VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

HLEDÁNÍ ŘÍDICÍCH STRATEGIÍ POMOCÍ UPPAAL STRATEGO

CONTROL STRATEGY SEARCHING USING UPPAAL STRATEGO

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **FILIP HRUŠKA**
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 bakalářské práce

Student: **Hruška Filip**
Program: Informační technologie
Název: **Hledání řídicích strategií pomocí UPPAAL STRATEGO**
Search of Control Strategies Using UPPAAL STRATEGO
Kategorie: Formální verifikace

Zadání:

1. Seznamte se s principy modelování a analýzy systémů pomocí prostředků rodiny nástrojů UPPAAL; zvláštní pozornost věnujte nástroji UPPAAL STRATEGO.
2. Zdokumentujte možnosti nástroje UPPAAL STRATEGO, zejména se zaměřte na klíčové pojmy, prostředky a principy související s tvorbou a analýzou řídicích strategií.
3. Po dohodě s vedoucím zvolte několik problémů z různých oblastí (cestování, doručování zásilek, řízení vozidel, hraní her atp.) nabízejících dostatečnou volnost z hlediska volby strategií řízení s ohledem na daná kritéria (např. čas pro řešení, počet úkonů, vzdálenost či energie).
4. Zvolené problémy popište prostředky nástroje UPPAAL STRATEGO způsobem umožňujícím následné hledání a zhodnocení řídicích strategií pro řešení těchto problémů.
5. Ověřte schopnost nástroje UPPAAL STRATEGO najít řídicí strategie s ohledem na různá kritéria, nalezené strategie zhodnoťte.
6. Kriticky zhodnoťte nástroj UPPAAL STRATEGO z hlediska hledání strategií a práce s ním, nastiňte další problémy vhodné k řešení tímto nástrojem.

Literatura:

- David A., Jensen P.G., Larsen K.G., Mikučionis M., Taankvist J. H.: UPPAAL STRATEGO. In: Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS vol. 9035, Springer, Berlin, 2015, pp. 206-211. DOI: 10.1007/978-3-662-46681-0_16.
- Mikučionis, M.: UPPAAL STRATEGO - Strategy Synthesis, Learning, Optimization and Evaluation, 2015. [on-line, cit. 19. 6. 2019]. Dostupné z http://people.cs.aau.dk/~marius/stratego

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání, nastínění způsobu modelování a analýzy strategií pro alespoň jednu oblast či problém.

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: 28. května 2020
Datum schválení: 25. října 2019

Abstrakt

Tato práce se zabývá hledáním řídicích strategií v předem vybraných problémech z různých oblastí pomocí nástroje Uppaal Stratego. Byly vybrány čtyři oblasti k řešení, jmenovitě šachy, hanojské věže, posuvné pole a kinematický problém zahrnující balíček, auto a letadlo. Pro zvolené problémy a oblasti byla navržena a vytvořena sada jim odpovídajícím modelů. U hanojských věží a posuvného pole bylo možné úspěšně vyhodnotit relevantní strategie, zvedající pravděpodobnosti úspěchu až na více než 90 %. U dalších modelů byl nalezen problém ve velikosti stavového prostoru a strategie nebylo možné vyhodnotit, protože maximální kapacita paměti, kterou nástroj využívá, nebyla dostatečná. U kinematického problému se po omezení a zjednodušení modelu podařilo strategie vyhodnotit, ovšem u šachů to nebylo možné ani po významném zjednodušení.

Abstract

This thesis deals with finding control strategies for pre-selected problems from various areas using tool Uppaal Stratego. Four areas were selected, namely chess, a sliding puzzle, the tower of Hanoi, and a kinematic problem involving a package, a car, and an airplane. For the selected areas and problems, a set of models was designed and implemented. For the tower of Hanoi and the sliding field, it was possible to successfully evaluate relevant strategies, raising the probabilities of success to more than 90 %. For other models, a problem was found in the size of the state space and the strategies could not be evaluated because the maximum memory capacity that the tool uses was not sufficient. For the kinematic problem, after limiting and simplifying the model, the strategies were successfully evaluated, but for chess, this was not possible even after significant simplification.

Klíčová slova

formální verifikace, ověřování modelů, časované automaty, modální logika, dotazovací jazyk, Uppaal, Uppaal SMC, Uppaal Stratego, řídicí strategie

Keywords

formal verification, model checking, timed automata, modal logic, query language, Uppaal, Uppaal SMC, Uppaal Stratego, control strategy

Citace

HRUŠKA, Filip. *Hledání řídicích strategií pomocí Uppaal Stratego*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Hledání řídicích strategií pomocí Uppaal Stratego

Prohlášení

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

.....
Filip Hruška
27. května 2020

Poděkování

Zde bych chtěl poděkovat svému vedoucímu Ing. Josefu Strnadelovi, Ph.D. za cenné rady, ochotu, časté konzultace a stálou podporu během tvorby bakalářské práce.

Obsah

Kapitola 1

Úvod

Tato práce se zaměřuje na softwarový nástroj Uppaal Stratego, který nám slouží k validaci, verifikaci a modelování systémů, ve kterých existuje nějaká řídicí strategie (např. hraní her, doručování zásilek, cestování apod.). Dovoluje nám provádět nad modely různé simulace, které můžeme řídit či nechat běžet náhodně, a můžeme tyto simulace také vizualizovat pomocí grafů. Je možné i kontrolovat různé vlastnosti modelu (např. zda ve všech stavech platí, že ujetá vzdálenost je menší než nějaká hodnota).

Práce má potenciál ukázat výhody tohoto nástroje a případně i rozšířit jeho užití, což by bylo důležité nejen pro samotné tvůrce tohoto nástroje, ale i pro ostatní, kteří se v této oblasti pracují a pro tento nástroj mají využití. V minulosti se nátroj postupně vyvíjel (původní nástroj Uppaal) a na jeho základě byly vytvořeny další verze (Stratego, SMC, TIGA atd.), které vždy buď přidaly další funkcionality nebo se zaměřily na speciální oblasti.

Jsem rád, že se této práci mohu věnovat, protože mě tato oblast zajímá a pomocí tohoto tématu do ní můžu trochu nekonvenčně proniknout a pořádně si vyzkoušet, co to všechno obnáší a jaké z toho mohou být výsledky. Líbí se mi i možnost si vybrat pro modelování i simulaci jakýkoliv problém, abych si mohl vybrat skutečně něco, co mě zajímá a případně něco, kde by výsledky mohly pomoci.

Cílem práce je především ukázat schopnosti a možnosti nástroje Uppaal Stratego, naučit se s ním pracovat a vyřešit pomocí něj předem zvolené problémy a vytvořit v nich použitelné a rozumné strategie. Výsledky a výsledné strategie bych rád porovnal u jednodušších problémů s výsledky mého vlastního řešení problémů. Tím bych mohl vidět, jak moc efektivní strategie mohou být a zda odhalí něco navíc.

V následující kapitole se budeme zabývat všemi důležitými teoretickými prvky. Řekneme si něco o formální verifikaci, metodě model checking a časovaných automatech. Dále je zde popsáná základní verze nástroje Uppaal (jak jsou v něm řešené časované automaty, jeho vlastnosti, principy, klíčové pojmy, dotazovací jazyk apod.) a rozšířené verze Uppaal SMC a Uppaal Stratego, které z něj výchází. Nakonec si v této kapitole ukážeme příklad použití nástroje Uppaal Stratego. V kapitole 3 budu mluvit o problémech, které byly vybrány k vyřešení a budu popisovat návrhy, jak tyto problémy řešit. V kapitole 4 bude popsána konkrétní implementace modelů těchto problémů a v kapitole 5 budou popsány strategie, nalezené výsledky a případně problémy, které při práci se strategiemi vznikly u daného modelu.

Kapitola 2

Teoretická část

První se budeme zabývat pojmem **verifikace**. U systému (či jejich modelů) je třeba důkladně kontrolovat jejich správnost dle daných vlastností či specifikací. Pokud tedy máme nějaké požadavky, je třeba ověřit, zda systém splňuje tyto požadavky nebo najít nějaké příklady ukazující nekorektní chování tohoto systému. Verifikovat můžeme různými způsoby: testování, simulace či formální verifikace, která nás zajímá nejvíce. Formální verifikace je velmi náročná (výpočetně i časově), proto se nehodí u obyčejných softwarových aplikací, ovšem můžeme díky ní velmi spolehlivě ukázat správnost systému. Formální verifikaci můžeme rozdělit na dva základní přístupy, a to **deduktivní** a **automatickou**. V deduktivních metodách se soustředíme hlavně na matematické důkazy, pomocí kterých potvrdíme správnost systému dle požadavků. Tento proces můžeme dělat buď ručně nebo jej z části automatizovat, ovšem není vůbec jednoduchý a také je velmi časově náročný. Automatické metody jsou naopak plně automatizované a spoléhají se na hrubou sílu, pomocí které prohledávají všechny možné chování systému ve snaze dokázat, že splňuje požadavky ve všech stavech. Metody jsou schopné ukázat i protipříklad, aby ukázaly špatné chování. [15]

Z automatických metod formální verifikace je pro nás nejdůležitější metoda model checking. Ta systematicky prohledává stavový prostor systému a na základě toho určuje jeho správnost podle dané správnostní specifikace. Uživatel musí poskytnout model tohoto systému a určitou logickou formuli popisující vlastnosti. Algoritmus poté zkontroluje, zda model splňuje dodané formule a může také poskytnout protipříklad v případě, že je model nesplňuje. Stavový prostor je možné znázornit pomocí grafu, kde jednotlivé uzly jsou stavy systému a hrany jsou přechody mezi těmito stavy. Graf by měl ukazovat a popisovat všechny možné chování modelu.[14]

Tato metoda může narazit v systémech s konečným počtem stavů na problémy. Jeden z nich se nazývá `výbuch stavového prostoru`. V systému máme nějaký počet dosažitelných stavů a ty exponenciálně rostou. K vyřešení tohoto problému existuje mnoho heuristik. Pro tuto metodu, i když byla navržena pro systémy s konečným počtem stavů, byly navrženy způsoby na řešení systémů s nekonečným počtem stavů. Zde ovšem máme ten problém, že jednoduše nelze znát a vyjmenovat všechny dosažitelné stavy systému. Problém může ještě samozřejmě vzniknout při tvorbě samotného modelu systému, což může být velmi náročné detekovat a může ignorovat chování, které je možné ve skutečném systému, což vede k chybám. Naopak model také může povolit chování, které v modelovaném systému možné není, tudíž bychom mohli dostávat falešné a nerelevantní chyby. [10]

2.1 Časované automaty, dotazovací jazyk a nástroj Uppaal

Pro popis modelu či systému k verifikaci existuje mnoho různých jazyků a způsobů, ale my se budeme zabývat nástrojem Uppaal. Uppaal je softwarový nástroj, který můžeme používat pro vytváření modelů pomocí časovaných automatů (jejich rozšířená verze) a kde můžeme ověřovat vlastnosti těchto modelů pomocí prohledávání stavového prostoru. Uppaal nám poskytuje tři hlavní složky: editor pro tvorbu šablon (časované automaty, sítě časovaných automatů), kde můžeme používat jazyk syntakticky podobný C, poté simulační prostředí, kde můžeme spouštět různé simulace (můžeme je ovládat nebo je nechat běžet náhodně), ve kterých lze sledovat různé cesty, a nakonec prostředí pro model checking, kde kontrolujeme, zda model splňuje dané formule.[3]

Nástroj byl vytvořen ve výzkumném centru BRICS[1] na univerzitě Aalborg v Dánsku a na univerzitě Uppsala v oddělení informační technologie[2] ve Švédsku. [16]

V následujících sekcích si první povíme něco o časovaných automatech, poté dotazovacím jazyku, který se v Uppaalu používá, a o základních vlastnostech tohoto nástroje (jako např. invarianty, stráže, šablony apod.). Dále se podíváme na rozšířené verze Uppaalu (specificky na Uppaal SMC a Uppaal Stratego) a nakonec si ukážeme příklad tempomatu vytvořený v Uppaalu.

Časované automaty

Na časované automaty se můžeme dívat jako na konečné automaty, které jsou rozšířené o proměnné, jež budeme nazývat hodiny. Hodiny mohou nabývat reálných hodnot. Časovaný automat nám může sloužit jako abstraktní model libovolného časovaného systému. Hodiny se při inicializaci nastaví na nulu a poté se synchronně zvyšují. Na hranách jsou omezení neboli stráže, které různými způsoby omezují chování automatu. Přejít z jednoho stavu do dalšího pomocí hrany můžeme pouze tehdy, pokud hodnoty proměnných hodin splňují podmínku stráže na dané hraně (pokud tam nějaká je). Pokud se přechod provede, hodiny mohou být resetovány na nulu.[2]

Všechny následující definice a jejich notace jsou napsány podle [4]. Následuje definice časovaných automatů. Použité notace: X je konečná množina hodin, které mají hodnoty $\mathbb{R}_{\geq 0}$. Ohodnocení hodin v nad X je funkce $v : X \to \mathbb{R}_{\geq 0}$, což přiřazuje všem hodinám x hodnotu $v(x) \in \mathbb{R}_{\geq 0}$. Pokud máme $d \in \mathbb{R}_{\geq 0}$, tak píšeme $v + d$ pro ohodnocení hodin, které přiřazují hodinám x hodnotu $v(x + d)$. Pokud r je podmnožina X, tak $[r \leftarrow 0]$ ohodnocení v' takové, že $v'(x) = 0$, pokud $x \in r$, jinak $v'(x) = v(x)$.

Množinu hodinových omezení nad X zapisujeme jako $C(X)$, jinak řečeno množina boolovských kombinací s atomickými omezeními tvaru $x \bowtie c$, kde $x \in X$, $\bowtie \in \{=, <, \leq, >, \geq\}$ a $c \in \mathbb{N}$. Píšeme $C_<(X)$, což omezuje $C(X)$ pouze na pozitivní boolovské kombinace obsahující pouze omezení tvaru $x \leq c$ nebo $x < c$. Ohodnocení v splňuje atomické omezení $x \bowtie c$ kdykoliv, kdy $v(x) \bowtie c$. Pokud ohodnocení v splňuje omezení g, píšeme $v \models g$.

Definice 1 (Časovaný automat) *Časovaný automat je šestice* $(L, l_0, X, Inv, T, \Sigma)$, *kde:*

- L *je konečná množina stavů, také známy jako místa,*
- $l_0 \in L$ *je počáteční místo,*
- X *je konečná množina hodin,*

[1] angl. Basic Research in Computer Science, https://www.brics.dk/
[2] http://www.it.uu.se/

- $T \subseteq L \times C(X) \times \Sigma \times 2^X \times L$ *je konečná množina přechodů:* $e = \langle l, g, a, r, l' \rangle \in T$ *označuje přechod z* l *do* l', g *je stráž* e, r *je množina hodin, která je resetována* e, a *je akce* e. e *také píšeme jako* $l \xrightarrow{g,a,r} l'$,
- $Inv : L \to C_<(X)$ *přiřazuje invarianty místům,*
- Σ *je abeceda akcí*

Stav či konfigurace časovaného automatu je pár $(l, v) \in L \times \mathbb{R}^X$, kde l je současné místo a v je ohodnocení hodin. Následuje definice časovaného přechodového systému a sémantiky časovaného automatu, kterou můžeme definovat jako časovaný přechodový systém s přechody akcí a přechody zpoždění.

Definice 2 (Časovaný přechodový systém (TTS)) *TTS je čtveřice* $(S, s_0, \to, \Sigma)$ *kde* S *je množina stavů,* $s_0 \in S$ *je počáteční stav a* $\to \subseteq S \times (\Sigma \cup \mathbb{R}) \times S$ *je přechodová relace. Relace* $\to$ *splňuje následující tři podmínky:*

- *pokud* $s \xrightarrow{0} s'$, *pak* $s = s'$,
- *pokud* $s \xrightarrow{d} s'$ *a* $s' \xrightarrow{d'} s''$, *kde* $d, d' \in \mathbb{R}$, *tak* $s \xrightarrow{d+d'} s''$,
- *pokud* $s \xrightarrow{d} s'$, *kde* $d \in \mathbb{R}$, *tak pro všechny* $0 \leqslant d' \leqslant d'$ *existuje* $s'' \in S$ *takové, že* $s \xrightarrow{d'} s''$ *a* $s'' \xrightarrow{d-d'} s'$

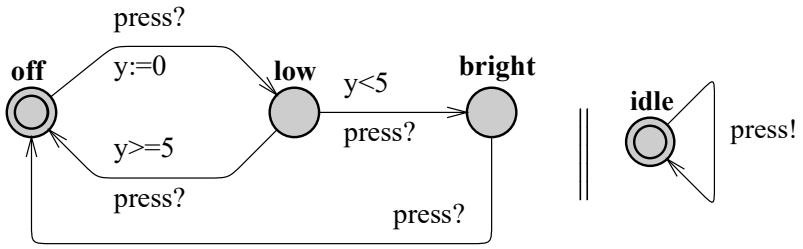
Pro $s \in S$ platí, že je dosažitelné v S, pokud existuje provedení z s_0 do S.

