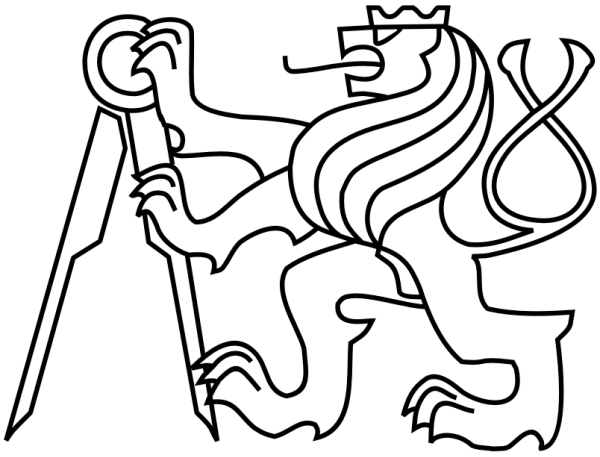
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ

KATEDRA ŘÍDICÍ TECHNIKY



DIPLOMOVÁ PRÁCE

Stavové automaty v C++ pro robotické aplikace

Autor: Bc. Petr Šilhavík

Vedoucí práce: Ing. Michal Sojka, Ph.D.

Praha, 2013

Prohlášení

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

V Praze, dne 10.5.2013

Petr Šilhánek

podpis

Poděkování

Chtěl bych poděkovat zejména vedoucímu Ing. Michalu Sojkovi, Ph.D. za jeho vstřícnost a vždy podnětné připomínky a rady. Dále bych chtěl ostatním členům robotického týmu Flamingos za dobrou spolupráci během úprav řídicího SW. A v neposlední řade bych chtěl poděkovat svým rodičům a nejbližším za jejich podporu při vypracování této diplomové práce.

Abstrakt

Tato práce se zabývá aplikací stavových automatů pro řízení robotů. V rámci práce bylo provedeno doplnění knihovny Boost Statechart o časovače a poté využití této knihovny pro řízení robotů v robotickém týmu Flamingos. Knihovna Boost Statechart byla vybrána jako náhrada za původní vlastní knihovnu. V souvislosti se změnou programovacího jazyku knihovny z C na C++ byla také přepsána do C++ část řídicího SW. Další část tvoří doplnění nástroje pro vizualizaci stavových automatů vytvořených za pomoci knihovny Boost Statechart. Veškeré provedené úpravy souvisí s analýzou zdrojového kódu a hledání možných chyb. Program tedy neprovádí pouze vizualizaci, ale také pomáhá programátorovi hledat možné logické chyby v implementaci.

Abstract

This thesis deals with application of state machines in robot control. As part of work timers were added to Boost Statechart Library and then this library was used for robotic control by robotic team Flamingos. The Boost Statechart library was chosen as replacement of the original non-standard library. Due to the change of the programming language of library from C to C++ also part of other control software was rewritten to C++. The another part is focused on state machine visualizer. New features were added to visualizer. All of these features concentrate on finding errors in source code of state machine, especially errors in implementation.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Petr Šilhavík**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Stavové automaty v C++ pro robotické aplikace**

Pokyny pro vypracování:

1. Seznamte se se softwarem robotu Flamingos a knihovnou Boost/Statechart.
2. Rozšiřte knihovnu Boost/Statechart o podporu časovačů.
3. Převeďte řídicí software robotu Flamingos do C++ s využitím rozšířené Boost knihovny.
4. Integrujte sestavovací proces řídicího softwaru s nástrojem Boost Statechart Viewer a rozšiřte tento nástroj o další potřebné vlastnosti.
5. Výsledek otestuje na reálném robotu a v reálném prostředí a zdokumentujte ho.

Seznam odborné literatury:

David Vandevoorder and Nicolai M. Josuttis: "C++ Templates: The Complete Guide"
https://rtime.felk.cvut.cz/dragons/doc/
http://www.boost.org/doc/libs/1_52_0/libs/statechart/doc/index.html

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2013/2014

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 18. 12. 2012

Obsah

Seznam obrázků

1 Úvod

Práce se zabývá aplikací stavových automatů implementovaných pomocí programovacího jazyku C++ v robotice.

Se stavovými automaty se můžeme setkat téměř v celé oblasti lidské činnosti. Jelikož jsou stavové automaty modelem výpočetních procesů, tak se mezi jejich první aplikaci řadí zpracování regulárních výrazů, což jsou řetězce, které popisují celou množinu řetězců. Další oblastí pro jejich aplikaci je popis průmyslových procesů [DP04]. V neposlední řadě je důležité zmínit i robotiku, kde se stavové automaty používají pro řízení robotů kvůli jejich názornosti [Sal+10].

Cílem práce je rozšířit schopnosti knihovny stavových automatů. Jedná se o možnost použití více časovačů v rámci stavu i stavového automatu a zjednodušit seznámení se zdrojovými kódy pro nové členy. Pro splnění této podmínky bylo rozhodnuto o kompletní výměně knihovny. Z původní, která je napsána v jazyce C, byl kód převeden na novou knihovnu Boost Statechart, která již využívá jazyk C++. Samotná knihovna Boost Statechart neobsahuje časovače vůbec, takže bylo nutno do knihovny časovače doplnit. Ke knihovně Boost Statechart existuje nástroj pro vizualizaci, který byl vytvořen v rámci mé bakalářské práce. V souvislosti s diplomovou prací byly provedeny změny i v tomto nástroji.

Motivací pro vytvoření této práce je zejména snaha o používání standardních knihoven při řízení robotů v týmu Flamingos. Původní knihovna byla sice vytvořena speciálně pro řízení robotů, ale nejedná se o standardní knihovnu, kterou by používali i jiní vývojáři. Ačkoliv se v případě výsledku této práce také jedná o upravenou knihovnu, tak jádro knihovny zůstalo zachované a všechny provedené úpravy jsou plně kompatibilní s ostatními částmi knihovny.

Dalším cílem, který již nesouvisí se změnou knihovny, je snaha o zjednodušení zdrojových kódů, které patří do řídicí části robotu. Zjednodušení se dosahuje pomocí C++, kde se využívá tříd a struktur a v nich sdílených proměnných, díky kterým lze částečně omezit paměťovou náročnost. Vytvořením tříd dojde rovněž k zpřehlednění celého kódu. Části, které spolu souvisí, budou sloučeny do jedné třídy a nebudou rozděleny do jiných souborů.

V případě doplnění nástroje na vizualizaci stavových automatů je cílem zlepšit zpětnou vazbu pro programátora. Kromě obyčejného generování popisu stavového automatu bude programátor dostávat zpětnou vazbu o chybách, které se v automatu nacházejí. I přes tyto chyby může být automat kompilovatelný a na chyby se přijde například pouze testováním nebo zkoumáním stavového diagramu. Pro programátora je ale v tomto případě mnohem užitečnější

nalézt tyto informace hned při výpisu kompilátoru.

Pro volbu funkcí přidaných do nástroje na vizualizaci byly prozkoumány ostatní nástroje, které se starají o vizualizaci stavových automatů. Prozkoumané nástroje se o kontrole logiky ve zdrojových kódech přímo nezmiňují. Specifikace pro doplnění funkcí v této oblasti byly konzultovány i s vedoucím práce.

Celá práce se dá rozdělit do tří hlavních směrů, kterými jsou úprava a nahrazení knihovny, změny ve struktuře řídicího SW a doplnění funkcí do nástroje pro vizualizaci. Z tohoto důvodu jsou tyto části cíleně odděleny i v jednotlivých sekcích, které se věnují návrhu a implementaci či testování.

Veškeré UML diagramy tříd, které jsou zobrazeny v této práci, byly vytvořeny v programu dia-gnome. Popis značek v UML diagramech je k dispozici v [Ně10]. V případě, kdy je použita čárkovaná šipka, která znamená závislost, tak je vždy tato závislost popsána v obrázku.

