell Tobias, Stigge Martin a Ekberg Pontus. *Uppaal 4.0 : Small Tutorial*. 2009. https://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf.

[7] Alastair Donaldson Alice Miller. *Symetry reduction methods for model checking*. https://www.doc.ic.ac.uk/crg/events/ARW07/submissions/Miller.pdf.

[8] Jan Janousek. *Programovací jazyky a překladače - lexikalní analýza*. 2011. https://edux.fit.cvut.cz/oppa/BI-PJP/prednasky/pjp2.pdf.

[9] Marie Demlova. *Jazyky, automaty a gramatiky*. 2018. http://math.feld.cvut.cz/demlova/teaching/jag/jag7dohromady.pdf.

[10] *UPPAAL Web Help*. 2012. http://www.it.uu.se/research/group/darts/uppaal/help.php.