Definice 3 (Sémantika časovaného automatu) *Nechť* $(L, l_0, X, Inv, T, \Sigma)$ *je časovaný automat. Sémantika je definovaná jako časovaný přechodový systém* $(S, s_0, \to, \Sigma)$, *kde:*

- $S = L \times \mathbb{R}^X$
- $s_0 = (l_0, v_0)$ *kde* $v_0(x) = 0$ *pro každé* $x \in X$,
- *přechodový vztah* $\to$ *je složen z:*
  - *přechody akcí:* $(l, v) \xrightarrow{a} (l', v')$ *pouze v případě, že existuje* $l \xrightarrow{g,a,r} l' \in T$ *tak, že* $v \models g$, $v' = [r \leftarrow 0]v$ *a* $v' \models Inv(l')$.
  - *přechody zpoždění: pokud* $d \in \mathbb{R}$, $(l, v) \xrightarrow{d} (l, v + d)$ *pouze v případě, že* $v + d \models Inv(l')$.

Systém začíná v počáteční konfiguraci, kde se nachází v místu l_0 a všechny hodiny jsou vynulovány, a poté může udělat okamžitý přechod (který může vynulovat hodiny), pokud ohodnocení hodin splňuje podmínky stráží (přechody akcí) nebo bude jen zvyšovat hodnotu všech hodin o stejný čas (hodiny jsou synchronní), pokud splňují podmínky invariant.[4]

Nástroj UPAAL nám nabízí možnost modelovat pomocí většího množství časovaných automatů, které běží paralelně. Tento model můžeme rozšířit i o proměnné, které jsou omezené a diskrétní. Tyto proměnné lze číst, lze do nich zapisovat, lze s nimi provádět matematické operace a také jsou součástí každého stavu. Stavem je myšlen stav systému, který je definován místy všech automatů, hodinovými omezeními a hodnotami diskrétních proměnných. Každý z paralelně běžících časovaných automatů může provést své přechody



(a) Lamp. (b) User.

Obrázek 2.1: Příklad jednoduché lampy, kde část (a) reprezentuje lampu a část (b) uživatele. Lampa může být vypnutá, zapnuta se slabým světlem nebo s jasným světlem. Uživatel může kdykoliv lampu zapnout. Převzato z [1].

(na hranách) odděleně od ostatních automatů nebo se s jiným automatem synchronizovat, což vede do nového stavu. [1]

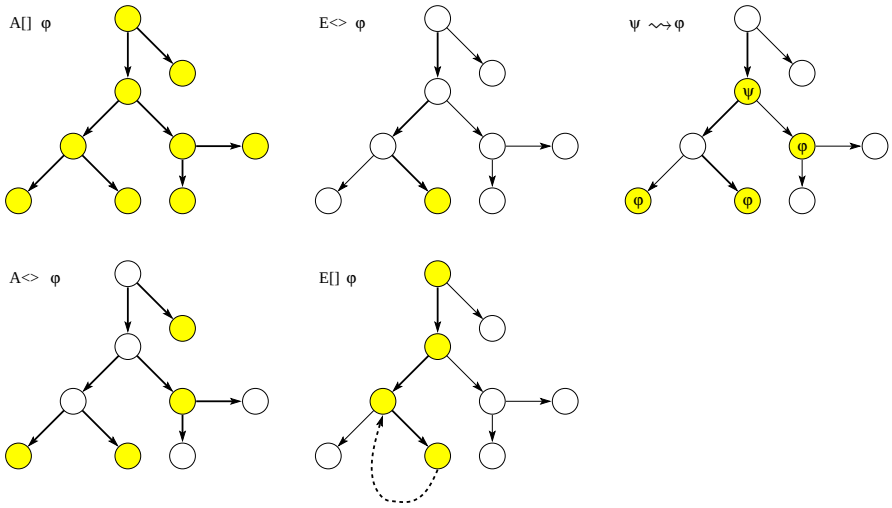
V obrázku 2.1 máme příklad časovaného automatu, který modeluje jednoduchou lampu a model uživatele. Nachází se zde proměnná y, která reprezentuje hodiny, jež kontrolují rychlost klikání tlačítka pro ovládání lampy uživatelem. Uživatel může kliknout na tlačítko náhodně v jakýkoliv čas nebo ho nezmáčknout vůbec. Lampa má tři stavy: `off` (vypnutá), `low` (slabé světlo) a `bright` (jasné světlo). Jakmile uživatel klikne na tlačítko, tak vyšle signál `press!`, kterým se synchronizuje s `press?` u lampy a tudíž se lampa slabě rozsvítí a čas je vynulován. Pokud poté uživatel klikne na tlačítko dostatečně rychle ($y < 5$), tak se přesune do stavu jasného světla, jinak se lampa vypne ($y \geq 5$). V jasném světle se lampa po dalším kliknutí vypne.[1]

Dotazovací jazyk

Mnohdy potřebujeme v logice více možností než pravdivá formule či nepravdivá formule (jak tomu je u predikátové nebo výrokové logiky). Pravda může mít mnoho podob, například něco může být pravdivé až v budoucnu, musí to být nutně pravdivé, je to známo jako pravdivé nebo se věří, že to je pravdivé. K tomuto nám slouží modální logika. Aby vyjádřila tyto různé verze pravdy, tak používá unární operátory (`[]`, což znamená, že něco musí platit nutně a $\Diamond$, což značí, že něco možná platí). Speciálním případem modální logiky je temporální logika. Zde se zmíním o lineární temporální logice[3], která modeluje čas jako sekvenci stavů, které se nekonečně prodlužují do budoucnosti (této sekvenci se také říká komputační cesta nebo zjednodušeně cesta). Budoucnost ovšem není rozhodnutá, tudíž se díváme na vícero cest (více verzí budoucnosti), z nichž jakákoliv může být realizována. Nelze zde ovšem vyjádřit vlastnosti, které potvrzují existenci cest. K tomu nám pomůže komputační stromová logika[4]. CTL modeluje čas jako stromovou strukturu, kde budoucnost není rozhodnuta a je v ní více cest, ze kterých může být jakákoliv použita. CTL používá operátory z LTL, a to U (dokud), F (nějaký budoucí stav), G (všechny budoucí stavy) a X (další stav). Také máme kvantifikátory A (všechny cesty) a E (existuje alespoň nějaká cesta).[9]

[3] angl. Linear Temporal Logic, LTL
[4] angl. Computation Tree Logic, CTL

Zbytek této podsekce byl napsán podle [1]. Uppaal používá zjednodušenou verzi časované komputační stromové logiky[5]. Dotazovací jazyk[6] se skládá z formule stavu, která popisuje jednotlivé stavy, a z formule cesty, u níž můžeme sledovat vlastnosti dosažitelnosti, bezpečnosti a živosti. Na obrázku 2.2 lze vidět různé formule cest v nástroji Uppaal.



Obrázek 2.2: Na obrázku lze vidět formule cest podporované Uppaalem. Tučné hrany znázorňují cestu, kterou formule vyhodnocuje. Vyplněné stavy jsou ty, pro které daná formule stavu φ platí. Převzato z [1].

Formule stavu (angl. State Formulae) je výraz, který lze vyhodnotit pro daný stav bez znalosti chování modelu. Pod tím si můžeme představit např. výraz `j == 10`, který je pravdivý ve stavu vždy, když platí, že `j` se rovná `10`. Syntaxe formule stavu je nadmnožinou stráží a lze se pomocí výrazu `P.1` (kde `P` je proces a `1` je místo) zeptat, zda je daný proces v určitém místě.

Ve stavech může také nastat `deadlock`, což v tomto kontextu znamená, že není možné provést žádné odchozí přechody ze stavu samotného a ani z žádného z jeho pozdějších následníků. V Uppaalu je značíme jednoduše klíčovým slovem `deadlock`. Tato formule je splněná pro všechny stavy, které se nachází v deadlocku.

Vlastnosti dosažitelnosti (angl. Reachability Properties) nám říkají, že danou formuli stavu φ je možné vyhodnotit jakýmkoliv dosažitelným stavem. Jinak řečeno se ptáme, zda existuje cesta z počátečního stavu taková, ve které se kdykoliv vyhodnotí φ. Jako příklad mohu uvést model komunikačního protokolu, ve kterém je příjemce a odesílatel. Pro ověřeně základního chování modelu dává smysl se ptát, zda příjemce vůbec může zprávu přijmout nebo zda odesílatel může zprávu odeslat. Tím se však negarantuje správnost protokolu.

Vyjádřit, že stav splňující φ je dosažitelný můžeme pomocí formule cesty $E\Diamond\varphi$. V Uppaalu se zapisuje jako `E<>`φ.

Vlastnosti bezpečnosti (angl. Safety Properties) nám říkají, že se nikdy nic špatného nestane. Jako příklad mohu uvést hraní hry, kde za bezpečný stav můžeme považovat takový, ve kterém stále můžeme vyhrát, tudíž je možné, že neprohrajeme. Nebo v modelu

[5] angl. Timed Computational Tree Logic, TCTL
[6] angl. Query Language

autonomního vozidla může být bezpečností vlastnost, že rychlost je vždy (invariantně) pod nějakou danou hranicí nebo že nikdy nedojde k výbuchu motoru.

Nechť φ je formule stavu. Vyjádřit, že φ by mělo být pravdivé ve všech dosažitelných stavech můžeme pomocí formulce cesty $A\Box\varphi$, zatímco $E\Box\varphi$ říká, že by měla existovat maximální cesta taková, že φ je vždy pravdivé. V Uppaalu to můžeme zapsat jako `A[]` φ a `E[]` φ.

Vlastnosti živosti (angl. Liveness Properties) nám říkají, že něco se eventuálně stane, například po stisknutí tlačítka u lampy pro její rozsvícení by se lampa měla eventuálně rozsvítit nebo při poslání zprávy v komunikačním protokolu by zpráva měla být i eventuálně přijata.

V nejjednoduší podobě je živost vyjádřena formulí cesty $A\Diamond\varphi$, kde φ je eventuálně splněno. Ale lepší je použít zápis $\varphi \rightsquigarrow \psi$, což znamená, že kdykoliv je φ splněno, tak eventuálně bude splněno i ψ(když zpráva bude poslána, tak bude i doručena). V Uppaalu můžeme tyto vlastnosti zapsat jako `A<>` φ a φ `-->` ψ.

Vlastnosti Uppaalu

- Jako první se zmíním o **šablonách**. Pomocí šablon tvoříme automaty a jsou definovány množinou parametrů jakéhokoliv typu (`chan, int`). Během deklarace procesů jsou parametry šablon nahrazeny za dané argumenty. Každá šablona může mít své lokální deklarace(funkce, proměnné atd.).[1]

- Dále můžeme používat **konstanty**, které musí mít hodnotu integer a nemůžeme je modifikovat. Jsou deklarovány takto: `const name value`.[1]

- Můžeme deklarovat **ohraničené hodnoty typu integer**, které slouží k vytvoření proměnných s omezeným rozsahem hodnot. Deklarujeme je v tomto tvaru: `int[min, max] name`, kde `min` je spodní hranice a `max` je horní hranice. Tyto proměnné můžeme používat ve strážích, invariantách a přiřazeních. Pokud nemáme hranice specifikovány, tak výchozí rozsah je -32768 až 32768. Ve verifikaci kontrolujeme, zda jsou hranice splněny, a pokud ne, tak nevalidní stav je odstraněn (za běhu).[1]

- Pro některé prvky také můžeme vytvořit **pole**, specificky pro hodiny, kanály, konstanty a proměnné typu integer. Definují se tak, že napíšeme velikost za jméno proměnné, např. `chan c[5];, clock x[3];` a podobně.[1]

- **Inicializátory** slouží k inicializaci proměnných typu integer nebo polí typu integer (např. `int i = 2;`).[1]

- Uživatel si může definovat **funkce** (lokálně k šablonám či globálně) se syntaxí podobnou jazyku `C` až na to, že zde nemáme žádné ukazatele.[1]

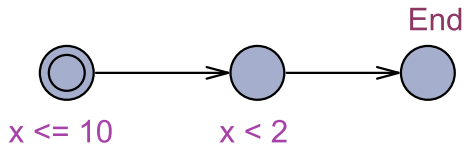
- Můžeme vytvářet **vlastní typy** pomocí `typedef` podobně jako v jazyce `C` a můžeme deklarovat **záznamy (struktury)** jako `struct` podobně jako v jazyce `C`.[1]

Invarianty

K místům v modelu můžeme přidat invarianty. Invarianty jsou výrazy, ve kterých můžeme používat pouze hodiny, proměnné typu integer a konstanty. Invarianty musí být konjunkce podmínek ve formě `x < e` nebo `x` $\leq$ `e`, kde `x` značí hodiny a `e` je libovolná hodnota typu

integer. Invarianty také může zavolat funkce vracející boolovskou hodnotu, ovšem v takových funkcích nelze používat omezení hodin. Jednoduše řečeno nám invarianty říkájí, že nelze setrvat v nějakém stavu déle než určitou dobu. Stavy, které porušují invarianty, nejsou definovány, tudíž neexistují.[1]

Na obrázku 2.3 jednoduchý model, kde v počatečním stavu je invarianta `x` $\leq$ `10`, což znamená, že v tomto stavu není možné být déle než 10 časových jednotek, tudíž se do dalšího stavu přesuneme v rozmezí $\langle 0, 10\rangle$. V dalším stavu nemůžeme čekat více než 2 (ani stejně) časové jednotky, tudíž se do konečného stavu z prostředního přesuneme buď hned nebo až po jedné časové jednotce.

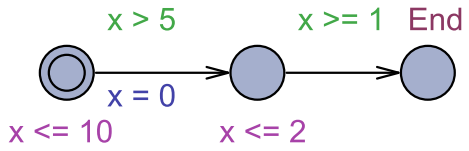


Obrázek 2.3: Příklad invariant. V prvním stavu nesmíme být více než 10 časových jednotek a v druhém nesmíme být 2 časové jednotky nebo více.

Stráže

Stráže jsou pouze u hran. Přechod do nového místa pomocí hrany, na které je stráž, je možný pouze tehdy, pokud je stráž vyhodnocena jako pravdivá. Stráže musí být vyhodnocena pouze jako booleovská hodnota. Je do ní možné zadat pouze hodiny, proměnné typu integer nebo konstanty (či pole těchto typů). Hodnoty hodin či jejich rozdíly je možné porovnávat pouze s hodnotami typu integer. Také je možné zavolat funkce, které vrací hodnotu typu bool, ovšem omezení hodin v těchto funkcích nelze používat. Jednoduše stráže omezují přechody a říkají nám, za jakých podmínek nebo v jakém časovém rozmezí lze danou hranu použít jako přechod do nového místa.[1]

Na obr. 2.4 je příklad modelu používající stráže a invarianty. Z prvního stavu do druhého stavu se můžeme dostat v rozmezí $(5, 10\rangle$ časových jednotek a z druhého stavu do posledního se můžeme dostat v rozmezí $\langle 1, 2\rangle$ časových jednotek.



Obrázek 2.4: Příklad stráží. Z počátečního stavu musíme odejít za více než 5 ale méně nebo stejně jako 10 časových jednotek. Z druhého stavu odejdeme za 1 nebo 2 časové jednotky.

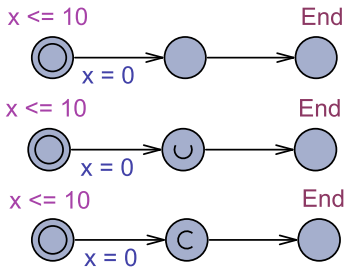
Místa

Tato podsekce je napsána podle [1]. Ukázka míst je na obrázku 2.5.

V nástroji UPAAL máme více druhů míst. Celkově jsou tři, a to:

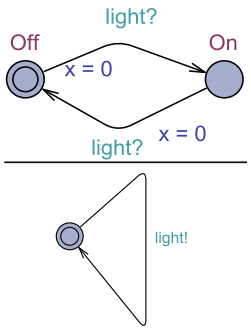
- **Normální**. Tato místa jsou všechny, které jsou bez značení a mohou být s invariantami nebo bez nich.

- **Urgentní (angl. urgent)**. Urgentní místo je značeno pomocí `u` uvnitř místa. V urgentním stavu nemůže běžet čas, tzn. že čas v tomto stavu zamrzne. Sémanticky jsou urgentní stavy ekvivalentní tomu, že přidáme extra hodiny `y`, které jsou resetovány na každé příchozí hraně a je přítomná invarianta `y<=0`.
- **Dedikované (angl. comitted)**. Tato místa jsou značena pomocí `C` uvnitř místa. V dedikovaných místech nemůže běžet čas, stejně jako u urgentních míst. Rozdíl je v tom, že pokud je jakýkoliv proces v dedikovaném místě, tak další přechod, který nastane (u všech automatů), musí být takový, který zarhnuje jednu z hran vycházející z dedikovaného místa, ve kterém proces je. To jednoduše znamená, že pokud proces dorazí do dedikovaného místa a je to jediné aktivní dedikované místo, tak musí okamžitě odejít a nesmí se provést žádný jiný proces, který není v dedikovaném místě.