V sekci 2 jsou popsány základní termíny, se kterými se lze setkat při studiu stavových automatů. Kromě teoretických základů ke stavovým automatům se tato sekce věnuje stavovým diagramům, které slouží k vizualizaci chování stavových automatů. V této sekci je popsán vizualizační nástroj pro knihovnu Boost Statechart, ale také jiné nástroje, které se vizualizaci stavových automatů rovněž věnují. V další sekci 3 je krátce představen tým Flamingos včetně jeho robotů a jejich řídicího SW. V sekci 4 jsou představeny a následně i porovnány obě knihovny stavových automatů a to původní knihovna FSM a nová Boost Statechart. Následující sekce 5 se věnuje návrhu a implementaci řešení pro všechny úpravy. Poté již následuje sekce 6, která se věnuje testování. Práci uzavírám sekcí 7, kde shrnuji dosažené výsledky.

2 Stavové automaty

V této kapitole je představen pojem stavového automatu. Nejprve je v sekci 2.1 představen samotný pojem stavového automatu a poté v sekci 2.2 jsou popsány dva základní přístupy k hierarchicky umístěným stavům.

Po úvodní části, která se věnuje stavovým automatům, se další část věnuje jejich reprezentaci 2.3.

2.1 Konečné stavové automaty

Aby bylo jasné, co tento pojem znamená, je vhodné zmínit, že slovo automat pochází z řeckého slova *automatos*, což znamená samohybný. Tento význam si toto slovo zachovalo dodnes [Dic13]. Veškeré dále uvedené informace v této části jsou citovány z [Kr06] kapitola 2.

Konečné stavové automaty jsou matematickým modelem pro výpočetní procesy. Aby se automat dal zařadit do této skupiny, je nutné, aby měl konečný počet stavů.

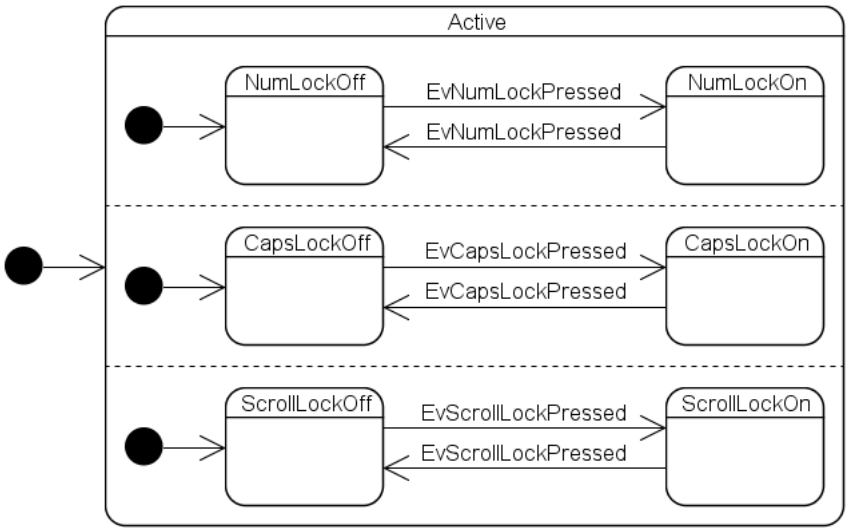
Definice: Konečný automat (Finite Automaton, FA) M je pětice (Q, Σ, δ, q_0, F), kde:

- Q je neprázdná konečná množina stavů.
- Σ je konečná množina vstupních symbol, nazývaná také vstupní abeceda.
- $\delta : \quad Q\,x\,\Sigma \rightarrow Q$ je parciální přechodová funkce.
- $q_0 \in Q$ je počáteční stav.
- $F \subseteq Q$ je množina koncových stavů.

Při použití v robotice se za stavový automat považuje pouze čtveřice. Vynechána je množina koncových stavů. S touto množinou se můžeme setkat v jiných oblastech jejich použití, například při analýze regulárních výrazů [She11].

V případě stavových automatů mohou existovat tzv. ortogonální stavy někdy také nazývané složené [Dou99]. Takový stav v sobě obsahuje několik stavových automatů, které běží paralelně a tím pádem nezávisle. Jakmile stavový automat přejde do ortogonálního stavu, začne chovat jako více paralelně běžících automatů. Jakmile přijde do ortogonálního stavu událost, tak se předá do všech paralelně běžících stavových automatů. Na obrázku 2.1 je ukázka UML diagramu ortogonálního stavu ve stavovém automatu. Stavový automat reprezentuje chování

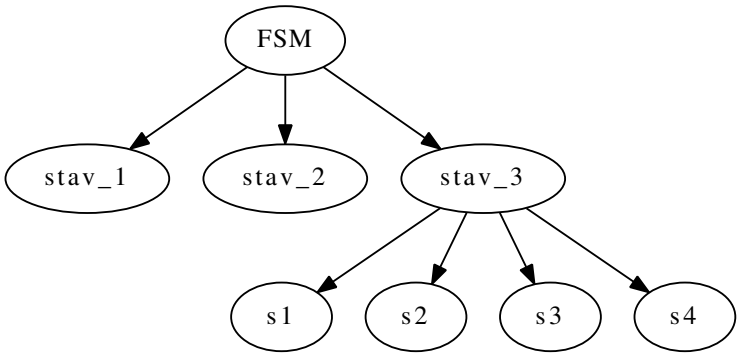
LED diod, které jsou běžně na klávesnici. Jakmile přijde do stavu Active událost, tak se předá do všech tří automatů (NumLock, CapsLock i ScrollLock).



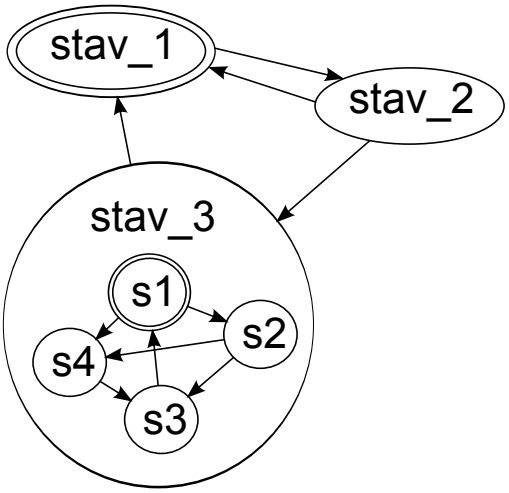
Obrázek 2.1: UML diagram automatu s ortogonálním stavem, převzato z [HD07]

2.2 Stavové automaty s hierarchickými stavy

Hierarchický stavový automat a subautomat jsou dva základní koncepty, pokud potřebujeme vnořit více stavů do sebe. Jejich principy se sice překrývají, ale jejich vzájemné použití se prakticky vylučuje. Oba koncepty využívají hierarchického uspořádání stavů, ale liší se v přístupu ke stavům v jednotlivých vrstvách. Na obrázku 2.2 je strom hierarchických stavů ve stavovém automatu. Stavy stav_1, stav_2 a stav_3 leží v jedné vrstvě. To samé platí pro stavy s1, s2, s3 a s4. Ačkoli jejich stavové diagramy vypadají stejně, ale automaty se chovají jinak. Ukázka stavového diagramu pro automat s hierarchickými stavy je na obrázku 2.3. Strom i stavový diagram odpovídají stejnému stavovému automatu. Informace o problematice hierarchických stavových automatů a porovnání dvou různých přístupů lze nalézt také v [Cha07].



Obrázek 2.2: Strom stavů v hierarchickém stavovém automatu



Obrázek 2.3: Stavový automat s hierarchicky umístěnými stavy

2.2.1 Hierarchický stavový automat

V tomto případě se obvykle využívá možnosti sdílení přechodů mezi stavy. Každá událost, která přijde do automatu, se zpracovává od nejnižší vrstvy po nejvyšší. V případě, že se událost nezpracuje v dané vrstvě, odesílá se do vrstvy vyšší. Pokud se událost zpracuje, tak se samozřejmě neodesílá výše. Často se zde také mluví o tzv. kontextu stavu, což představuje stav, který leží v hierarchii výše. V praxi to znamená, že takový automat má obvykle více aktivních stavů. V případě, že by v automatu byl aktivní například stav s3, tak současně s ním by byl aktivní stav_3.