Obrázek 2.5: Příklad míst. V horním automatu můžeme v prostředním stavu čekat donekonečna. V prostředním automatu v prostředním stavu nemůže uběhnout žádný čas. A v posledním automatu ještě navíc musíme z prostředního stavu odejít ještě dříve než aktivujeme procesy v jakémkoliv jiném automatu.

Kanály



Obrázek 2.6: Příklad kanálů a jejich synchronizace. Dolní model v neurčitou dobu (nemusí ani nikdy nastat) vyšle signál `light!`, podle kterého se vrchní model může přesunout do dalšího stavu pomocí hrany, na které je synchronizace `light?`.

Kanály můžeme deklarovat následovně: `chan c`. Pokud máme hranu, u které je např. `c!`, tak tato hrana se synchronizuje s jinou, u které je `c?`. Kanály můžeme také označit klíčovým

slovem `urgent`, což nám říká, že pokud je na urgentním kanálu povolena synchronizace, tak nemůže nastat zpoždění. Hrany, které mají urgentní kanály, nemohou obsahovat časové omezení, tzn. žádné hodinové stráže.[1] Na obrázku 2.6 je ukázan příklad.

2.2 Rozšíření Uppaal SMC

Celá tato sekce byla napsána podle [7]. Uppaal SMC[7] je nová větev Uppaalu, která dovoluje reprezentovat systém jako sítě automatů, jejichž chování může záviset na stochastických a lineárních dynamických vlastnostech. Každá složka systému je popsána automatem, jehož hodiny se vyvíjí různými tempy. Hlavní náplní SMC je monitorovat některé simulace systému a poté využít statistické výsledky k rozhodnutí, zda systém splňuje vlastnost s nějakou mírou sebejistoty. Metody založené na simulaci zaberou mnohem méně paměti a času a jsou mnohdy jediným řešením. Hlavní rozdíly oproti klasickému Uppaalu jsou takové, že SMC dovoluje specifikovat pravděpodobnosti přechodů a pravděpodobnostní rozložení ve stavech, můžeme vypočítat odhad pravděpodobnosti, porovnávat pravděpodobnosti s hodnotou a porovnávat dvě pravděpodobnosti mezi sebou bez nutnosti výpočtu.

Pravděpodobnosti v modelu

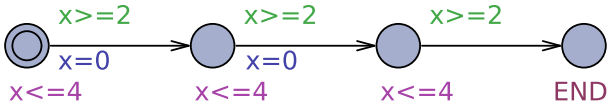
SMC rozšiřuje klasický časovaný automat používaný v originálním nástroji Uppaal. Stochastická interpretace nahrazuje nedeterministické volby mezi různými povolenými přechody pravděpodobnostními volbami (mohou, ale nemusí být definovány uživatelem). Nedeterministické volby časového zpoždění jsou rozšířené o pravděpodobnostní rozložení, které může být buď rovnoměrné či exponenciální (s hodnotami definovanými uživatelem).

Na obrázku 2.7 vidíme tři stochasticky časované automaty A_1, A_2 a A_3. Lze vidět, že u automatu A_1 se dostaneme do konečného stavu v časovém rozmezí $\langle 6, 12\rangle$. V každém stavu budeme mít zpoždění $\langle 2, 4\rangle$ s rovnoměrným rozložením pravděpodobnosti. Tudíž celkový čas dosažitelnosti získáme tak, když sečteme rovnoměrné rozložení ve všech třech stavech. V automatu A_2 se dostaneme do konečného stavu v rozmezí $\langle 4, 12\rangle$. Do horní řady se dostaneme s pravděpodobností $\frac{1}{6}$ a budeme mít celkové zpoždění $\langle 6, 12\rangle$ a do spodní řady se dostaneme s pravděpodobností $\frac{5}{6}$ a budeme mít celkové zpoždění $\langle 4, 8\rangle$. V automatu A_3 se dostaneme do konečného stavu v rozmezí $\langle 0, +\infty)$. V tomto automatu nejsou invarianty a zpoždění je zvoleno na základě exponenciálního rozložení s hodnotami definovanými uživatelem ($\frac{1}{2}, 2$ a $\frac{1}{4}$). Ještě navíc máme šanci $\frac{1}{4}$, že se dostaneme na horní řádek a šanci $\frac{3}{4}$, že se dostaneme na spodní.

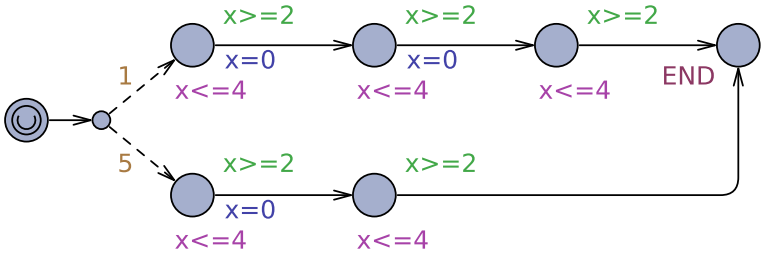
Rozšíření dotazovacího jazyka

Uppaal SMC rozšiřuje originální dotazovací jazyk Uppaalu o další způsoby dotazů související se stochastickým vyjádřením časovaného automatu. Hodnoty různých výrazů a hodnot si můžeme zobrazit graficky (např. ve formě grafů). To nám dává větší přehled a syntaxe je následující: `simulate N [<=bound] { E1,..,Ek}`, kde `N` je přirozené číslo říkající kolik simulací má proběhnout, `bound` je časové ohraničení simulací a `E1,..,Ek` je `k` výrazů, které je třeba zobrazit a sledovat. Příklady budou ukázany v příkladu na konci této kapitoly v sekci 2.4.

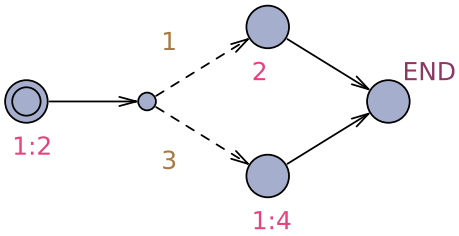
[7] angl. Statistical Model Checking



(a) A_1.



(b) A_2.



(c) A_3.

Obrázek 2.7: Stochasticky časované automaty. Nahoře je automat bez pravděpodobností, uprostřed vidíme pravděpodobnost u hran (čím menší číslo, tím menší pravděpodobnost). A dole vidíme exponenciální rozložení čekání u stavů. Převzato z [7].

2.3 Rozšíření Uppaal Stratego

Uppaal Stratego je nástroj, který integruje nástroj Uppaal a jeho dvě odvětví, Uppaal SMC a Uppaal TIGA[8]. Také nám rozšiřuje dotazovací jazyk tak, že strategie jsou objekty první třídy, které lze vytvářet, porovnávat, optimalizovat či používat při používání metody model checking nad hrou. Se strategií můžeme pracovat pomocí přiřazení `Strategy S =` a používat pomocí výrazu `under S`, kde S označuje strategii. V dalších částech si povíme něco konceptech tohoto nástroje, ukážeme si přehled různých strategií, modelů a vztahů mezi nimi. [6]

Koncepty

Tato podsekce byla popsána podle [13]. Uppaal Stratego má několik hlavních konceptů. První koncept, který bude probírán, je **hra**. Hry jsou zajímavé v tom, že jejich modelování je relativně jednoduché, tudíž místo složitých matematických důkazů nám stačí pouze vyřešit hru. Hra je nějaký matematický model, kde figuruje několik hráčů (procesů), kde každý z nich má své nezávislé cíle a mnohdy spolu soupeří a mohou být ve hře konfliktní. Nejjednoduší hra se skládá ze dvou hráčů. Aby hráč vyhrál, tak musí splnit nějaký cíl nebo cíle, které mohou být splnitelné pouze jednou nebo i vícekrát. Hra může být symetrická,

[8] angl. synthesis for timed games

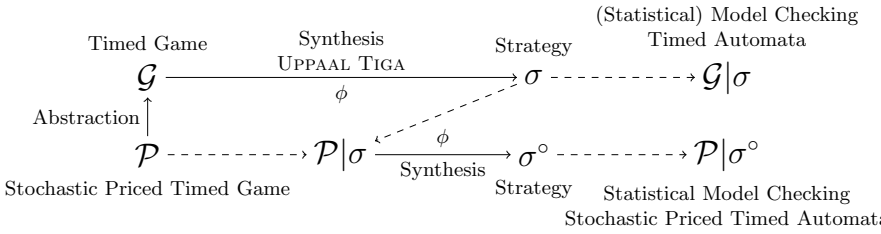
kdy hráči mají stejné startovní pozice i stejné cíle, ale určitě to tak nemusí být a většinou to tak není. Hráči mohou mít úplně jiné cíle a přesto hrát ve stejném prostředí, hra taky nemusí být spravedlivá pro všechny strany. Pro vyřešení hry je třeba najít výherní strategii, která vede do cíle nebo dokázat, že cíl není splnitelný.

Dalším důležitým konceptem je **strategie**. Strategie je plán nebo předpis, který hráče vede k jeho cíli (vítězství) a určuje akce (přechody) toho hráče ve všech možných situacích (stavech). Není zaručené, že vítězná strategie bude vždy existovat. Strategie můžeme rozdělit na deterministické a nedeterministické. Deterministická strategie specifikuje jednu akci za stav a nedeterministická strategie může specifikovat několik alternativ. Plně přípustná strategie je taková, která specifikuje všechny alternativní akce vedoucí do cíle.

Další koncept je **časovaná hra**. V této hře jsou dva hráči a je specifikována sítí časovaných automatů, které mají dva druhy hran, a to plné a přerušované. Plné čáry reprezentují akce (přechody) hráče (kterého můžeme ovládat) a čárkované čáry reprezentují akce (přechody) protivníka (kterého nemůžeme ovládat). Když časovanou hru rozšíříme o pravděpodobnosti, tak jí říkáme **stochastická časovaná hra**. Pokud přidáme ještě další rozšíření se spojitými cenovými výrazy, které dovolují estimaci distribuce ceny, tak mluvíme o **stochastické oceněné časované hře**.

Přehled

Uppaal Stratego má různé modely a reprezentace strategií, jejichž přehled a vztahy mezi nimi jsou znázorněny na obrázku 2.8.



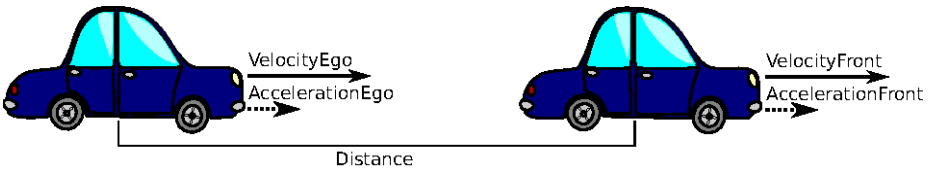
Obrázek 2.8: Přehled modelů, strategií a jejich vztahů v Uppaal Stratego. Plné čáry indikují transformace a výpočty, kdežto čárkované čáry reprezentují znovupoužití objektů. Převzato z [6].

Máme nějakou stochastickou ohodnocenou časovanou hru $\mathbf{P}$, kterou můžeme pomocí abstrakce převést na časovanou hru $\mathbf{G}$ (např. jednoduše tak, že budeme ignorovat ohodnocení a stochastické stránky modelu). Pomocí $\mathbf{G}$ můžeme v Uppaal TIGA syntetizovat strategii $\boldsymbol{\sigma}$. Na tuto strategii souběžně s časovanou hrou $\mathbf{G}|\boldsymbol{\sigma}$ lze aplikovat model checking jako klasicky v Uppaal. Strategii můžeme také použít v $\mathbf{P}$ a obdržet tak $\mathbf{P}|\boldsymbol{\sigma}$. Z $\mathbf{P}$ i z $\mathbf{P}|\boldsymbol{\sigma}$ je možné se učit pomocí určité metody (která je popsána v [5]). Učící algoritmus se může optimalizovat pod strategií $\boldsymbol{\sigma}$ (pokud je plně přípustná a zaručuje nějaký cíl) a výsledkem bude téměř optimální strategie $\boldsymbol{\sigma}^\circ$, která má záruky strategie $\boldsymbol{\sigma}$. Abychom mohli provádět statistický model checking, tak je ještě třeba vytvořit $\mathbf{P}|\boldsymbol{\sigma}^\circ$.[6]

2.4 Příklad: bezpečný a optimální tempomat

Tento příklad byl vytvořen podle [11] a celý popsán podle [12]. Řeší se zde tempomat, který má za úkol udržovat stabilní rychlost vozidla a udržovat bezpečnost vzdálenost od vozidla

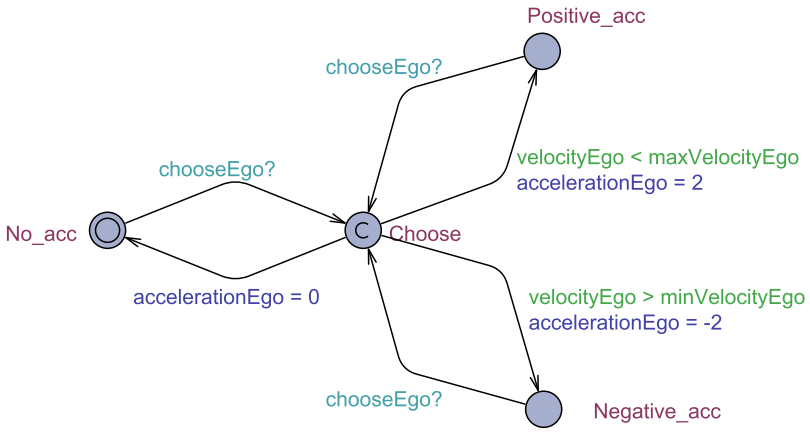
před ním. Z tohoto příkladu lze vidět využití symbolických a statistických technik pouze pomocí nástroje Uppaal Stratego.



Obrázek 2.9: Na obrázku vidíme dvě auta. Naše Ego, které má tempomat a auto Front, což je auto, podle kterého se tempomat řídí. Převzato z [11].

Na obrázku 2.9 máme dvě auta, **Ego** a **Front**. **Ego** je vybaveno senzorem vzdálenosti. Auta mohou zrychlovat a tak se hýbat kupředu, ovšem mohou mít i negativní rychlost, tudíž zpomalovat. Vzdálenost mezi auty se mění dynamicky na základě relativní rychlosti mezi auty. Cílem je vyvinout kontrolní strategii, která nám řekne, zda musíme zrychlit, zpomalit, držet rychlost či začít couvat.

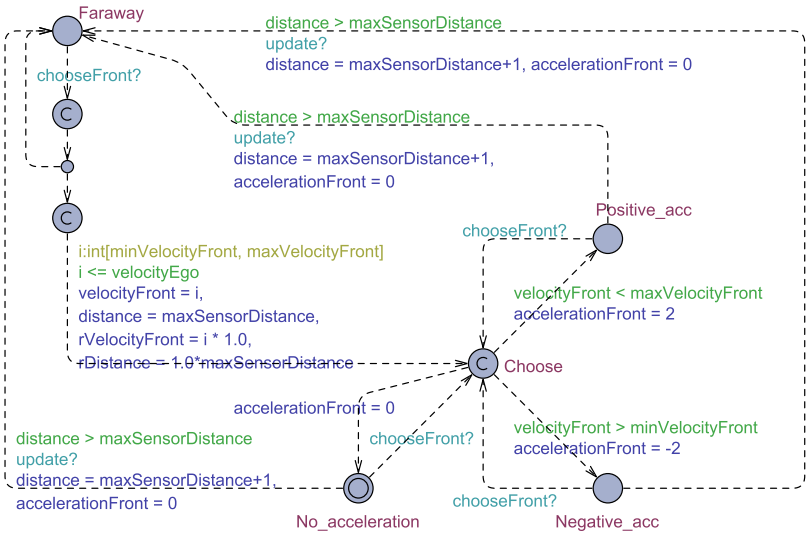
Systém bude modelován jako stochasticky oceněná časovaná hra. V abstrakci se vynechá stochastická a spojitá dynamika. Cílem bude bezpečnost (zda by **Ego** mělo brzdit nebo zrychlovat, ale mělo by udržovat bezpečnou minimální vzdálenost nehledě na to, co se auto **Front** rozhodne udělat). Poté se může tato bezpečná strategie **Ega** vrátit do původního stochasticky ohodnoceného prostředí a optimalizovat bezpečnou strategii ovladače do rychlé strategie pomocí učících technik.