Úlohy prováděné tímto automatem lze dekomponovat. Tím, že se události předávají do vyšší vrstvy, tak je možno sdílet přechod mezi stavy pro celou skupinu stavů. Tím se ušetří délka kódu. Nevýhodou je to, že při úpravě části úlohy je třeba si ohlídat celý automat. Naopak se zde velmi jednoduše hlídá časové omezení u jednotlivých částí úlohy.

2.2.2 Subautomat

Na rozdíl od hierarchického stavového automatu má subautomat vždy pouze jeden aktivní stav. Bez ohledu na to kolik subautomatů má v sobě vnořeno. Událost se zpracovává v aktivním stavu. Jediná událost, která se předá do vyšší vrstvy, ale vždy pouze o 1 vyšší a zde se také ihned zpracuje, je informace o ukončení subautomatu.

Takový automat je vhodné použít tam, kde lze úlohu, kterou má vykonávat, dekomponovat na jednoduché úlohy. Pokud poté potřebujeme upravit jednotlivou úlohu, není se třeba starat o zbytek automatu. V praxi není možné například hlídat časové omezení u jednotlivých částí úlohy. Z důvodu pouze jediného aktivního stavu.

2.3 Reprezentace stavových automatů

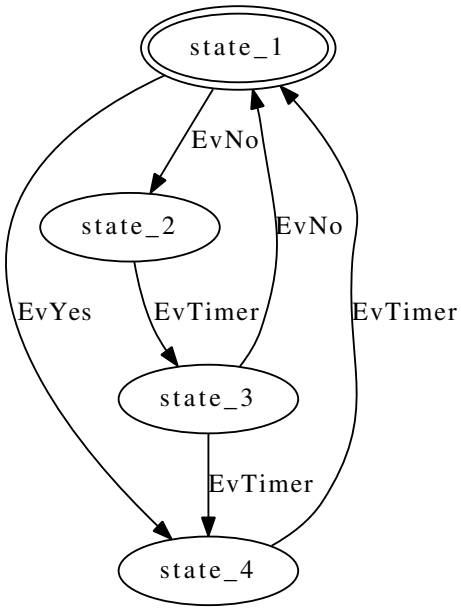
Tato sekce se věnuje grafické reprezentaci stavových automatů. Do této skupiny patří stavový diagram a stavový strom. Obě metody jsou prakticky stejné a proto je tato část věnována pouze stavovému diagramu.

Poté jsou představeny dva základní přístupy k vizualizaci. Na závěr jsou ještě uvedeny nástroje pro vizualizaci stavových automatů pro porovnání toho, co jednotlivé programy dokážou.

2.3.1 Stavový diagram

Pro zobrazování struktur stavových automatů se nejčastěji používá stavový diagram. Stavový diagram je grafický nástroj pro zobrazení struktury stavového automatu. V diagramu se zobrazují všechny stavy (včetně jejich hierarchického uspořádání). Kromě stavů se zobrazují přechody mezi stavy ve formě orientovaných šipek. Ze stavového diagramu jsou tedy patrné všechny stavy, ve kterých se automat může nacházet, a přechody mezi stavy inicializované událostmi. Počáteční stav je obvykle vyznačen dvojitým ohraničením. U šipek, které značí přechody mezi stavy, jsou obvykle popisy, které specifikují událost, při které se tento přechod provede. Tato metoda je velmi dobrou kontrolou správnosti stavových automatů. Jedním z důvodů je to, že programátor obvykle navrhuje stavové automaty právě ve formě stavového diagramu. Oproti stavovým stromům se dokáže vypořádat i s hierarchicky uspořádanými stavy.

Na obrázku 2.4 je pro ilustraci ukázka stavového diagramu.



Obrázek 2.4: Stavový diagram

2.3.2 Přístupy k vizualizaci

K vizualizaci stavových automatů existuje více přístupů. První přístup je zpětné generování stavového diagramu nebo jiné reprezentace stavového automatu. Tento přístup využívá náš nástroj pro vizualizaci.

Druhý přístup využívá grafického návrhu stavového automatu. Takový nástroj má grafické rozhraní, kde si lze stavový diagram nakreslit. Ze stavového diagramu je poté vygenerována kostra stavového automatu, do které si zbývající potřebné funkce doplní programátor. Tento přístup je pro vizualizační nástroje běžnější.

2.3.3 Boost Statechart viewer

Nástroj na vizualizaci stavových automatů napsaných v knihovně Boost Statechart byl vytvořen pro ulehčení práce programátora, který má díky němu jednoduchou zpětnou kontrolu nad vytvořeným stavovým automatem. Informace uvedené v této části jsou čerpány z mé bakalářské práce [Ši11] a současného stavu zdrojového kódu.

Stavový diagram je vytvořen s pomocí knihovny graphviz, resp. pomocí programu dot. Vstupem programu je samotný zdrojový kód a výstup programu tvoří zdrojový kód, který lze vizualizovat v programu dot a vytvořit tak stavový diagram. S pomocí stavového diagramu si lze zkontrolovat správně definované jednotlivé stavy a přechody mezi nimi.

K analýze zdrojového kódu se používají knihovny Clang a LLVM, jejichž dokumentace je k dispozici v [Lr13]. Samotná analýza je prováděna pomocí Abstract Syntax Tree (AST). V rámci AST jsou vyhledávány třídy, a pokud se jedná o stavy, automaty nebo události, jsou analyzovány parametry a v případě stavů i vnitřní obsah tříd. Program dokáže rovněž zobrazit i standardní diagnostiku, kterou provádí kompilátor.

Díky využití AST lze jednoduše nalézt třídy či struktury. Klasifikace jednotlivých struktur a tříd je poté prováděna pomocí porovnávání jejich předků. Názvy předků jsou porovnávány pomocí textových řetězců včetně namespace, ve kterých tyto třídy leží.

Přechod na plugin přinesl i výhody. Jednou z nich je omezení závislosti na verzi LLVM a Clang. Vývojáři bohužel průběžně odstraňují a upravují parametry jednotlivých funkcí. Převodem na plugin se odstranila závislost na funkcích, které zpracovávaly parametry příkazové řádky. Zjednodušil se tím tedy přechod mezi verzemi knihoven.

Program byl vyvíjen s ohledem na podobné projekty, které existují pro jiné knihovny stavových automatů. Další projekty, které se věnují vizualizaci stavových automatů jsou popsány v sekcích 2.3.4 až 2.3.7.

Program dokáže vizualizovat stavové automaty včetně hierarchického umístění stavů. Mezi stavy nakreslí šipky, které označují přechody. Veškeré přechody jsou označeny popisem, který označuje událost, na kterou se daný přechod vykoná. Dokáže rovněž rozpoznat počáteční stav

a ten do stavového digramu označit, standardně dvojitě orámovaným stavem. S těmito schopnostmi lze vizualizovat prakticky většinu stavových automatů vytvořených v knihovně Boost Statechart. Jediným větším omezením je v současné době to, že program dokáže současně vizualizovat pouze jeden stavový automat. Není tedy možné si zobrazit několik stavových automatů v rámci jednoho průchodu programem. Souvisí to s tím, že programátor stejně většinou důsledně odděluje od sebe stavové automaty, protože se snaží o co nejpřehlednější kód.