Obrázek 2.10: Model chování auta Ego. Může zrychlovat, zpomalovat nebo rychlost neovlivňovat. Převzato z [12].

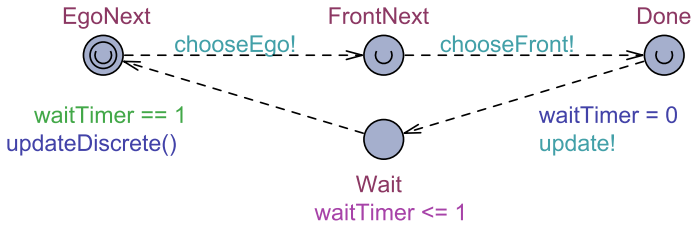
Na obrázku 2.10 je znázorněn model auta **Ego**. Začínáme ve stavu `No_acc`, který značí, že auto nezrychluje ani nezpomaluje. V tomto stavu čekáme na signál `chooseEgo`, který se generuje v automatu `System`. V místě `Choose`, které je dedikované, můžeme jít do stavu, kde budeme mít pozitivní rychlost nebo do stavu, kde budeme mít negativní rychlost. Na strážích kontrolujeme, zda současná rychlost auta (`velocityEgo`) je menší než maximální rychlost auta (`maxVelocityEgo`) nebo větší než minimální rychlost auta (`minVelocityEgo`). Poté je u každé stráže ještě úprava zrychlení (`accelerationEgo`) na pozitivní či negativní. Je zde také možné jít zpátky do stavu, kde nebude žádné zrychlení.

Auto **Front** má čárkované přechody, což znamená, že jej nemůžeme kontrolovat, ale může dělat podobné akce. Ještě navíc se auto **Front** může dostat z dosahu senzorů auta **Ego** (místo `Faraway`). Na obrázku 2.11 je znázorněno chování auta **Front**. Stejně jako auto



Obrázek 2.11: Model chování auta Front. Může zrychlovat, zpomalovat či neovlivňovat rychlost. Také se může dostat z dosahů auta Ego. Převzato z [12].

Ego má stav žádného zrychlení (`No_acceleration`), pozitivního zrychlení (`Positive_acc`), negativního zrychlení (`Negative_acc`) a výběru (`Choose`). Všechny stavy mají stejnou funkcionalitu, ovšem u auta **Front** je pár změn. Ve stavu žádného zrychlení se může stát, že aktuální vzdálenost (`distance`) je větší než maximální dosah senzoru (`maxSensorDistance`). Pokud to platí, tak po obrdžení signálu `update` se dostaneme do stavu `Faraway`, kde už senzor auta **Ega** nebude schopen auto **Front** detekovat. To samé se může stát ve stavu pozitivního zrychlení a negativního zrychlení. Ze stavu `Faraway` se můžeme dostat do stavu, kdy senzor **Ega** zase auto detekuje. V takovém případě se provedou potřebné nastavení rychlosti a vzdálenosti a auto **Front** je zpátky v místě `Choose`.

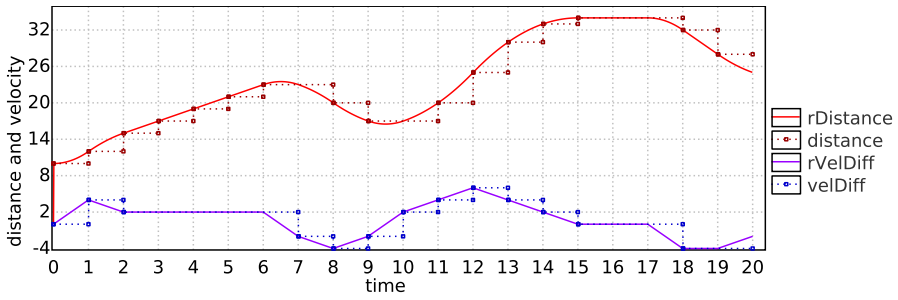


Obrázek 2.12: Plánovač pro auta Ego a Front. Jakmile Ego a Front udělají své akce, tak se aktualizují všechny hodnoty a po jedné sekundě mohou dělat další akce. Převzato z [12].

Na obrázku 2.12 vidíme plánovač, kteří říká, že auta se mohou rozhodovat jednou za sekundu. Vidíme, že první volbu musí udělat auto **Ego**, i když neví, co udělá auto **Front**, které je aktivováno hned potom. Po jedné sekundě se upraví a vypočítají všechny vzdálenosti a rychlosti.

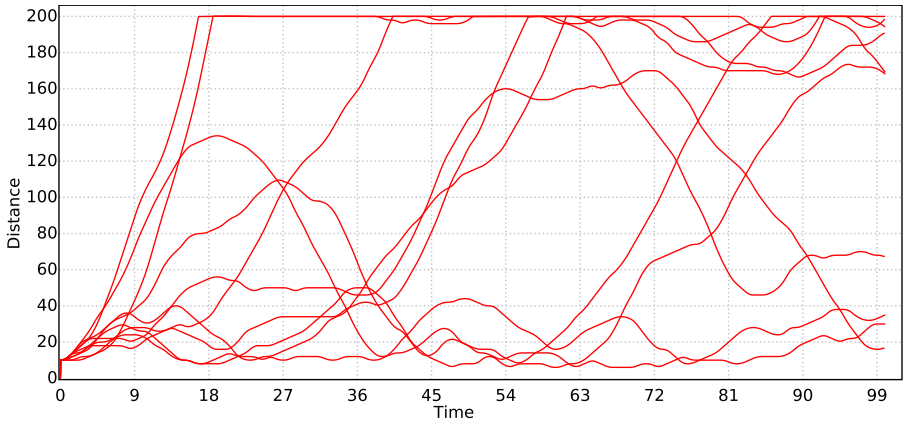
Mějme hybridní hodiny `rVelocityEgo a rVelocityFront`, které označují spojité hodnoty rychlostí a hybridní hodiny `rDistance`, což je spojitá hodnota vzdálenosti mezi auty.

Na obrázku 2.13 vidíme graf[9], který nám ukazuje diskrétní i spojité hodnoty (v určitých časech mají stejné hodnoty) vzdáleností a rozdíl rychlostí.



Obrázek 2.13: Graf pro znázornění spojitých a diskrétních hodnot vzdálenosti mezi auty (červená čára) a rozdíl jejich rychlosti (fialová čára). Převzato z [12].

Ovšem když zkontrolujeme následující podmínku `A[] distance > 5` (vzdálenost mezi auty je ve všech stavech větší než 5), tak zjistíme, že není splněna, tudíž nemáme zajištěno bezpečí. Podíváme se na problém jako na hru, kde auto **Front** je nepřítel a auto **Ego** hráč, kterého kontrolujeme, a cílem je zachovat bezpečnou vzdálenost. Pomocí Uppaal Tiga se odstraní (abstrakce) spojitá dynamika a výslednou strategii uložíme pod jménem `safe` následovně: `strategy safe = control: A[] dist > 5`. Syntetizovanou strategii můžeme ověřit tak, že spustíme simulaci pod strategií `safe`, kde sledujeme vzdálenost mezi auty. Graf lze vidět na obrázku 2.14, kde lze vidět, že vzdálenost mezi auty je vždy větší než 5, tudíž už nám nevznikají žádné nehody. Graf byl vytvořen pomocí následujícího příkazu: `simulate 10 [<=100] rDistance under safe`.



Obrázek 2.14: Graf pro znázornění vzdáleností mezi auty podle strategie `safe`. Lze vidět, že vzdálenost není nikdy menší než 5, tudíž nám už nevznikají nehody. Převzato z [12].

[9] vytvořen příkazem: simulate 1 [<=20] {rDistance, dist, rVelocityFront - rVelocityEgo, velocityFront - velocityEgo}

Kapitola 3

Popis řešených oblastí, jejich modifikace a návrh řešení

Pro hledání strategií bylo třeba vymyslet či najít problémy, které se mohou řešit a u kterých má smysl hledat nějaké strategie. Šachy, což byl první problém, který byl zvolen k řešení, je komplexní a obsáhlý, tudíž další problémy byly vybírány s ohledem na větší jednoduchost a proveditelnost (případně byla snaha zjednodušit model oproti modelovanému systému). Důvodem je to, že u komplexních modelů může být vyhodnocování strategií velmi náročné a možnosti nástroje nám nemusí stačit na úspěšné vyřešení problému.

Celkově byly vybráno k řešení 4 problémy. Ke každému problému byl vytvořen nějaký systém času, který musí dodržovat s ohledem na reálné vlastnosti (např. jak dlouho trvá danému hráči udělat tah nebo přesunout nějaký prvek na jiné místo je založen na odhadu toho, jak dlouho by mu to trvalo ve skutečnosti). Všechny problémy, jejich popis a mé nápady či návrhy k nim jsou popsány v následujících sekcích.

3.1 Návrh řešení šachů a jejich zjednodušení

První a nejsložitější problém k namodelování byly šachy. Počítám s tím, že čtenář zná obecná pravidla šachů a popíšu zde všechny důležité náležitosti související s pravidly a časem. Každý hráč by měl mít na svůj tah nějaké omezené množství času a existuje spoustu turnajových pravidel, které se v tomto omezení velmi liší a jsou vytvořeny na základě jiných kritérií. Řídil jsem se podle varianty, kterou používá Mezinárodní šahová federace FIDE[1], a to 90 minut na prvních 40 tahů a nějakou rezervu na další tahy. Na základě statistik[2] se počet tahů na jednu hru pohybuje kolem 40, proto jsem zvolil časové kritérium na tah 3 minuty (maximum času na prvních 40 tahů je tudíž 120 minut, aby i zde byla nějaká rezerva) s tím, že tato hodnota musí být větší než nula, jelikož hráč nemůže svůj tah provést okamžitě (vždy bude alespoň nějakou dobu trvat, než pohne figurkou).

Další problém, který bylo třeba vyřešit, je reprezentace herního pole a šachových figurek. Vzhledem k tomu, že Uppaal nepodporuje datový typ `String`, jsem se rozhodl pro číselné značení. Každá figurka je označená třímístným ID, kde první hodnota označuje barvu figurky, druhá typ figurky a třetí pořadí figurky zleva doprava (pokud je pro daný typ více figurek). Na obrázku 3.1 lze pak vidět, jak vypadá šachovnice se značením figurek podle tabulky 3.1.

[1] francouzsky Fédération Internationale des Échecs
[2] https://www.chessgames.com/chessstats.html

Číslo	Barva	Typ
0		Pěšec
1	Bílá	Věž
2	Černá	Jezdec
3		Střelec
4		Královna
5		Král

Tabulka 3.1: Tabulka ukazující přiřazení hodnot k barvám a typům figurek.

110	120	130	140	150	131	121	111
100	101	102	103	104	105	106	107
200	201	202	203	204	205	206	207
210	220	230	240	250	231	221	211

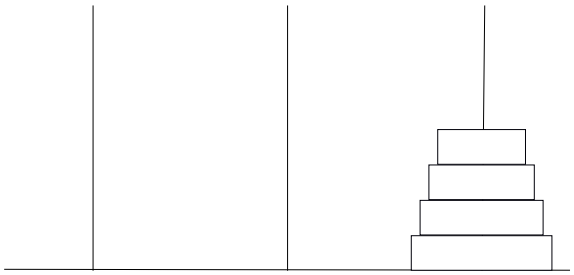
Obrázek 3.1: Obrázek ukazující šachovnici s figurkami podle třímístného ID.

Vzhledem ke komplexnosti modelu a z důvodu větší proveditelnosti byl model zjednodušen či pozměněn několik způsoby oproti normálním šachovým pravidlům. Aby hráč mohl vyhrát, tak stačí, aby protivníkovi sebral krále, což je sice podmínka pro šach mat, ale pokud v mém modelu hráč nebo protivník dostane šach, tak nemusí další tah hrát tak, aby už šach neměl. Podobně hráč či protivník může táhnout figurkou tak, aby umožnil protivníkovi sebrání krále. Další návrh na zjednodušení by byl takový, že pokud by nastal šach, tak hráč může hýbat pouze s králem, ovšem nemohl by sebrat útočící figurku (jinou figurkou) nebo zablokovat tento útok. Tento návrh je také možné implementovat, ovšem působí to jako více omezující podmínka, která mi nepřijde vhodná kvůli nemožnosti blokovat útočníka. Pokud nebudeme aplikovat tato pravidla šachů, tak alespoň je zde možnost se naučit chránit krále, i když to není pravidlem (podmínkou prohry však ano). Také není povoleno provést rošádu.

3.2 Popis hanojských věží a návrh řešení

Dalším problémem, který byl vybrán k řešení, jsou Hanojské věže. Jedná se o hlavolam, který vymyslel v roce 1883 matematik Édouard Lucas. Máme k dispozici tři tyče a libovolný počet disků různých rozměrů (i když v ukázkách jich většinou bývá méně než 10). Na obrázku 3.2 lze vidět hanojské věže se čtyřmi disky. Úkolem je přemístit tyto disky na prostřední tyč, kde nahoře bude nejmenší disk a dole bude největší disk. Hlavolam se řídí následujícími pravidly:

- V jednom tahu lze pohnout pouze jedním diskem.
- V každém tahu vezmeme nejvrchnější disk z jedné tyče a vložíme jej na vrchol jiné tyče či na prázdnou tyč
- Nelze položit větší disk na menší disk



Obrázek 3.2: Obrázek ukazující hanojské věže se čtyřmi disky a třemi tyčemi. Cílem je dostat tyto disky na prostřední tyč ve stejném složení.

Můj model pracuje se čtyřmi disky, jelikož není třeba volit zbytečně velký počet disků a na otestování strategií čtyři budou stačit. Disky jsou označený číslem v sestupných hodnotách, tj. největší disk má hodnotu 4 a nejmenší disk 1. V tomto modelu bude pouze hráč a šablona, která bude provádět jeho akce, jelikož zde není třeba žádný protihráč. Co se týče časových podmínek, tak hráč má na přemýšlení a vzatí disku 60 sekund s tím, že minimum jsou tři sekundy, kdyby se rozhodl hned a okamžitě vzal a následně položil disk. Model nebylo třeba nijak zjednodušovat.

3.3 Popis posuvného pole a návrh řešení

Posuvné pole je hlavolam, který vymyslel Noyes Chapman v roce 1880. Jedná se v podstatě o dvoudimenzionální plochu ve tvaru čtverce s libovolnými rozměry (např. 4x4, 7x7, 5x5 atd.). Na této ploše mohou být všelijaké symboly, obrázky nebo čísla, podle toho, co chce člověk řešit. Na ploše je vždy jeden prázdný dílek, kterým lze posunovat do čtyř směrů (nahoru, dolů, doleva, doprava) a pokaždé, když se tento dílek takto posune, vymění své místo s dílkem, který byl na dané pozici. Je možné posouvat pouze s tímto prázdným dílkem a není možné manipulovat s žádnými jinými dílky. Na obrázku 3.3 můžeme vidět příklad posuvného pole.

3	6	4	8
2	11	7	15
5	9	12	14
10	13	1	

Obrázek 3.3: Posuvné pole o velikosti 4x4 s prázdným místem v pravém dolním rohu. Cílem je čísla seřadit od 1 do 15 zleva doprava.

V mém modelu pracuji s rozměrem 3x3 a s hodnotami typu `integer`. Cílem hráče je vytvořit sekvenci 1 až 8 v daném tvaru. V tomto modelu bylo třeba udělat pouze hráče a model, který bude vykonávat jeho akce, ovšem jsou zde zabudovány i prvky, aby počáteční sestavení čísel v poli šlo libovolně zadávat bez toho, aniž by se muselo cokoliv jiného upravovat. Z časového hlediska u tohoto typu hlavolamu hráč nebude příliš dlouho přemýšlet, než udělá nějakou akci, proto má na každý tah maximálně 10 sekund s tím, že pokud se rozhodne okamžitě, bude trvat alespoň dvě sekundy, než udělá všechny potřebné akce.

Model není nijak zjednodušen a rozměr 3x3 byl zvolen jako adekvátní velikost pro hledání strategií z důvodu menší složitosti.