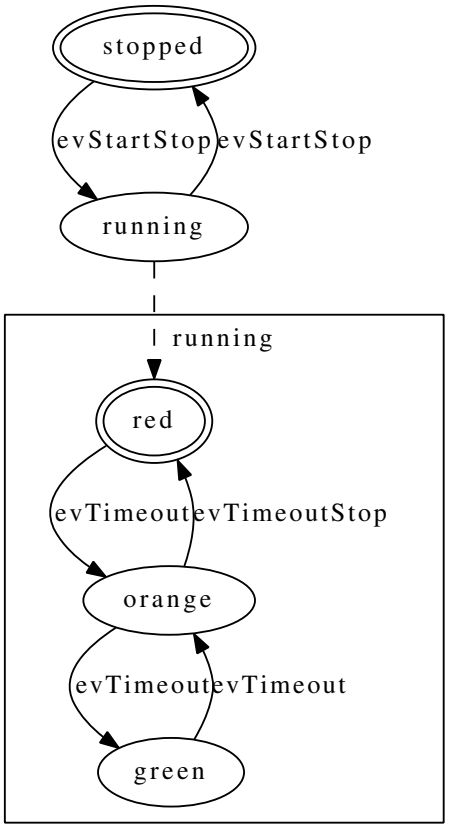
V původní verzi byl program vytvořen jako samostatná aplikace, která využívala potřebné knihovny. Ke korektnímu fungování potřebovala mít k dispozici standardní přepínače a šlo zvolit název výstupního souboru. V současné době je z programu vytvořen plugin pro překladač Clang. Z tohoto důvodu je nutno mít celý překladač zkompilován na svém operačním systému. To platilo i v původní verzi, protože knihovny, které se linkovaly k programu, se vytvářely při kompilaci celého překladače. Jelikož se jedná o plugin, tak se využívají přímo přepínače pro překladač. Ke správné funkci a kompletnímu vykreslení celého automatu musí zdrojový kód projít korektně analýzou, kterou provádí překladač.

Ještě jedna funkce byla odebrána z původní verze, kdy byl program nezávislý. Spolu se stavovým diagramem byla k dispozici i přechodová tabulka. A zobrazeny statistiky o stavovém automatu. Ve statistikách byla k dispozici stručná informace o počtu nalezených stavů, počtu nalezených událostí a celkovém počtu přechodů mezi stavy. Tato funkcionalita byla odstraněna právě v souvislosti s přechodem k pluginu.

Na závěr této sekce je uveden stavový diagram 2.5 jednoduchého automatu reprezentujícího chování semaforu na křižovatce. Zdrojový kód tohoto automatu je k dispozici na přiloženém CD včetně souboru, který umožňuje vygenerovat přesně tento obrázek.

Ve stavových diagramech jsou čárkovanou čarou vyznačeny vztahy mezi stavem a počátečním stavem ze skupiny jeho vnitřních stavů, pokud tedy nějaký existuje.

Bližší informace o tomto nástroji lze nalézt na webových stránkách projektu tohoto vizualizačního nástroje https://rtime.felk.cvut.cz/Statechart-viewer/, kde je také k dispozici zdarma zdrojový kód v repozitáři. Celý projekt je distibuován pod licencí GNU GPL.



Obrázek 2.5: Stavový diagram chování semaforu

2.3.4 FSME

FSME, Finite State Machine Editor, je vizualizační nástroj pro stavové automaty. Informace o tomto nástroji lze nalézt také v [Dar03].

I když se nejedná o zpětnou vizualizaci, ale stavový automat se přímo vizuálně navrhuje. Nástroj tedy neslouží jako zpětná vazba pro programátora, ale jeho práci při návrhu mu výrazně ulehčuje.

Tento nástroj není závislý na žádné knihovně. Stavové automaty jsou poté generovány s pomocí programovacího jazyku C++ nebo Python. Ve skutečnosti je výstupem z návrhu soubor ve formátu XML, ze kterého se poté dá dalším programem, fsmc, vygenerovat samotný zdrojový kód v C++ nebo s použitím pyfsmc v jazyku Python.

Kromě těchto dvou nástrojů, tedy fsme na kreslení stavových diagramů a fsmc na generování zdrojových kódů, obsahuje tato skupina i třetí nástroj, kterým je fsmd. Tato část slouží k online sledování běhu stavového automatu. Všechno, co se děje ve stavovém automatu, si lze prohlédnout graficky.

Nástroj nepoužívá žádnou standardní knihovnou pro stavové automaty. Další nevýhodou je nepřehlednost kódu v C++, protože celý automat včetně událostí je vytvořen jako jedna třída.

2.3.5 FSM tool

Tento nástroj je velmi podobný nástroji FSME. Opět slouží hlavně k návrhu stavových automatů. Asi největším rozdílem je to, že se již nejedná o SW, který je zdarma. Platí se ale pouze generátory zdrojových kódů. Samotný grafický editor je bezplatný. Dokumentace k nástroji StateForge je dostupná v [Hee13].

Stavový automat se nevytváří pomocí stavových diagramů, ale přechody mezi stavy jsou znázorněny pomocí přechodové tabulky. Ke grafickému editoru existuje i varianta, která je dostupná online bez nutnosti instalace. K popisu navržených stavových automatů slouží stejně jako v případě nástroje FSME upravené XML.

Celá kolekce dokáže generovat zdrojové kódy pro jazyky C++, Java a C#. Nástroj také kromě editoru stavových automatů a generátorů zdrojových kódů obsahuje online monitorování, ale již ne v grafickém režimu, ale pouze textové logování.

Velkou výhodou tohoto nástroje je podpora asynchronních automatů, událostí závislých na časovačích, paralelních automatů (ortogonální stavy) a také možnost návrhu hierarchických automatů. Škála podporovaných věcí se velmi blíží schopnostem knihovny Boost Statechart.

Pro využití nástroje jako vizualizátoru, tedy jako zpětnou vazbu by bylo nutno vytvořit program, který by z upraveného zdrojového kódu vygeneroval popis ve formátu FsmML, Finite State Machine Modeling Language. Právě ten je je právě použit k popisu pro generátory zdrojových kódů.

Z porovnání těchto nástrojů (FSME a StateForge) vyplývá, že v případě, pokud není potřeba využívat všech schopností, které StateForge při návrhu nabízí, tak je zbytečné platit za něco, co je k dispozici zdarma.

2.3.6 Visualsc

Visualsc, Visual State Chart Editor for SCXML, je nástroj pro vizualizaci stavových automatů popsaných pomocí SCXML. Nástroj umožňuje kromě vizualizace i návrh a editaci.

Vizualizací pomocí SCXML se zabývá několik dalších nástrojů, rovněž existují i nástroje, které z popisu v SCXML dokáží interpretovat zdrojový kód v různých programovacích jazycích. Příkladem mohou být Common SCXML pro Javu, PySCXML pro Python nebo Qt SCXML engine pro jazyk C++.

Na vizualizaci nebo generování kódu SCXML z navrženého stavového automatu existuje mnoho různých nástrojů. Jejich schopnosti jsou ve většině případů velmi podobné. Většina z nich využívá všech možností, které definuje UML, Unified Modeling, Language, podporují tedy prakticky stejné možnosti jako knihovna Boost Statechart. Všechny nástroje umí obrázek stavového diagramu nejen zobrazit, ale i vygenerovat.

Popis jazyku SCXML lze nalézt v [W3C12], krátký popis nástroje visualsc je v [Dre11],

odkazy na jednotlivé nástroje, které umí pracovat s SCXML jsou v [Apa13] pro jazyk Java, v [Rox13] pro Python a v [Qt12] pro C++.

2.3.7 Nástroj smach_viewer

Tento nástroj je součástí ROS, robotický operační systém. Patří k němu i knihovna stavových automatů smach. Pro vytváření stavových automatů v této knihovně se používá skriptovací jazyk Python. V knihovně lze vytvářet hierarchické stavové automaty. V tomto případě se z hlediska základních možností podobá knihovně Boost Statechart. Stejně jako knihovna je i nástroj na vizualizaci naprogramován v jazyku Python.