3.4 Popis kinematického problému a návrh řešení

Tento příklad je udělán podle příkladu z [8]. Jedná se o kinematický problém, ve kterém figurují tři prvky: letadlo, balíček a auto. Letadlo letí v dané výšce $\boldsymbol{d_l}$ v metrech s danou horizontální rychlostí $\boldsymbol{v_l}$ v metrech za sekundu a nulovou vertikální rychlostí. V tomto letadle je balíček, který je v počátečním stavu v klidu a jakmile bude ve správné pozici, bude vypuštěn. Na zemi jede auto, které je $\boldsymbol{w}$ metrů široké a jede rychlostí $\boldsymbol{v_a}$ metrů za sekundu. Auto musí chytit balíček a začíná $\boldsymbol{X}$ metrů před letadlem, jelikož kdyby začínalo za letadlem, tak by jej nebylo možné dohnat. Otázka v zadání je: Jaké auto musí mít vzdálenost $\boldsymbol{X}$, aby balíček mohlo chytit? Budeme pracovat s následujícími vzorečky:

$$\boldsymbol{t_{bal}} = \sqrt{\frac{\boldsymbol{2d_l}}{\boldsymbol{g}}} \tag{3.1}$$

$$\boldsymbol{d_{bal}} = \boldsymbol{v_l t_{bal}} \tag{3.2}$$

$$\boldsymbol{d_{auto}} = \boldsymbol{v_a t_{bal}} \tag{3.3}$$

$$\boldsymbol{X} = \boldsymbol{d_{bal}} - \boldsymbol{d_{auto}} \tag{3.4}$$

Vzoreček 3.1 říká, jak dlouho bude balíčku trvat, než dopadne na zem. Poté pomocí vzorce 3.2 lze zjistit, jakou vzdálenost balíček za tuto dobu urazí horizontálně. Následně si zjistíme pomocí vzorečku 3.3 vzdálenost, kterou urazí auto za tuto dobu. A nakonec se vzorečkem 3.4 zjistíme, jak daleko auto musí být, aby balíček chytilo.

V mém modelu bylo zadání upraveno a trochu zkomplikováno. Auto i letadlo budou moci zpomalovat a zrychlovat a letadlo bude moci ještě klesat a stoupat ve vertikálním směru. Každá akce je omezená nějakou maximální či minimální hodnotou, které byly zvoleny tak, aby modely a akce byly uvěřitelné a blízké realitě (tzn. nemohou se vyskytovat žádné neuvěřitelné hodnoty rychlosti či pozice). Každý prvek bude samostatný a budou provádět své akce nezávisle na sobě (akce letadla neovlivní to, co udělá auto). Pro časový systém jsem zvolil provádění všech akcí letadla i auta každou 1 sekundu. Po této době se také upraví a vypočítají všechny hodnoty. Jiný návrh by obsahoval i samostatný model pro balíček, ovšem ten by mohl být poněkud zbytečný, jelikož by nemohl dělat jiné akce než padat (pokud by v sobě neměl zabudovaný nějaký pohybový systém).

Kapitola 4

Implementace modelů a jejich testování

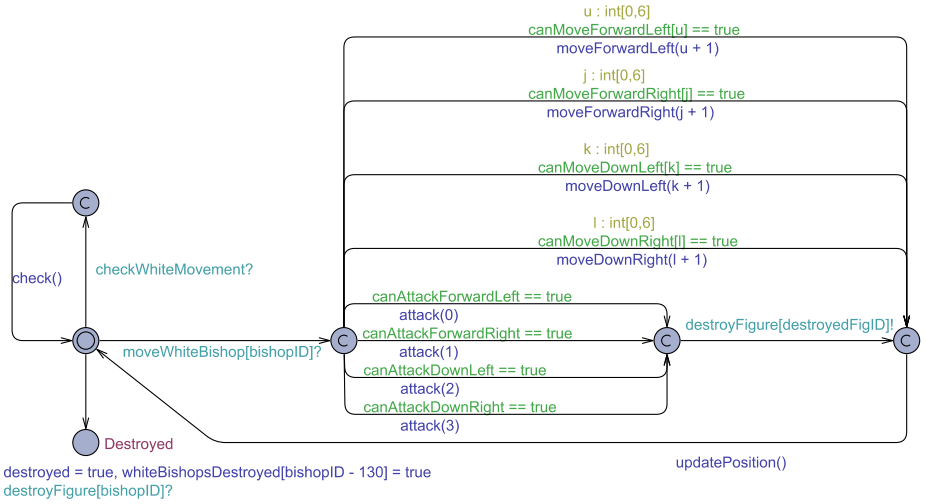
V této kapitole se zaměřím na popis implementace šablon a případně některých funkcí. Nebudu zde popisovat všechny šablony, jelikož v některých modelech jich je více, a zaměřím se pouze na ty nejdůležitější šablony a prvky. Testování bylo prováděno při počátečních fázích například v simulátoru, kde se dají testovat všechny jednoduché a krátké přechody, ovšem hotové modely už byly testovány pomocí různých verifikací. Používal jsem například verifikace pro všechny stavy, které kontrolují splnění daných podmínek nebo zda je možné nějakého stavu dosáhnout, abych našel specifické chybové stavy či hodnoty.

4.1 Implementace a testování šachů

Každá figurka má velmi podobný model, který se liší podle jejího způsobu pohybu a nějakých speciálních pravidel, které se dané figurky mohou týkat (např. pohyb pěšáka z první pozice může být až o dvě políčka). Všechny figurky také znají své vlastní pozice a mají přístup ke globální herní ploše, tudíž nemají problém znát své okolí.

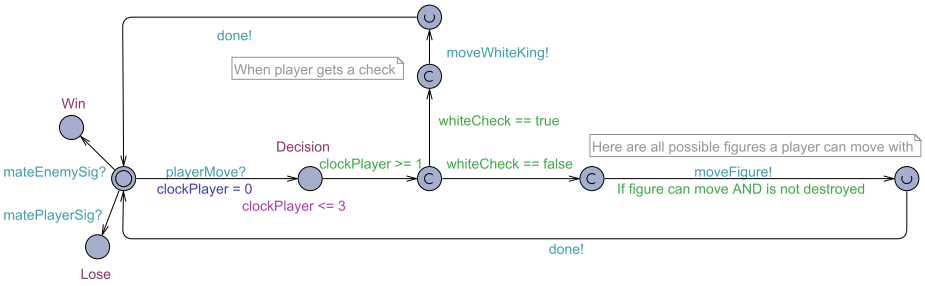
Jako příklad se zaměřím na šablonu figurky bílého střelce, který lze vidět na obrázku 4.1. V této šabloně v počátečním stavu čekáme na jeden ze signálů `checkWhiteMovement, moveWhiteBishop nebo destroyFigure`. Signál `checkWhiteMovement` je na začátku posílán všem bílým figurkám, aby zkontrolovaly všechny své možnosti pohybu a útoků pomocí funkce `check()` a na základě toho jsou upraveny indikační proměnné typu `bool`, které značí, kterými figurkami hráč může pohnout i jak se mohou hýbat. Pomocí kanálu `destroyFigure` signalizují ostatní figurky úspěch zničení aktuální figurky. Pokud se hráč rozhodne z touto figurkou zahrát, tak jí signalizuje tuto akci pomocí kanálu `moveWhiteBishop` a figurka se přesune do stavu rozhodování, co vlastně udělá. Může se buď pohnout nebo útočit. Hrany jsou chráněny strážemi, které nastavuje funkce `check()` a figurka může vybrat platnou hranu (která pomocí `select` má více variant) a udělat daný pohyb. V případě útoku lze vybrat směr a poté ještě zaslat signál `destroyFigure` zničené figurce, aby se mohla zničit a deaktivovat. Úplně na konec je třeba ještě aktualizovat novou pozici dané figurky a zanést ji do herní plochy, což se vykoná pomocí funkce `updatePosition()`.

Další důležitou šablonou je šablona hráče a nepřítele. Budu zde popisovat pouze šablonu hráče, jejíž kostru lze vidět na obrázku 4.2 a kompletní šablona je v příloze A.1. Nepřítel je téměř totožný, ovšem nelze ho ovládat. V počátečním stavu hráč, jakmile je signalizován pomocí kanálu `playerMove` indikující, aby začal hrát, může přejít do stavu rozhodování



Obrázek 4.1: Obrázek znázorňující šablonu figurky bílého střelce. Ostatní figrurky vypadají velmi podobně, pouze se liší ve formách pohybů a útoků.

`Decision`, kde se bude čekat, než udělá rozhodnutí. Následně se přesune do stavu, který je zde připraven pro hraní s mechanikou šachu, ovšem tato mechanika ještě není využita, tudíž se rovnou přesune do stavu, kde si může vybrat, jakou figurkou zahraje. Hráč může danou figurkou hrát pouze v případě, pokud není zničena a pokud je s ní vůbec možné udělat nějaký pohyb. Jakmile tuto figurku vybere, tak ji signalizuje pomocí specifického kanálu a dostane se do stavu, ze kterého může přejít zpátky do startovního stavu. Musí dát ovšem ještě signalizovat ovladač pomocí kanálu `done`, že ukončil svou aktivaci.



Obrázek 4.2: Obrázek kostry šablony hráče. Ukazuje hráčovy možnosti, jak dlouho trvá jeho tah a do jakých stavů se může dostat. Hráč zde může prohrát, vyhrát, rozhodovat se a nakonec pohnout figurkou (také má omezené možnosti,pokud dostane šach).

Poslední šablonou je ovladač `controller`, který je velmi jednoduchý a řídí posloupnost událostí. Vesměs pouze signalizuje všechny figurky pomocí kanálu `checkWhiteMovement`, aby zkontrolovaly své možnosti. Hráče signalizuje pomocí kanálu `playerMove`, aby mohl hrát. Poté udělá to samé s nepřátelskými figurkami a nepřátelským hráčem a celé se to stále opakuje. Jsou zde i 2 stavy, do kterých se může ovladač dostat, pokud byl zničen král (tudíž nastal šach mat) a tím svou činnost ukončit.

V testování bylo třeba se soutředit především na správné fungování figurek a na to, aby nebyly porušeny žádné zásadní stavy hry. U figurek bylo třeba zkontrolovat, zda mohou překročit hraníce herní plochy nebo vstoupit na pro ně nemožná místa (dle pravidel jejich

hraní). Pár těchto problémů bylo odhaleno i při vykonávání strategií, kdy se nalezly chybové stavy v polohách figurek, ovšem pro specifickou kontrolu byly vytvořeny následující verifikace (pro všechny figurky):

```
A[] whitePawn(100).x <= 7 && whitePawn(100).x >= 1
E<> whitePawn(100).x > 7 || whitePawn(100).x < 1
```

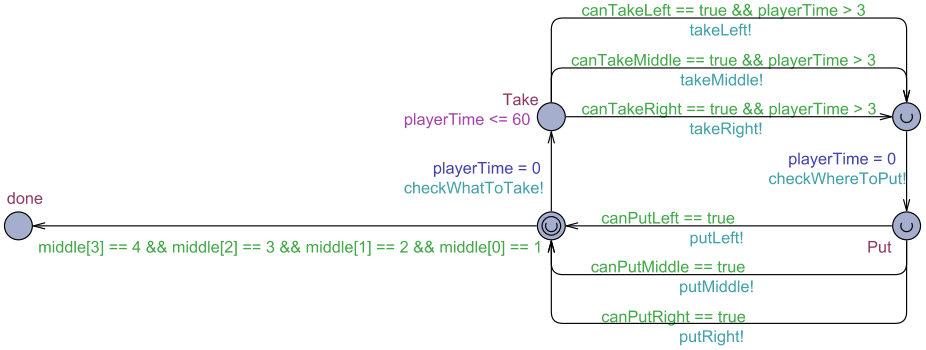
Zde se ptáme, zda pro všechny stavy platí, že bílý pěšák nepřekročí hranice herního pole (a nedostane se na počáteční řadu, protože jeho pohyb to neumožňuje). Taky je možné napsat dotaz stejně jako v druhém případě, kde se ptáme, zda existuje stav, kde figurka porušuje hranice svého pohybu, což je mnohem rychlejší, pokud takový stav existuje. Tyto testy byly napsány pro každý typ a barvu figurek a testují i sloupce, nejen řady.

Další testy ověřovaly především stavy, do kterých se během hry můžeme dostat a které by měly platit. Napíšu zde alespoň nějaké příklady. V prvním případě se ptáme, zda je možné, aby se hráč dostal do vítězného stavu a zároveň by černý král nebyl zničen, což by nemělo platit. V druhém případě se ptáme na to, že pokud je černý král zničen, tak to pro všechny stavy implikuje, že hráč dosáhl vítězného stavu, což by mělo platit. Příkazy lze vidět níže:

```
E<> player.Win && blackKingDestroyed == false
A[] blackKingDestroyed == true imply player.Win
```

4.2 Implementace a testování hanojských věží

Z globálních hodnot jsou nejdůležitější konstanty pro disky, které mají hodnoty 1 až 4 a proměnné tyčí jako pole, na jejichž indexech jsou umístěny hodnoty disků. Je zde také hodně pomocných proměnných, indikátorů a pomocných funkcí. Byly vytvořený pouze dvě šablony, kde jedna je pro hráče a druhá je pro ovladač.



Obrázek 4.3: Šablona pro hráče hanojských věží znázorňující akce, které provádí. Může se dostat ke správnému výsledku, brát disky z věží a poté vkládat disky na věže.

Jako první budu mluvit o šabloně `player` znázorňující hráče, která lze vidět na obrázku 4.3. První hráč signalizuje ovladač pomocí kanálu `checkWhatToTake` pro kontrolu toho, ze kterých tyčí je možné vzít disk a následně přejde hráč do stavu, kde se rozhoduje podle času popsaného v kapitole 3. Až se rozhodne, signalizuje ovladač a přejde do stavu, ze kterého poté pomocí kanálu `CheckWhereToPut` zjistí, kam je možné daný disk uložit. Po vyhodnocení tohoto kanálu se hráč nachází ve fázi ukládání disku, ze které signalizuje

ovladač, aby vložil disk na zvolené místo a vrací se zpět do počátečního stavu. Z tohoto stavu lze také přejít do stavu `done`, který indikuje, že hráč nalezl řešení a není tedy důvod dále pokračovat v hraní.

Šablona `controller` (ovladač) vypadá velmi obdobně a slouží pouze k vykonávání potřebných funkcí podle kanálů, které mu posílá hráč. Vše by bylo možné udělat pouze do jedné šablony, ale rozhodl jsem se to udělat takto z důvodů srozumitelnosti a přehlednosti, tudíž u této šablony jsou implementovány všechny potřebné funkce k provádění operací nad hanojskými věžemi. Nachází se v ní také funkce `reset()` sloužící k navrácení všech indikátorů do výchozích hodnot, aby nevznikaly chyby při vícenásobném průchodu.

Zde se v testování bylo potřeba soustředit na každou tyč a kontrolovat, zda se na ní neobjevuje špatná funkcionalita či nevalidní hodnoty. U každé tyče se kontroluje, zda existuje nějaký stav, kde by pozice na vyšších místech obsahovala větší disk než pozice na nižších místech, což nemůže nastat. Také se zde kontroluje ve všech stavech, že na žádné místo nejde vložit disky jiných velikostí než těch zadaných. Ukázky verifikací jsou k dispozici níže:

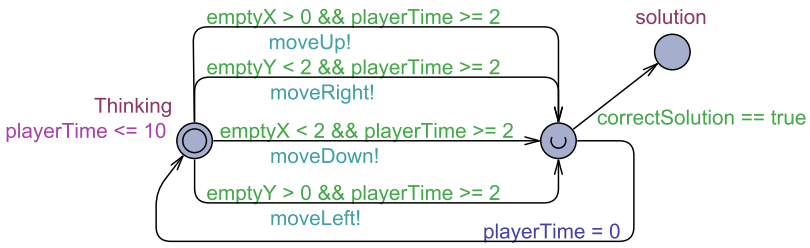
```
E<> middle[2] > middle[3]
A[] middle[0] >= 0 && middle[0] <= 4
```

Dále je zde pár kontrol stavu her a jiných prvků. Například níže ukazuji kontrolu, že nikdy nevezmeme disk, který má jinou velikost než disky, které máme k dispozici. Je zde také kontrola toho, že pokud hráč dorazil do konečného stavu, tak to implikuje disky vložené na správných pozicích. Ukázka:

```
A[] taken < 5 and taken >= 0
A[] player.done imply middle[3] == 4
```

4.3 Implementace a testování posuvného pole

Zde máme globální hodnoty, kterými značíme, kde se nachází prázdné místo a vícedimenzionální herní plochu (v našem případě o rozměrech 3x3), jejíž hodnoty lze libovolně měnit pro hledání strategií různých rozpoložení. Změna dimenzí je také možná, ovšem to už by bylo třeba více úprav. Nachází se zde také velmi malá šablona `Finder` o dvou stavech, která se spustí na úplném začátku a slouží k nalezení prázdného místa v modelu. To nám dovoluje pracovat s jakýmkoliv rozložením hodnot v posuvném poli.



Obrázek 4.4: Obrázek šablony pro akce hráče posuvného pole, které může vykonat. Může s prázdným místem pohnout nahoru, dolů, doprava a doleva. Také se může dostat k výsledku se správnými hodnotami v poli.