Nástroj pro vizualizaci má grafické uživatelské rozhraní. Samozřejmě umožňuje zobrazit stavový diagram. Stavové automaty jsou zobrazeny včetně jejich hierarchického uspořádání. Nástroj umožňuje zobrazit stavové automaty podle jednotlivých hloubek umístění stavů. Kromě zobrazení ve formě stavového digramu umožňuje zobrazit i strom stavů, ale již ne graficky, ale pouze textově. Ke generování diagramů je používán nástroj xdot, který umí interaktivně zobrazit diagram, který je popsán v jazyce dot z knihovny graphviz. Stejný jazyk je používán i pro vizualizaci v našem nástroji.

Díky uživatelskému rozhraní nástroj nejen zobrazí stavový diagram, ale také umí provádět analýzu stavového automatu přímo za běhu včetně posílání událostí do stavového automatu.

Program neprovádí žádnou kontrolu kódu stavového automatu ani neumí generovat stavové automaty z vytvořeného stavového diagramu. Informace o nástroji pro vizualizaci, tedy programu smach_viewer, jsou k dispozici zde [Boh10b] a popis knihovny smach je dostupný zde [Boh10a]

3 Tým Flamingos a jeho roboty

Tato kapitola se zabývá popisem robotu týmu Flamingos a hlavně popisem řídicího softwaru, jehož úprava je cílem této diplomové práce.

Nejprve je krátce představen tým Flamingos 3.1 a poté je zbytek kapitoly věnován popisu robotu 3.2 s důrazem na použitý řídicí SW 3.3.

3.1 Tým Flamingos

Tým Flamingos je robotický tým na Katedře řídicí techniky Fakulty elektrotechnické Českého vysokého učení technického. Tým vznikl přejmenování týmu CTU Dragons v roce 2010.

Tento robotický tým se od roku 2010 účastní mezinárodní soutěže Eurobot a v roce 2012 se také účastnil soutěže Sick robot day, která se konala ve městě Waldkirch v Německu. Pod starším názvem se tým samozřejmě účastnil také mezinárodní soutěže Eurobot.

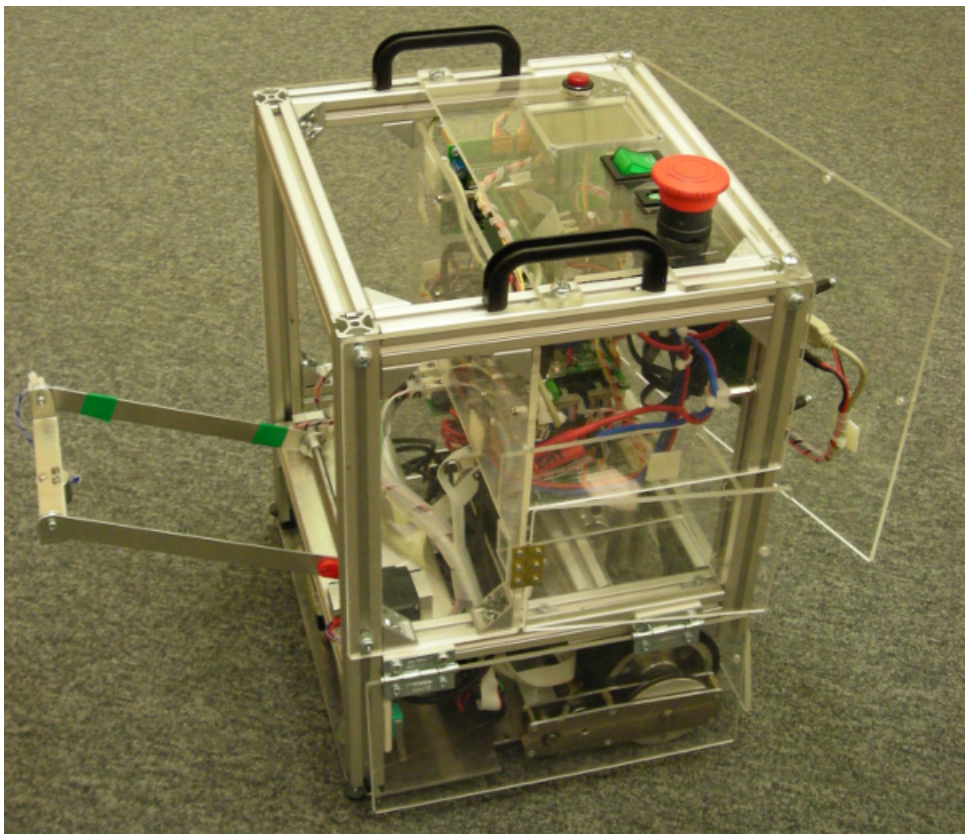
3.2 Roboty

K týmu patří v současné době 2 roboty. Oba jsou uzpůsobeny pro pohyb ve vnitřním prostředí. Na obou běží rovněž operační systém Linux. Princip řízení je u obou také stejný.

První robot je určen pro účast na soutěžích. Před každou soutěží jsou jeho akční členy upravovány tak, aby byly schopny splnit požadované úkoly. Robot má přibližné rozměry 310 x 260 x 340 mm. Informace o použité elektronice na robotu lze nalézt v [Kub10] a také v [Ša11] a informace o stavových automatech v [Jar10]. Druhý robot je určen pro demonstrační účely. Hledá objekty ve volném prostoru. Informace o demonstračním robotu jsou k dispozici v [Vok12]. Na obrázku 3.2 je fotka hlavního robotu a na obrázku 3.1 je fotka demonstračního robotu. Na soutěže se tedy používá vždy pouze hlavní robot.

3.3 Hlavní řídicí software

Část řídicího SW robotu je implementována pomocí stavových automatů. V současné době jsou v robotu použity 2 stavové automaty, kde jeden se stará o pohyb robotu a druhý je hlavní stavový automat, ve kterém je implementována strategie.

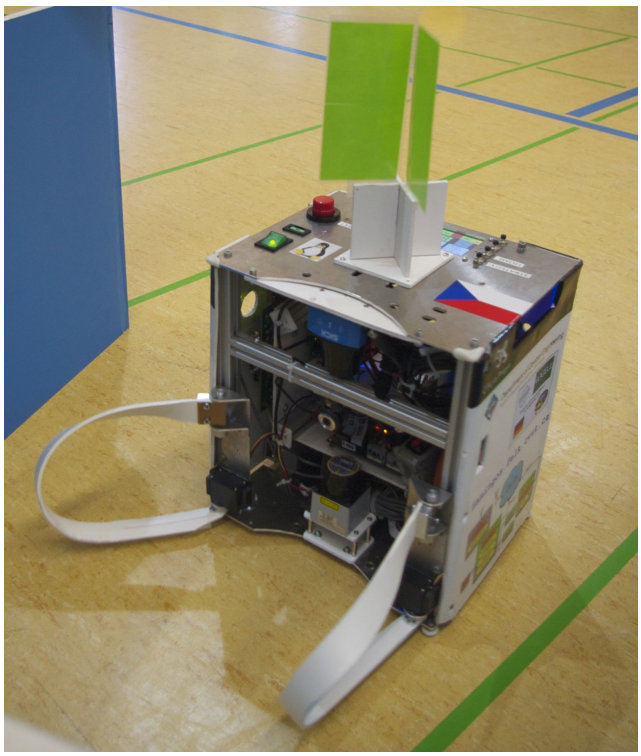


Obrázek 3.1: Fotka demo robotu, převzato z [Vok12]

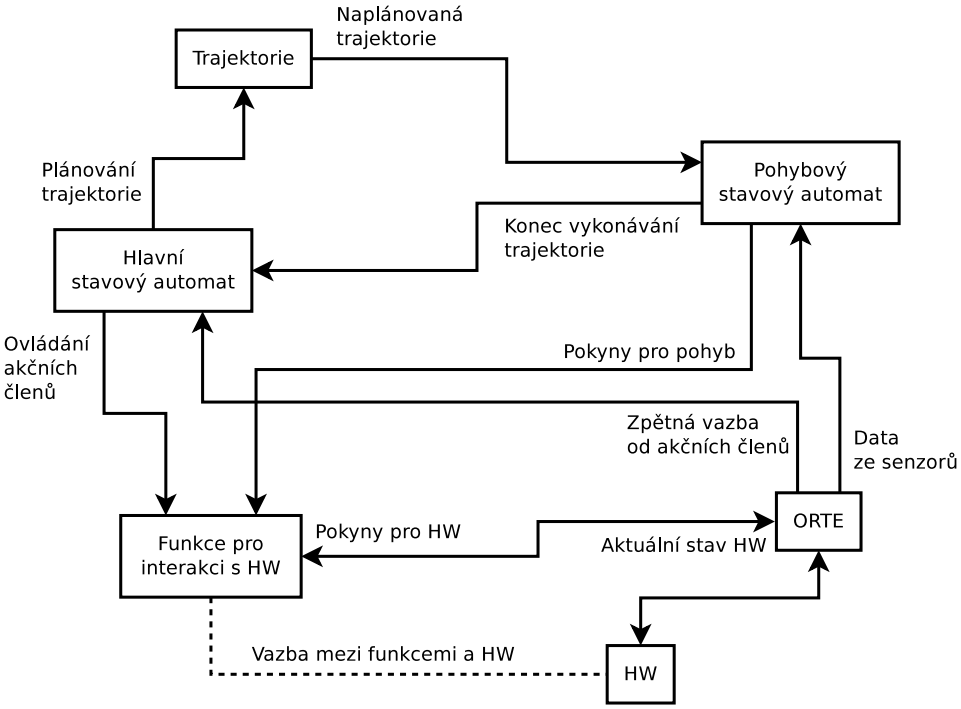
Použití stavových automatů je v tomto případě velmi výhodné. Jejich vytvoření je jednoduché a naprogramování celé soutěžní strategie je velmi rychlé. Stavovými automaty lze rovněž popsat většinu dějů. Mezi další výhody patří rovněž velmi názorná grafická reprezentace pomocí stavového diagramu. V případě, že existuje nástroj, který umožňuje generovat ze zdrojového kódu výsledný obrázek, tak je možnost získat grafické znázornění v podobě stavového diagramu ještě zjednodušena. Krátký popis knihovny je uveden v části 4.1.

Samotnou kapitolou v celém řídicím SW jsou pomocné funkce. Ty zajišťují například vytváření trajektorie, hlídání soutěžního času, ukládání informací o poloze, starají se o aktualizaci herní mapy a plní ještě další úkoly. Tyto funkce jsou také velmi důležitou součástí řídicího SW.

Kromě stavových automatů, které řídí vykonávání pohybu a připravené strategie, a pomocných funkcí se do řídicího softwaru řadí i další skupiny funkcí, které se starají o přímou interakci s HW robotu resp. tato interakce se děje prostřednictvím ORTE, což je komunikační middleware, kde se komunikuje pomocí zpráv. Na obrázku 3.3 je schéma komunikace mezi jednotlivými částmi robotu. Popis šipek je vždy uveden u jejich počátku. Další část se bude věnovat použitému komunikačnímu middleware.



Obrázek 3.2: Fotka hlavního robotu – soutěž Sick robot day 2012

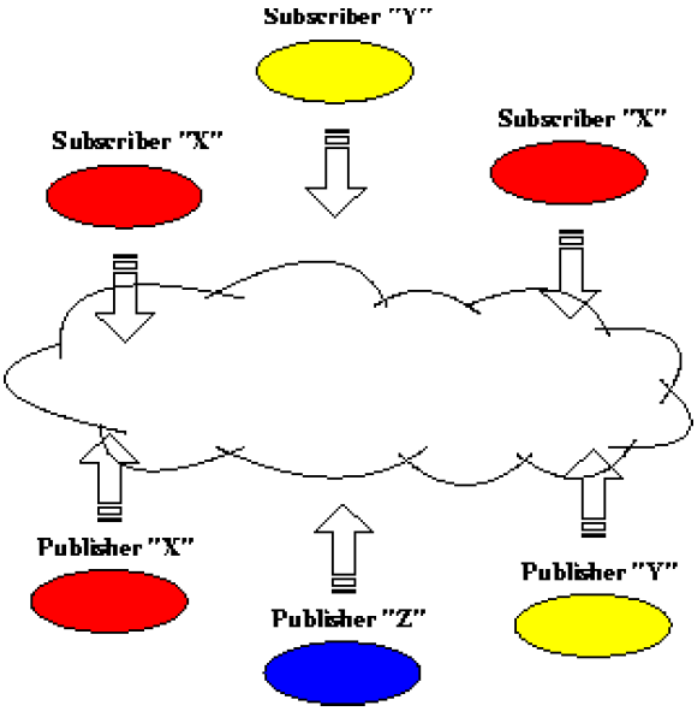


Obrázek 3.3: Schéma komunikace mezi částmi robotu

3.3.1 ORTE

ORTE, Open Real-Time Ethernet, je komunikační middleware pro real-time komunikaci. Implementuje nový aplikační protokol RTPS, Real Time Publisher Subscriber. Tento protokol je velmi jednoduše portovatelný na různé platformy. Celý protokol je vytvořen na základě UDP zásobníku, tím pádem je schopen řídit časování a spolehlivost přenosu.

ORTE poskytuje architekturu „publisher–subscriber“. Kdy každá aplikace je buď producentem (publisher) informací do ORTE nebo konzumentem (subscriber) informací z ORTE, ale může se samozřejmě chovat jako oba zároveň. Z tohoto důvodu se dá mluvit o komunikaci one to many. Veškerá komunikace může probíhat anonymně, protože distribuci dat řídí ORTE a data přeposílá do všech aplikací, které jsou připojeny. Na obrázku 3.4 je principiální schéma komunikace přes ORTE.



Obrázek 3.4: Schéma publisher–subscriber, převzato z [Smo+12]

Mezi výhody této architektury se dá zařadit:

- Aplikace, které používají architekturu publisher–subscriber jsou modulární a škálovatelné
- Podpora redundance publisher a subscriber, možnost replikace programu
- Vyšší efektivita oproti klient-server komunikaci

Není vhodné používat tuto architekturu při nízkém zatížení komunikačního kanálu. Další informace ohledně celého ORTE lze získat v [Smo+12].

V rámci aplikace v robotu týmu Flamingos se ORTE používá jako komunikační middleware a s pomocí ORTE zpráv se předávají data mezi jednotlivými aplikacemi. Kromě již popsaných aplikací, popsané v 3.3, se mezi další řadí například data z odometrie a to jak z nezávislé, tak data získaná z otáčení motorů. Dále sem patří také zpětné hlášky o stavu akčních členů nebo data z laserového dálkoměru.

4 Knihovny stavových automatů

Tato kapitola obsahuje popis obou knihoven stavových automatů a to původní knihovny FSM 4.1 a nové z kolekce knihoven Boost 4.2. Na závě této kapitoly 4.3 jsou ještě obě knihovny stavových automatů porovnány.

4.1 Knihovna FSM

Knihovna stavových automatů byla vytvořena bývalými členy robotického týmu Flamingos. Z tohoto důvodu se knihovna nepoužívá jinde než v týmu Flamingos. Celá knihovna je naprogramována pouze v jazyce C, aby šla tato knihovna jednoduše používat a výsledný zdrojový kód byl lépe čitelný, je pomocí maker vytvořeno jednoduché API (Application Programming Interface). Stavové automaty mohou běžet jak v jednom vlákně, tak každý stavový automat může běžet ve svém vlákně. Řešení problémů souběhu využívá POSIX API (mutexy, semafory, atd.).

Knihovna disponuje širokou podporou pro debuggování a logování veškerých procesů, které se v daném automatu odehrávají. Při testování má programátor k dispozici spoustu důležitých informací.

Knihovna podporuje kromě standardních automatů i subautomaty v jednotlivých stavech až do úrovně 10.