Na obrázku 4.4 lze vidět šablonu hráče, ve které se v počátečním stavu hráč rozhoduje, co udělá. Jakmile se rozhodne, kam posune prázdné místo, zašle signál ovladači, který udělá

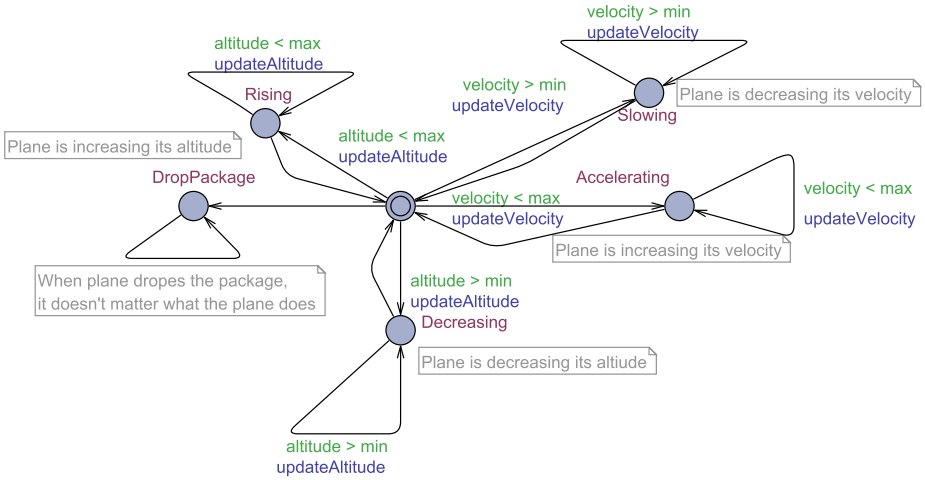
všehny potřebné změny na herním poli a dostane se do stavu, ze kterého může jít buď zpátky na startovní stav nebo stavu `solution`, který říká, že správné řešení bylo nalezeno. Stejně jako v Hanojských věžích zde nebylo nutné dělat hráče a ovladač zvlášť, ovšem je to tak implementováno z důvodu větší přehlednosti.

U posuvného pole v testování si bylo třeba dát pozor, aby prázdné místo (neboli 0) se pohybovalo vždy v rámci toho pole a nikdy jej neopustilo, což by vedlo do špatného stavu. Také je třeba si dát pozor na jiné hodnoty uvnitř pole, jelikož by byla chyba, kdyby se v něm vyskytovaly jiné hodnoty než by tam pro dané rozestavení měly být. Také se zde kontroluje samotné řešení tak, že pokud se hráč dostal do stavu řešení, tak to implikuje, že na daném místě v poli je správná a konečná hodnota. Ukázky těchto verifikací jsou ukázány níže:

```
A[] emptyX >= 0 && emptyX <= 2
A[]puzzle[0][0] <= 8 && puzzle[0][0] >= 0
A[] Player.solution imply puzzle[0][0] == 1
```

4.4 Implementace a testování kinematického problému

Zde si je třeba uchovávat u globálních proměnných všechny konstanty maximálních a mininálních hodnot stejně jako proměnné všech aktuálních a počátečních hodnot. Počáteční hodnoty jsou buď pevně dané nebo je lze přepsat na náhodné hodnoty v daném rozmezí. Jsou zde také připraveny proměnné pro výsledky důležitých výpočtů. Všechny výsledné hodnoty jsou typu `integer`, jelikož práce s hodnotami typu `double` u hran způsobovala problémy, což může způsobovat jistou nepřesnost, ale výsledek můžeme ve verifikacích hledat v daném rozsahu.

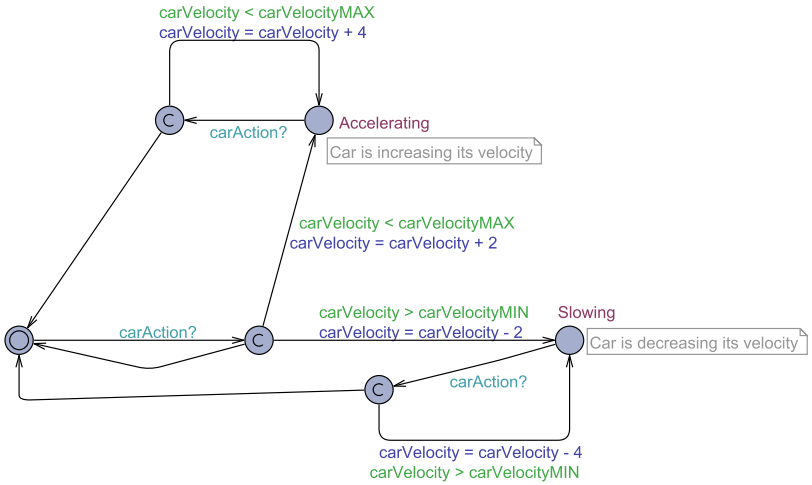


Obrázek 4.5: Obrázek kostry pro šablonu letadla. Poté, co je letadlo signalizováno, aby udělalo akci, může vzlétnout výše či níže nebo může zrychlit, zpomalit a shodit balíček. V těchto stavech může letadlo zůstat a nadále provádět danou akci nebo přestat dělat danou akci a vrátit se do počátečního stavu.

Na obrázku 4.5 lze vidět kostru šablony pro letadlo a kompletní šablona pro letadlo je v příloze A.2. Jakmile je letadlo signalizováno ovladačem pomocí kanálu `planeAction`, tak se může rozhodnout, jakou akci udělá. Může také neudělat nic a vrátit se zpět do

počátečního stavu, tudíž žádné hodnoty se měnit nebudou. Letadlo může letět výše či níže, ale jelikož není reálné, aby letadlo letělo nahoru či dolů čistě vertikálně, tak se jeho pozice zvýší či sníží za sekundu pouze o polovinu jeho rychlosti. Jakmile z těchto stavů dostane další pokyn k akci, může zůstat a dále měnit svou pozici nebo se vyrovnat a přejít do startovního stavu. Obdobně letadlo může zvyšovat či snižovat svou rychlost. Jakmile svou rychlost změní a bude ve stavu zrychlování či zpomalování, tak po další signalizaci mohou ještě více změnit svou rychlost nebo se vrátit zpátky do počátečního stavu. Tyto akce by měly simulovat více stupňů zrychlování/zpomalování (i když velmi jednoduše). Poslední důležitá akce, kterou letadlo může udělat, je shodit balík. Jakmile se letadlo dostane do tohoto místa, tak už akce letadla přestávají být relevantní a zajímá nás pouze to, zda byl balík vypuštěn ve správnou chvíli.

Na obrázku 4.6 máme šablonu pro auto, která je obdobná letadlu. Jakmile je signalizováno pomocí kanálu `carAction`, tak nemusí dělat nic či zrychlovat a zpomalovat. Pokud zrychlí či zpomalí, tak má potom při další signalizaci možnost ještě více došlápnout na plyn či brzdu a zvýšit či snížit rychlost. U auta nás vždy zajímá, co dělá. V případě implementace šablony pro padající balíček by auto mohlo ještě stále upravovat svou rychlost a pozici, aby jej mohlo chytit.



Obrázek 4.6: Obrázek šablony auta. Poté, co je auto signalizováno, aby udělalo akci, může zrychlit, zpomalit nebo udržovat stejnou rychlost. V těchto stavech může auto zůstat a nadále provádět danou akci nebo přestat dělat danou akci a vrátit se do počátečního stavu.

Šablona pro ovladač je velmi jednoduchá. Po každé sekundě aktualizuje všechny relevantní hodnoty a signalizuje letadlo a auto k provedení akce.

Jelikož model není příliš omezen v tom, co může dělat, tak v testování bylo nejdůležitější se soustředit na to, aby se neobjevovaly nesmyslné hodnoty a aby se pohybovaly v rozumných mezích. Byly tedy vytvořeny verifikace pro kontrolu, zda se rychlosti letadla i auta, stejně jako výška letadla, pohybují ve správných mezích (nepřekročují minima a maxima). Také zde kontroluji, zda vypočtené hodnoty nedosahují nějakých nesmyslných hodnot (např. zda čas pro dopad balíčku na zem nebyl menší než 0). Ukázky verifikací jsou níže, které by neměly uspět:

```
E<> carVelocity > carVelocityMAX
E<> packageHitGround < 0
```

Kapitola 5

Popis a vyhodnocování strategií

V této kapitole se soustředím především na popsání různých strategií, které se dají využít v daném typu problému. Pokud je možné tyto strategie spustit a plně provést, tak jsou zde ukázany i výsledky většinou ve formě grafů ze simulací různých běhů nebo pravděpodobnostních verifikací. Bohužel se u některých modelů objevil problém v tom, že mají příliš obsáhlý stavový prostor a kapacita RAM paměti, kterou využívá Uppaal, není dostatečná, tudíž tyto strategie není možné nalézt dříve, než se zaplní paměť (pak program nahlásí chybu a hledání se ukončí). K získání takových výsledků je pak třeba drasticky zjednodušit model. V podkapitolách níže je u každého problému zmíněno, zda je u něj těžké získat výsledky. Také se zde soustředím na popis dalších možných rozšíření modelu či alternativních přístupů.

5.1 Nalezené problémy, strategie a rozšíření šachů

U šachů lze najít obrovské množství různých strategií na mnoho různých oblastí s jinými cíly. Jednou z nejzákladnějších strategií je porazit soupeře, což u tohoto modelu znamená sebrat nepřátelského krále (ne dát šach mat). Tato strategie se dá rozšířit, např. lze vyhrát do omezeného počtu kol (nebo do nějakého časového úseku) či pokusit se zničit krále pouze s určitými figurkami. Také je možné omezit ve strategiích pohyb nějakých figurek, což může být také zajímavé (např. zajistit, aby královna mohla pouze bránit, tzn. zakázat její pohyb na nepřátelskou stranu). Ukázky některých těchto strategií jsou níže:

```
strategy win = control: A<> player.Win
strategy qWin = control: A<> time <= 120 under win
strategy kWin = control: A<> whiteKnight(120).destroyedFigID == 250
strategy qDefend = control: A<> whiteQueen(140).x <= 4 under win
```

Se strategiemi se samozřejmě není třeba soustředit na vítězství. Je možné se soutředit na jiné figurky a jejich pohyb. Když se pěšec dostane na konec herní plochy, tak se může proměnit v dámu, tudíž je možné hledat takovou strategii, u kterých se jakýkoliv pěšec (nebo nějaký konkrétní) dostane na tato políčka. Můžeme se také soustředit na sebrání nějakých specifických figurek, jako třeba dámy, nebo najít strategii, která bude bránit naši vlastní dámu za každou cenu. Další zajímavou strategií by mohlo být vložení omezení počtu táhnutí s určitými figurkami, např. za celou hru bychom mohli s dámou táhnout pouze pětkrát (a následně s ní třeba něčeho dosáhnout, jako třeba sebrat krále). Ukázky těchto strategií jsou níže:

```
strategy pawnToTheEnd = control: A<> whitePawn(100).x == 7
strategy noQueenDestroyed = control: A<> whiteQueenDestroyed == false
strategy qDestroyed = control: A<> blackQueenDestroyed == true
strategy qTurns = control: A<> whiteQueen(140).turns <= 5 under qDestroyed
```

Bohužel se u tohoto modelu objevily při vyhodnocování strategií problémy s velkým stavovým prostorem, tudíž maximální velikost paměti, kterou Uppaal využívá, byla dosáhnuta dříve, než se strategie stihly vyhodnotit. Pro řešení tohoto problému jsem zkoušel nastavení agresivnější redukce stavového prostoru, poté omezení hraní pouze s nějakými figurkami (např. pouze pěšci nebo královny) a nakonec zmenšení samotého hracího pole a na něm hra pouze s pěšci, ovšem ani s těmito úpravami problém nebyl vyřešen.

Samotný model se dá ještě mnoha způsoby rozšířit. Bylo by možné zkoušet různá rozložení figurek na herním poli a různé stavy hry (což by vyžadovalo nějaké přepisování, ale model by tomu mohl být uzpůsoben). Nepřítel by mohl mít více funkcionalit, např. obtížnost. Na základě toho, jakou nepřítel obtížnost má, by měl různé pravděpodobnosti na špatné či dobré tahy. To, jestli je tah dobrý nebo špatný, by se muselo předem zjistit a vypočítat na základě nějaké vnitřní logiky (např. pokud by figurka sebrala hodnotnější figurku a zároveň by nevystavila nebezpečí samu sebe či hodnotnější figurky, tak by to byl dobrý pohyb, ovšem pokud by se královna vystavila nebezpečí vůči pěšci, tak by to byl špatný pohyb). Také by model mohl být více uzpůsoben úpravě specifických podmínek, jako např. přiřazení barvy, zvolení figurek, se kterými se bude hrát a podobně.

5.2 Strategie a rozšíření hanojských věží

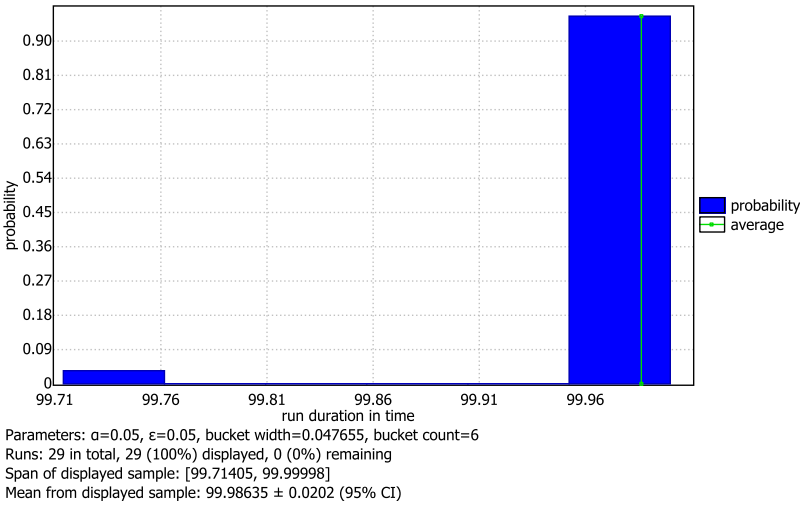
U hanojských věží je nejdůležitější najít strategii správného řešení, tzn. vložit disky na prostřední tyč v pořadí velikosti. Ke hledané strategii je třeba také přidat časové omezení. Řešení lze hledat postupně, jinak řečeno vytvořit strategii pro každý disk tak, aby se dostal na správnou pozici a další strategie stavět na těch předchozích. Řešení lze nalézt i v případě, pokud se hledá jako celek.

Strategie `largest` se soustředí na dosazení největšího disku na spodek prostřední tyče do 100 sekund. Obdobně jsou napsány další strategie, které určí, který disk by měl na jakou pozici zapadnout a mají zvýšený časový limit. Zakládají přitom na předešlých strategiích, aby byly zajištěny všechny disky na správných pozicích. Strategie `winQ` ukazuje možnost nalézt plné řešení. Strategie a pravděpodobnosti jsou ukázány níže:

```
strategy largest = control: A<> middle[3] == 4 && time <= 100
strategy large = control: A<> middle[2] == 3 && time <= 200 under largest
strategy small = control: A<> middle[1] == 2 && time <= 300 under large
strategy tiny = control: A<> middle[0] == 1 && time <= 400 under small
Pr[<=400] (<>middle[0] == 1) under tiny
strategy winQ = control: A<> player.done && time <= 400
Pr[<=400] (<>player.done) under winQ
```

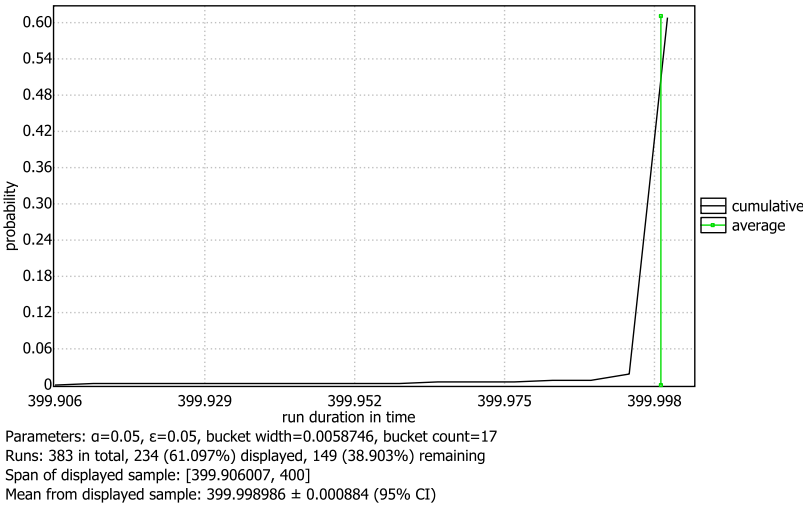
Pokud aplikujeme strategii `largest`, tak je s pravděpodobností větší než 90 % zajištěno, že se největší disk dostane na spodek prostřední tyče. Bez této strategie je pravděpodobnost něčeho takového velmi malá. Pravděpodobnost této strategie lze vidět v grafu rozdělení pravděpodobnosti na obrázku 5.1. Lze vidět, že pravděpodobnost největšího disku na spodu prostřední tyče je velmi vysoká v čase blížícím se 100 sekundám, jak bylo určeno strategií.