V rámci knihovny jsou definovány pouze obecné události. Vstup do stavu, výstup ze stavu, událost časovače a návrat ze subautomatu. Ostatní události jsou specifické pro každý automat a musí být definovány. Pokud bychom tedy potřebovali použít jednu událost ve dvou stavových automatech, tak ji musíme i dvakrát definovat. Definice se provádí v konfiguračních souborech a pomocí skriptu v jazyku Python se z nich vygenerují zdrojové kódy v C.

Omezením knihovny je možnost existence pouze jednoho časovače v rámci celého stavového automatu. Tím pádem je v rámci soutěže nutno měřit soutěžní čas ve speciálním vlákně a před každým soutěžním kolem je nutno hlavní program znovu zapnout. Toto omezení souvisí i s chybějící podporou hierarchických stavů.

Bližší informace o této knihovně lze nalézt v [Soj11]. V části 4.3 je ještě uvedena ukázka zdrojového kódu pro porovnání s novou knihovnou.

4.2 Knihovna Boost Statechart

Knihovna Boost Statechart, dříve známá jako Boost FSM, slouží k implementaci stavových automatů v C++. Základní myšlenkou, která utvářela celý styl knihovny je snaha o jednoduchý přepis mezi UML a C++. Celá knihovna je založena na metaprogramování pomocí šablon. Automaty, které lze vytvářet, patří mezi hierarchické stavové automaty.

Knihovna se dá rozdělit na dvě základní části podle toho, jaký typ automatu se dá vytvořit. První část je tvořena implementací pro synchronní stavové automaty. Tato část je pouze hlavičková. Druhou část tvoří asynchronní automaty, pro které jsou již zapotřebí další knihovny pro linkování, např. knihovna Boost Thread. V tomto případě je potřeba tedy celou sadu knihoven Boost zkompilovat. Bližší popis a konkrétní ukázky použití knihovny lze nalézt v internetové dokumentaci ke knihovně, která je dostupná v [HD07]-

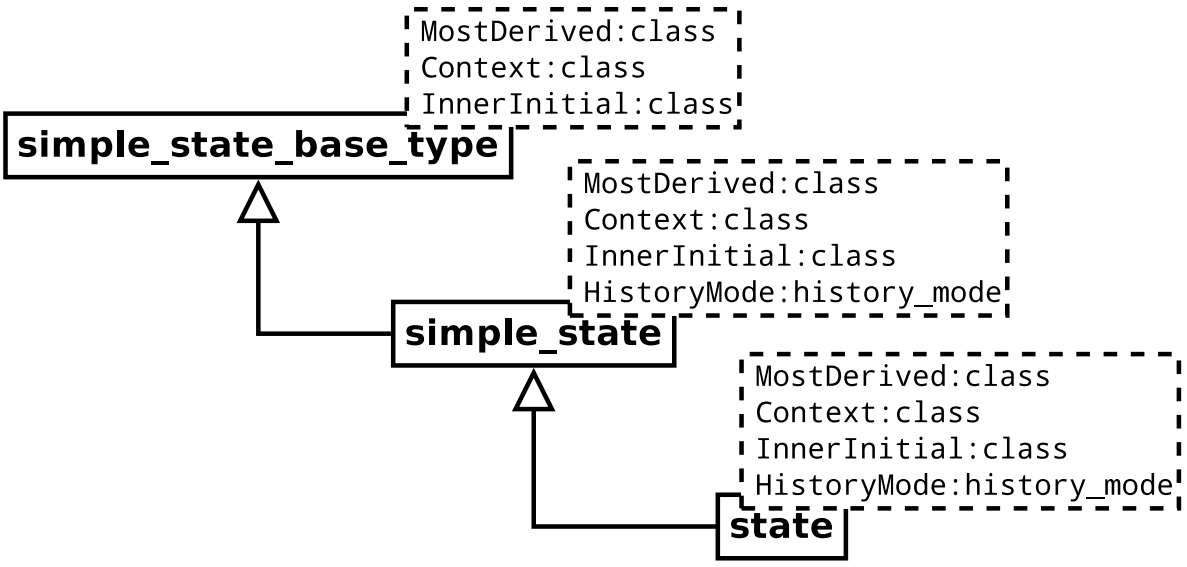
4.2.1 Synchronní automaty

Vůbec první částí, které je nutno věnovat pozornost je stavový automat. Ještě před samotným stavovým automatem je ale nejprve nutno definovat jednotlivé stavy (minimálně počáteční stav) a události. Pro definici stavů stačí pouze jejich výčet. Jak stavy, tak i události je výhodné reprezentovat jako struktury. Pokud použijeme struktury, není nutno se starat o přístupová práva. Veškeré metody, které vytvářejí chování stavového automatu, budou public a splní tak jeden z požadavků knihovny. V případě událostí je jejich společným předkem šablona třídy event, která má jeden povinný parametr a to název události. Druhým parametrem, který je nepovinný je alokátor. Ten určuje paměťový model pro danou třídu vytvořenou pomocí této šablony.

Nyní už lze deklarovat samotný stavový automat. I v tomto případě je vhodné použít strukturu. Synchronní automaty jsou z hlediska použití jednodušší. Předkem pro tyto stavové automaty je šablona třídy state_machine, která má 4 parametry. Prvním je název stavového automatu, druhým je počáteční stav. Třetím parametrem v pořadí je opět alokátor a posledním exception_translator, který slouží pro řešení výjimek, například při zpracování události, na kterou nemá automat definovanou reakci. První dva parametry jsou povinné a další dva již nepovinné. Události, které přijdou do synchronního automatu, se zpracovávají okamžitě, jakmile přijdou, neexistuje tedy žádná fronta událostí čekajících na zpracování. Ke stavovému automatu lze přistupovat přímo přes jeho objekt.

```
namespace sc = boost::statechart;
struct running;
struct stopped;
struct red;
struct orange;
struct green;
struct evTimeout : sc::event.< evTimeout >{};
struct evStartStop : sc::event.< evStartStop >{};
struct synchronous : sc::state_machine< synchronous, stopped > {};
```

Další částí, které je třeba se věnovat, jsou stavy. Opět je vhodné použít strukturu kvůli implicitním přístupovým právům. V rámci knihovny Boost Statechart existují dvě různé nadtřídy pro stavy. První je šablona třídy simple_state a druhou šablona třídy state. Obě třídy mají 4 stejné parametry, kde prvním je název stavu, druhým kontext, ve kterém se stav vyskytuje, třetím je název vnitřního stavu, pokud tedy existuje a posledním je history_mode. Kontextem stavu se zde rozumí stav, který leží z hlediska hierarchie výše. Místo kontextu by se tedy dal použít i pojem nadstav. Kontext stavového automatu je pro všechny stavy outermost (nejvzdálenější). Pokud už žádný stav výše není, tak je kontextem samotný stavový automat. Třetím parametrem je název vnitřního stavu, pokud tedy existuje a posledním je history_mode, který slouží k určení toho, jestli si má stavový automat pamatovat poslední konfiguraci aktivních stavů. Pro stavy jsou povinné pouze první dva parametry. Vztahy mezi těmito dvěma šablonami jsou ve formě diagramu tříd na obrázku 4.1.



Obrázek 4.1: Diagram tříd pro stavy

Hlavním rozdílem mezi třídami state a simple_state je v tom, že pro stavy typu state je

konstruktor povinný a má jeden parametr strukturu my_context. Použití šablony state je nutné jedině tehdy, pokud potřebujeme v konstruktoru stavu zavolat funkce, které pracují s historií či kontextem. Jakmile toto nepotřebujeme využít v konstruktoru, je lepší použít pouze simple_state a tím využít jednodušší a přehlednější kód.

```
struct stopped : sc::simple_state< stopped, synchronous >{
   /*Dalsi kod*/
};
struct running : sc::simple_state< running, synchronous, red >{
   /*Další kod*/
};
struct red : sc::state< red, running >{
   red(my_context ctx) : my_base(ctx){}
};
struct orange : sc::state< orange, running >{
   orange(my_context ctx) : my_base(ctx){}
};
struct green : sc::state< green, running >{
   green(my_context ctx) : my_base(ctx){}
};
```

Kromě obyčejných stavů knihovna umožňuje vytvářet i ortogonální stavy. Takové stavy umožňují paralelní, nezávislý, běh části stavových automatů. Každá událost, která přijde do takového stavu je replikována a poslána do každé nezávislé části.