Na prvotní strategii `largest` lze navázat dalšími strategiemi, které řeší následující disky. Poslední strategie `tiny` by měla dosadit na nejvyšší pozici střední tyče nejmenší disk a tím



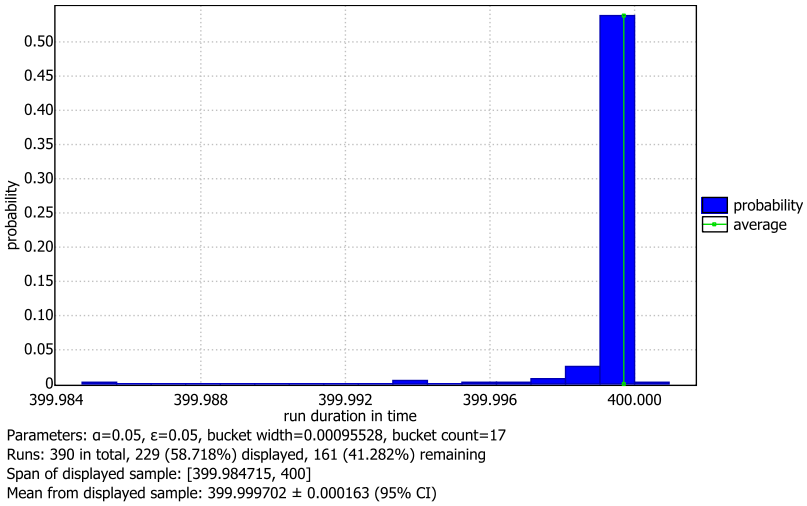
Obrázek 5.1: Obrázek ukazující rozdělení pravděpodobnosti strategie řídící dosazení největšího disku na spodek prostřední tyče do 100 sekund. Lze vidět, že byla úspěšně vyhodnocena.

najít úplné řešení. Tato strategie má úspěšnost okolo 60 %, podobně jako strategie `winQ`. Úspěšnost strategie `tiny` lze vidět na obr. 5.2, který ukazuje graf distribuční funkce, který téměř na 400 sekundách vyskočí na více než 60 %. Obdobně lze vidět na obr. 5.3 rozdělení pravděpodobnosti pro strategii `winQ`, které je velmi podobné. Obě strategie velmi zvyšují pravděpodobnost nalezení řešení oproti běhům s žádnými aplikovanými strategiemi, u nichž je pravděpodobnost nalezení řešení menší než 10 %.



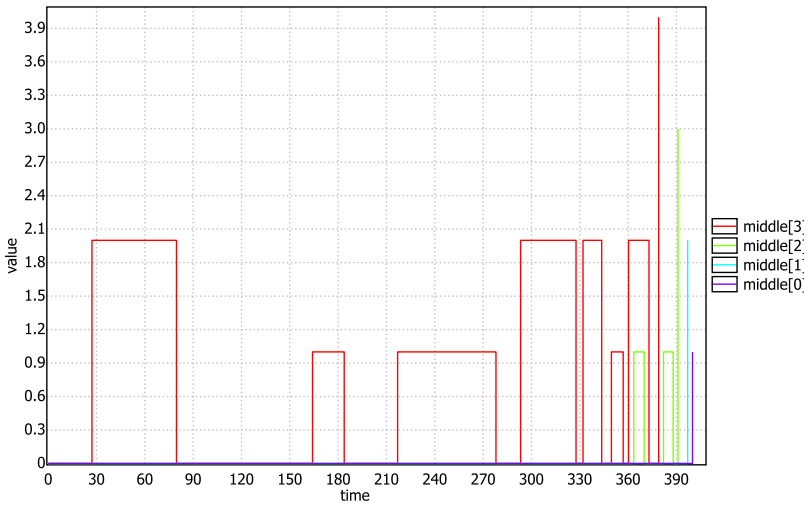
Obrázek 5.2: Obrázek znázorňující distribuční funkci pravděpodobnosti dosažení správného řešení hanojských věží. Pravděpodobnost se dostane až na více než 60 % v čase téměř 400 sekund.

Poslední obrázek 5.4 ukazuje, jak vypadá nalezené řešení v simulaci za použití strategie `tiny`. Lze vidět, že nejvíce aktivity bylo na nejspodnějším místě prostřední tyče a řešení se začínalo objevovat v čase 380 až 400 sekund. Ne vždy tohoto řešení můžeme dosáhnout, vzhledem k tomu, že pravděpodobnost jeho dosažení se pohybuje okolo 60 %, ale je to pod-



Obrázek 5.3: Obrázek znázorňující rozdělení pravděpodobnosti dosažení úspěšného řešení hanojských věží při použití strategie hledající celé řešení naráz. Pravděpodbonost se pohybuje kolem 60 % při dosažení času blízkému 400 sekund.

statné zlepšení oproti výsledkům bez strategií. Příkaz použitý pro vytvoření simulace je níže:
`simulate 1 [<=400] middle[3], middle[2], middle[1], middle[0] under tiny`

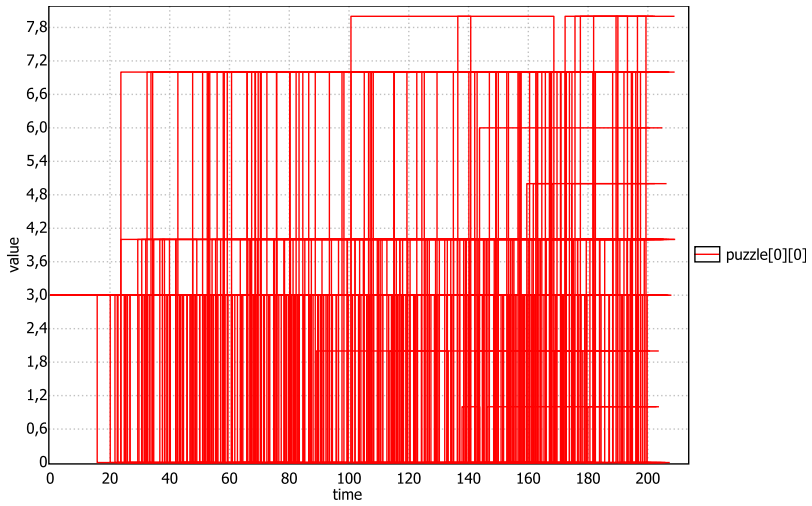


Obrázek 5.4: Obrázek znázorňující simulaci, která značí úspěch běhu hanojských věží pomocí strategie hledající správné řešení. Největší index prostřední tyče značí nejnižší pozici, nejmenší index značí nejvyšší pozici. Řešení bylo nalezeno do 400 sekund.

Další strategie, které by bylo možné hledat, by bylo například pokusit se dát na levou tyč disky 2 a 4 a na pravou tyč disky 1 a 3, k čemuž by se dalo opět přidat časové omezení. Bylo by možné i rozšířit původní strategii hledání řešení. Pokud bychom počítali u každého disku, kolikrát jím bylo pohnuto, bylo by možné hledat strategii správného řešení za předpokladu, že např. největším diskem pohneme jen jednou nebo u každého disku bychom měli nějaký omezený počet pohybů.

5.3 Strategie a rozšíření posuvného pole

U posuvného pole je důležitá především vítězná strategie, tudíž je třeba přemístit čísla tak, aby byla na správných pozicích. Tuto strategii můžeme hledat postupně, což znamená, že uděláme větší množství různých strategií zaměřující se pouze na jednu pozici a nalezení správného čísla. Na prvotní strategii se dá stavět poté následujícími strategiemi a pozicemi. Každá strategie má podstatně rychlejší vyhodnocení (než hledání celého řešení) a je možné tak dávat různé podmínky pro každé číslo, pokud by bylo zajímavé něco takového zkoumat. Časové omezení je také důležité.



Obrázek 5.5: Obrázek ukazující simulaci sto běhů posuvného pole. Sledujeme úplně první pozici, kde by měla ve správném řešení být hodnota 1. Na tomto místě se střídají všechny možné hodnoty.

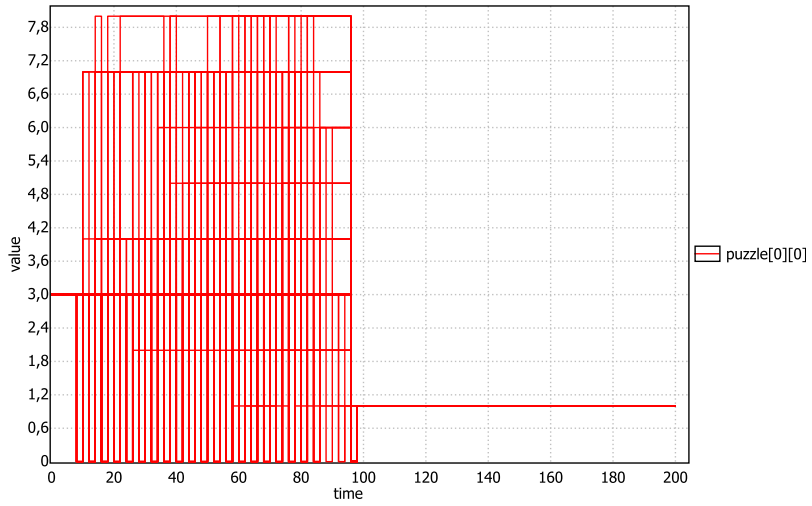
Ukázky strategií včetně hledání celého řešení jsou níže:

```
strategy one = control: A[] puzzle[0][0] == 1 || time <= 100
strategy two = control: A[] puzzle[0][1] == 2 || time <= 150 under one
strategy wholeSolution = control: A[] Player.solution || time <= 500
```

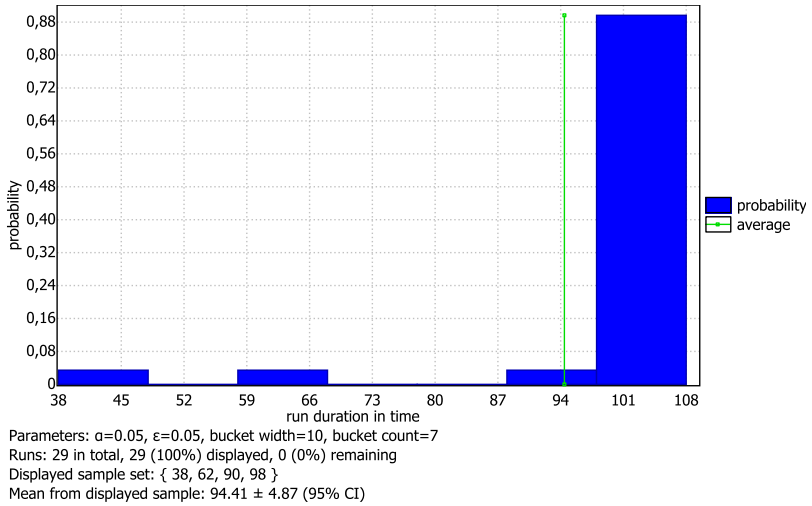
Na obr. 5.5 lze vidět simulaci 100 běhů bez jakékoliv aplikované strategie. Sledujeme první pozici posuvného pole, kde by měla být ve správném řešení hodnota 1. Lze vidět, že se na daném místě v různých bězích střídají všechna možná čísla a nemáme žádnou jistotu, že se tam skutečně objeví jednička. Simulace byla vytvořena pomocí příkazu:
`simulate 100 [<=200] puzzle[0][0]`

Na obr. 5.6 lze vidět simulaci 100 běhů vygenerovanou stejným příkazem pod strategií `one`. Tato strategie by nám měla zajistit, že na prvním místě bude do 100 časových jednotek správná hodnota. Ze simulace lze vidět, že v každém ze sto případů bylo nalezeno správné řešení včas a jednička se z daného místa nehne ani v nadcházejícím časovém úseku přesahující 100 časových jednotek.

Na obr. 5.7 lze vidět rozdělení pravděpodobnosti strategie `one`. Lze vidět, že jednička na první pozici posuvného pole je s největší pravděpodobností v časovém rozmezí 98 až 108 časových jednotek, z čehož lze vidět, že strategie je plně funkční a dá se na ní úspěšně stavět.



Obrázek 5.6: Obrázek ukazující simulaci sto běhů posuvného pole s použitou strategií. Sledujeme úplně první pozici, kde by měla ve správném řešení být hodnota 1. Lze vidět, že u všech běhů se na políčko dostala hodnota 1.



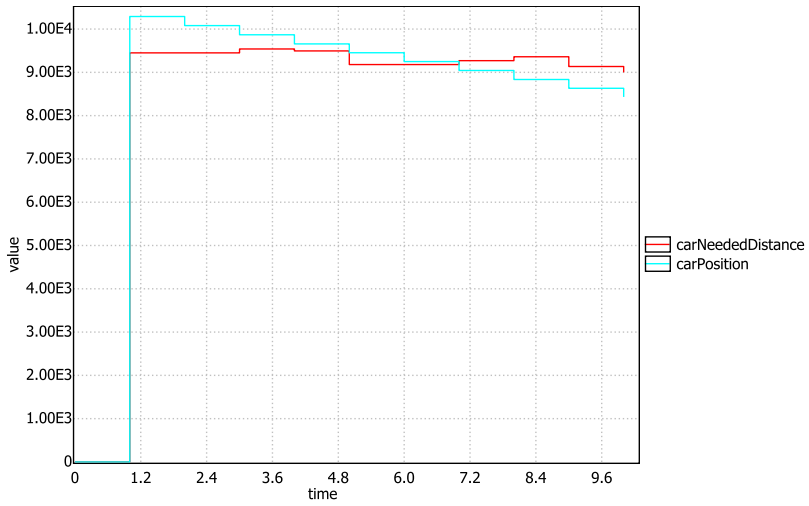
Obrázek 5.7: Obrázek ukazující rozložení pravděpodobnosti toho, v jakém čase se na první pozici posuvného pole objeví jednička. Lze vidět s velmi vysokou pravděpodobností, že se tam objeví kolem 100 časových jednotek, tudíž strategie funguje.

Další rozšíření by mohla být podpora n-rozměrných polí, kde by se mohl zadat jakýkoliv rozměr a model by mohl hledat správná řešení. Bylo by to pouze časově náročnější a rozměry by byly omezeny, jelikož máme omezenou paměť. Také by na různých pozicích bylo možné zamknout čísla (se správnými hodnotami) a sledovat, zda je možné zbytek hodnot přesunout na správná místa (ve stejném čase jako nezamknuté hodnoty). To by nám pomohlo zjistit, která políčka jsou důležitá a hodně využívaná.

5.4 Nalezené problémy, strategie a rozšíření kinematického problému

Zde nás především zajímá to, aby balíček a auto byly v takové poloze, aby auto bylo schopno chytit balíček, jakmile dopadne na zem. Strategie pro tuto podmínku je nejspíše nejdůležitější v modelu. Chceme tedy, aby se rovnala pozice auta (`carPosition`) s vypočítanou hodnotou (`carNeededDistance`), která říká, jak daleko by auto mělo být (nebo i větší rozsah hodnot díky šířce auta a menší nepřesnosti). Ukázka toho, jak tyto dvě proměnné spolu běží a souvisí je na obr. 5.8, který ukazuje v simulaci stále se zmenšující pozici auta vůči letadlu a vypočítanou potřebnou pozici auta. V jednom bodě jsou si velmi blízko a pak se začnou vzdalovat, až v nějakém bodě už můžeme běh ukončit, jelikož auto by nemělo šanci dohnat letadlo. Příkaz, který byl použit pro vytvoření simulace:

```
simulate 1 [<=10] carPosition, carNeededDistance
```



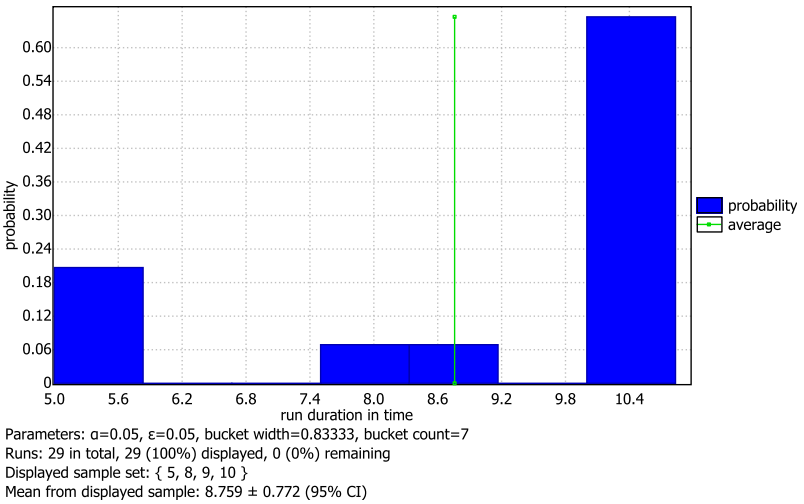
Obrázek 5.8: Obrázek ukazující simulaci pozice auta a hledané pozice auta. Lze vidět, že se sami sobě nesnaží blížit a jakmile se téměř střetnou, tak se začnou více oddalovat.

Strategie by nám měla vztah těchto dvou pozic stabilizovat a měli by být stejné nebo se k sobě blížit většinu času. Také by bylo dobré, aby po splnění této podmínky bylo auto v poloze, kdy vypustí balíček (`plane.DropPackage`) a aby to vše bylo splněno do nějakého časového limitu (případně hledat minimální čas, za který je toho možné dosáhnout). Ukázky strategií jsou níže:

```
strategy carPos = control:A<>carPosition == carNeededDistance && time <= 15
strategy planeHigh = control: A<> planeAltitude => 10000 under carPos
strategy packageDrop = control: A<> Plane.DropPackage under carPos
```

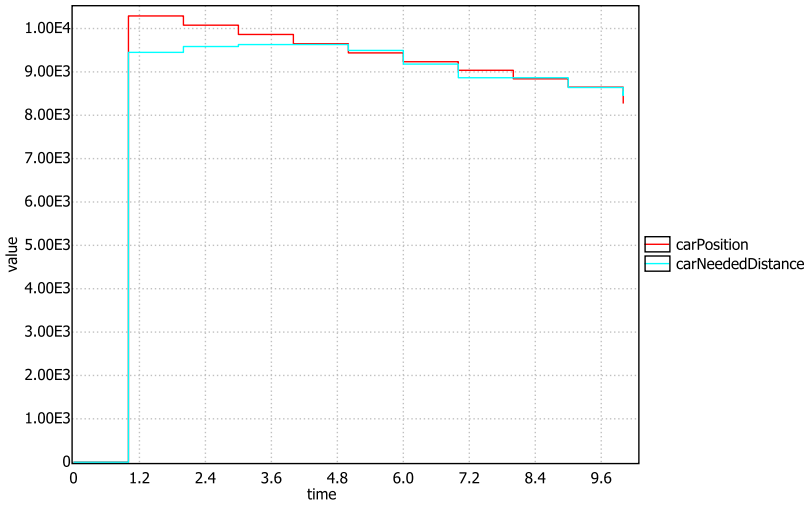
Bohužel se i u tohoto modelu objevily problémy s pamětí a bylo třeba pro získání výsledků ze strategií model omezit. Při následujících výsledcích bylo letadlu zakázáno klesat a stoupat a rozmezí, ve kterých se rychlosti mohly pohybovat, byly menší. Také se používala strategie `carPos`, která hledá pouze rovnost pozice auta a pozice, kde by auto mělo být (tzn. neřeší, zda je letadlo ve stavu shození balíčku). Ukázalo se, že s omezenými podmínkami strategie byla úspěšně vyhodnocená. Na obr. 5.9 je rozdělení pravděpodobnosti správné pozice auta. Lze vidět, že v 10 sekundách je pravděpodobnost největší, ale lze tohoto stavu

dosáhnout i v 5,8 či 9 sekundách. Bez použité strategie je pravděpodobnost něčeho takového menší než 10 %.



Obrázek 5.9: Obrázek ukazující rozdělení pravděpodobnosti správné pozice auta s použitou strategií. Nejvíce je auto na správné pozici v čase pohybujícím se okolo 10 sekund, ale i v čase okolo 5, 8 a 9 sekund.

Na obr. 5.10 lze pak vidět simulaci běhu pod strategií ukazující polohu auta a potřebnou polohu auta. Simulace byla vytvořena stejně jako simulace na obr. 5.8 pod strategií `carPos`. Lze zde vidět snahu o nalezení správné pozice auta, kde se aktuální poloha auta snaží přiblížit správné pozici a dosáhne toho přibližně v 9 sekundě.



Obrázek 5.10: Obrázek ukazující simulaci pozice auta a hledané pozice auta s použitou strategií. Lze vidět, že se sami sobě snaží blížit a strategie úspěšně najde řešení do 10 sekund.

Model i samotný návrh lze rozšířit o možnost kontrolovat samotný balíček, zatímco padá na zem (např. by balíček držel dron nebo by měl nějakou jinou možnost ovlivňování své dráhy), tudíž by mohl měnit svou rychlost a případně i polohu v horizontálním směru,

čemuž by se auto mohlo přizpůsobovat. Také by bylo možné na začátku všechny počáteční hodnoty zvolit náhodně (v reálném rozmezí) a sledovat, zda se nějak zásadně mění čas, za který lze dosáhnout správného výsledku.

Kapitola 6

Závěr

Cílem této práce bylo seznámit se s nástroji Uppaal, zvolit vhodné problémy k řešení z různých oblastí a následně vytvořit jejich modely pomocí nástroje Uppaal Stratego a ověřit schopnosti hledání řídicích strategií tohoto nástroje. Ke splnění zadání byly vybrány logické hry šachů, posuvného pole, hanojských věží a kinematický problém s letadlem nesoucí balíček a autem, které balíček má chytit. Jedna z nejdůležitějších částí pro splnění zadání bylo seznámení se a pochopení nástrojů Uppaal, zejména Stratego a SMC. Toho bylo dosaženo pročítáním relevantních dokumentací a testováním primitivních modelů. Jako první oblast pro řešení byly zvoleny šachy.

Implementace modelu šachů byla trochu více problematická kvůli omezené syntaxi v programovací části nástroje Uppaal, ale neobjevil se žádný větší problém. Závažný problém přišel až v hledání strategií, které jsem si nechal na konec, kde se ukázal problém ve velikosti stavového prostoru a počítače, na kterých jsem pracoval, nebyly schopny nalézt řešení dostatečně rychle než došla maximální hodnota paměti na RAM, kterou Uppaal využívá. Bylo by lepší, kdybych na tento problém narazil už na začátku, abych s touto znalostí vytvářel model. Po tomto byly k řešení vybrány dvě jednoduché hry, hanojské věže a posuvné pole. U nich se už tyto problémy neobjevily a strategie šly úspěšně nalézt, kde u posuvného pole strategie zvýšily drasticky šance úspěšného řešení a hanojských věží pouze částečně. Nakonec byl zvolen i kinematický problém, ovšem u toho se také objevil problém s vyčerpáním paměti. Pokusy o vyřešení těchto problémů byly především hledání strategií s agresivnější redukcí stavového prostoru a zjednodušování/omezování modelů, což se povedlo alespoň u kinematického problému, kde bylo možné vyhodnotit strategie, ovšem u šachů už ne.

Práce mě především naučila, jak důležitá je kontrola všech důležitých částí v brzké fázi projektu. Jakmile se objeví chyba v konečné fázi a implementace je hotová, tak je problematické celou práci předělávat. Také bylo třeba změnit svůj zakořeněný pohled na programování a přistoupit k implementaci s jiným přístupem. S nástrojem samotným se pracovalo pohodlně a jednoduše. U nejnovější verze byl problém s ukazováním konkrétních chyb, tudíž bylo náročné něco opravovat, z toho důvodu jsem implementaci prováděl ve verzi 5, která už ukazovala přesné chyby i konkrétní místa, kde se vyskytují. Ve vyhodnocování strategií ovšem byla nejúčinnější a nejrychlejší verze 7.

Každý specifický model by bylo možné rozšířit a nápady na rozšíření jsou popsány u každého modelu v kapitole 5. V práci by bylo možné pokračovat řešením dalších problémů z jiných oblastí, zejména oblasti fyzikálních systémů se spojitým časem, ve kterých by se využívalo hybridních hodin.

Literatura

[1] Behrmann, G., David, A. a Larsen, K. G. A Tutorial on Uppaal. In: Bernardo, M. a Corradini, F., ed. *Formal Methods for the Design of Real-Time Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 200–236. ISBN 978-3-540-30080-9. Dostupné z: https://doi.org/10.1007/978-3-540-30080-9_7.

[2] Bengtsson, J. a Yi, W. Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W. a Rozenberg, G., ed. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 87–124. ISBN 978-3-540-27755-2. Dostupné z: https://doi.org/10.1007/978-3-540-27755-2_3.

[3] Bourke, T. *Modelling and programming embedded controllers with timed automata and synchronous languages*. Sydney, 2009. Ph. D. Thesis. University of New South Wales. Dostupné z: https://www.tbrk.org/papers/tbourke-phd.pdf.

[4] Bouyer, P. a Laroussinie, F. Model Checking Timed Automata. In: *Modeling and Verification of Real-Time Systems*. John Wiley & Sons, Ltd, January 2008, kap. 4, s. 111–140. ISBN 9780470611012. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470611012.ch4.

[5] David, A., Jensen, P. G., Larsen, K. G., Legay, A., Lime, D. et al. On Time with Minimal Expected Cost! In: Cassez, F. a Raskin, J.-F., ed. *Automated Technology for Verification and Analysis*. Springer International Publishing, 2014, s. 129–145. ISBN 978-3-319-11936-6. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-319-11936-6_10.

[6] David, A., Jensen, P. G., Larsen, K. G., Mikučionis, M. a Taankvist, J. H. Uppaal Stratego. In: Baier, C. a Tinelli, C., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, s. 206–211. ISBN 978-3-662-46681-0. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-662-46681-0_16.

[7] David, A., Larsen, K., Legay, A., Mikučionis, M. a Poulsen, D. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*. Leden 2015, sv. 17. Dostupné z: https://www.researchgate.net/publication/270596354_Uppaal_SMC_tutorial.

[8] DeCross, M. a O'Brien, T. *1D Kinematics Problem Solving* [online]. Brilliant.org [cit. 14.3.2020]. Dostupné z: https://brilliant.org/wiki/velocity-and-acceleration-problem-solving-easy/.

[9] Huth, M. a Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2. vyd. Cambridge University Press, 2004. ISBN 978-0-521-54310-1.

[10] Křena, B. a Vojnar, T. Automated formal analysis and verification: an overview. *International Journal of General Systems*. 2013, sv. 2013, č. 42, s. 335–365. ISSN 0308-1079. Dostupné z: https://www.fit.vut.cz/research/publication/10284.

[11] Larsen, K. G., Mikučionis, M. a Taankvist, J. H. Safe and Optimal Adaptive Cruise Control. In: Meyer, R., Platzer, A. a Wehrheim, H., ed. *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015, Proceedings*. Cham: Springer International Publishing, 2015, s. 260–277. ISBN 978-3-319-23506-6. Dostupné z: https://doi.org/10.1007/978-3-319-23506-6_17.

[12] Mikučionis, M. *Safe and Optimal Cruise Control* [online]. Uppaal Stratego, Poslední změna 16.9.2015 [cit. 14.1.2020]. Dostupné z: https://people.cs.aau.dk/~marius/stratego/cruise.html.

[13] Mikučionis, M. *Introduction* [online]. Uppaal Stratego, Poslední změna 31.3.2015 [cit. 14.1.2020]. Dostupné z: https://people.cs.aau.dk/~marius/stratego/intro.html.

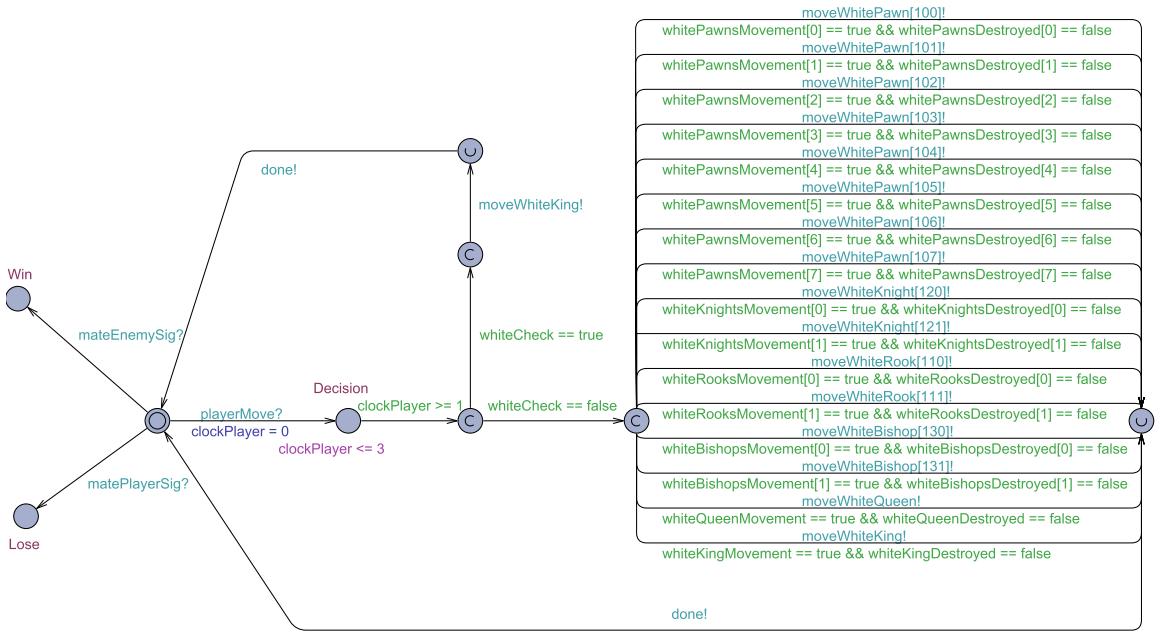
[14] Pelánek, R. *Formal Verification, Model Checking* [online]. 20. května 2011. [cit. 16.12.2019]. Dostupné z: https://www.fi.muni.cz/~xpelanek/IA158/slides/verification.pdf.

[15] Pelánek, R. *Reduction and Abstraction Techniques for Model Checking*. Brno, 2006. Doctoral theses, Dissertations. Masaryk University, Faculty of Informatics, Brno. Vedoucí práce RNDr. Ivana Černá, C. prof.

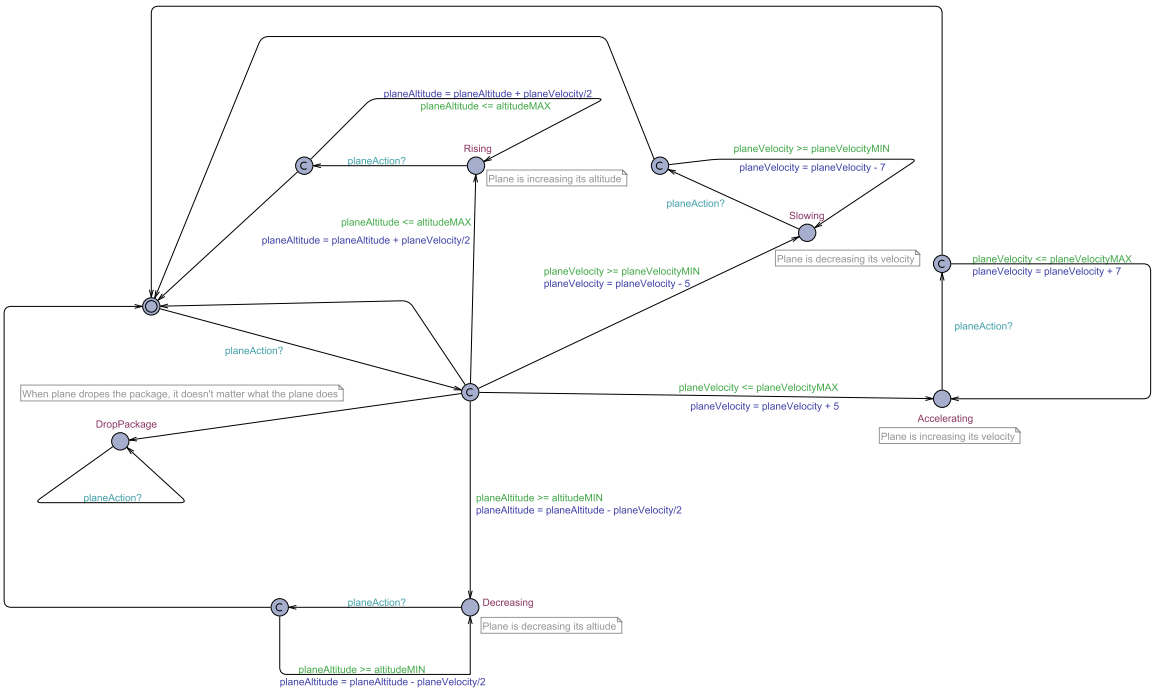
[16] Pettersson, P. *Introduction* [online]. Poslední změna 23.10.2019 [cit. 16.1.2020]. Dostupné z: http://www.uppaal.org/.

Příloha A

Šablony



Obrázek A.1: Obrázek znázorňující šablonu hráče. Zde hráč může po rozhodnutí hýbat s jakoukoliv figurkou nebo se dostat do konečných stavů. Nepřítel vypadá téměř totožně, akorát jej nelze ovládat.



Obrázek A.2: Obrázek šablony letadla. Letadlo musí být aktivováno pomocí kanálu `planeAction`. Může vzlétnout, snížit se, zrychlit, zpomalit či shodit balíček. V těchto stavech můžou zůstat nebo zrušit akce.