Poslední částí, které je třeba se věnovat, jsou reakce na události. Již ve stavu jsou připraveny metody, do kterých lze nadefinovat reakci na vstup a opuštění stavu. Dle očekávání jsou reprezentovány konstruktorem a destruktorem. Tyto události jsou po zpracování kódu zahozeny a nelze na ně tedy přejít do jiného stavu. V rámci knihovny je definováno pět základních druhů reakcí na události. Reakce se navzájem vylučují. Aby je bylo možné kombinovat, existuje ještě jedna. Šestá možnost, která je úmyslně v seznamu uvedena jako poslední, umožňuje určit více reakcí pro jednu událost na základě dalších podmínek vázaných například na hodnotu proměnné.

- Přechod (transition) – aktuální stav je opuštěn (zavolán destruktor) a automat vstoupí do jiného stavu (zavolán konstruktor)
- Odložení události (defferal) – událost je odložena pro následující stav, do kterého se přejde.
- Vnitřní stavová reakce (in_state_reaction) – při příchodu události se zavolá funkce, která leží v jakémkoli aktivním stavu nebo ve stavovém automatu.

- Předání události do kontextu (forward_event) – pokud není specifikována akce, tak se událost předává do kontextu.
- Ukončení (termination) – slouží k ukončení běhu stavového automatu.
- Vlastní reakce (custom_reaction) – programátor ji specifikuje v rámci metody react, má návratovou hodnotu a tou je jakákoli ze zde vyjmenovaných druhů reakcí na události.

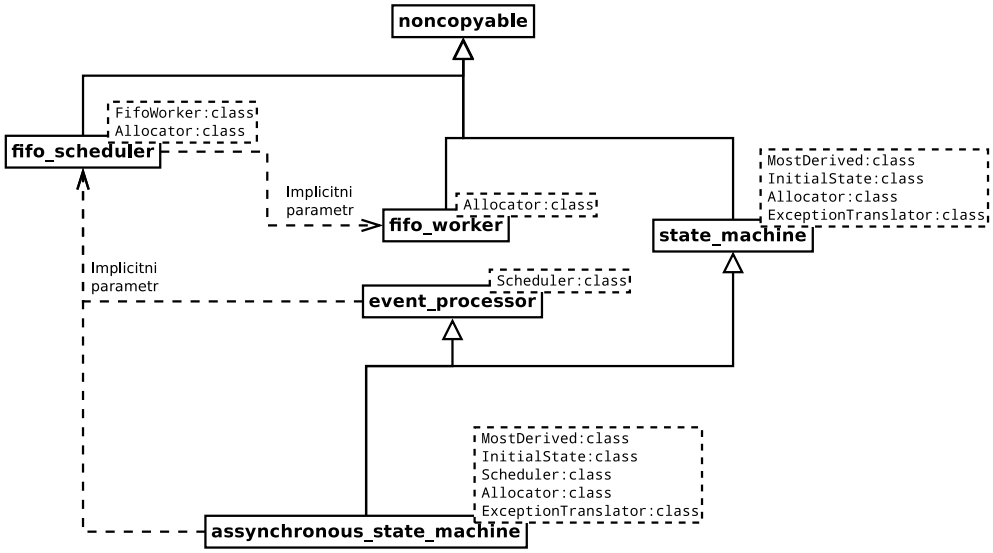
Veškeré události, na které se má reagovat musí být specifikovány v rámci vytvoření datového typu reactions uvnitř každého stavu pomocí typedef.

```
struct orange : sc::state< orange, running >{
   orange(my_context ctx) : my_base(ctx){}
   sc::result react(const evTimeout&){
      return transit< red >();
   }
   typedef sc::custom_reaction< evTimeout > reactions;
};
struct stopped : sc::simple_state< stopped, asynchronous >{
   typedef sc::transition<evStartStop, running> reactions;
};
```

4.2.2 Asynchronní automaty

V rámci plánované aplikace na robotu se budou používat právě asynchronní stavové automaty. Z hlediska definování stavů a používání událostí se dva typy, synchronní a asynchronní, vůbec neliší.

První rozdíl nastane při deklaraci třídy stavového automatu oproti třídě state_machine se musí použít třída asynchronous_state_machine. Pro ilustraci vztahů mezi jednotlivými šablonami tříd a třídami pro asynchronní část knihovny je na obrázku 4.2 UML diagram tříd s naznačenými vztahy.



Obrázek 4.2: UML diagram pro část knihovny

Jak je patrno z UML diagramu na obrázku, šablona třídy asynchronous_state_machine má jako předka šablonu state_machine. Tím pádem musí být schopna využít všechny její parametry. Kromě těchto čtyř parametrů je zde ještě jeden parametr a to je scheduler, přes který se přistupuje ke stavovému automatu (posílají se události, vytváří se, ukončuje se, ...). V případě asynchronních automatů již existuje fronta událostí čekajících na zpracování. V dalších odstavcích jsou probrány dva koncepty, které jsou při použití asynchronních automatů potřeba. V UML diagramu jsou tyto koncepty použity pro šablony tříd fifo_worker resp. fifo_scheduler. Na závěr je ještě popsána šablona třídy event_processor.

FifoWorker koncept V knihovně jsou v původní verzi dvě úrovně pro zpracování událostí. Událost je poslána přes objekt, který respektuje koncept FifoScheduler do objektu reprezentujícího koncept FifoWorker.

Nyní je čas na to uvést, co tedy koncept FifoWorker specifikuje. První specifikací je to, jak se chovat v případě, že fronta událostí je prázdná, a druhou je hlídání problémů souběhu. Zamykání kritických sekcí pomocí mutexu atd. Uživatel má možnost si podle tohoto konceptu takovou třídu vytvořit. Je na něm zda použije neprioritní obyčejnou frontu, nebo využije také prioritizaci určitých událostí.

V rámci knihovní implementace je událost poslána přes fifo_scheduler, kde se vytváří work_item (ukazatel na metodu, která událost zpracovává). Metoda, na kterou ukazuje je přímo zpracování události v určitém automatu, přesněji v jeho event_processor. Tento ukazatel se poté předává do na uložení právě do třídy, která využívá koncept FifoWorker. Uvnitř takové třídy se tedy udržuje pouze fronta událostí čekajících na zpracování ve formě work_item.

K tomuto objektu nemá uživatel přímý přístup. Veškerý proces probíhá automaticky. Lze pouze specifikovat, jak se má chovat v případě vyprázdnění fronty. Ovlivnit lze pouze to, při

kolika zpracovaných událostech se má zpracovávání ukončit.

Scheduler koncept Tento koncept je první úrovní při zpracování události. Právě přes objekt scheduler se stavové automaty vytvářejí, posílají se do nich události a ukončují se. Tento objekt je jedinou možností, kterou má programátor na to, aby ovlivnil chování stavového automatu v případě asynchronních stavových automatů.

Definuje to, jak se sdílí jedna fronta mezi více objektů třídy event_processor. Jak již bylo zmíněno, určuje také, jak se přidávají události do fronty. Stejně jako FifoWorker určuje, jak se má zachovat v případě, že se fronta vyprázdní. Jako poslední lze zmínit specifikaci toho, jak zpracovat událost, která patří event processoru, který již není aktivní.

V knihovně je reprezentován pomocí šablony třídy fifo_scheduler. Tato šablona má 2 parametry (fifo_worker a alokátor). Oba jsou nepovinné. I pro fifo_scheduler lze specifikovat, jak se má chovat v případě vyprázdnění fronty. Ovlivnit lze pouze to, při kolika zpracovaných událostech se má zpracovávání ukončit.

Event processor Jak již název napovídá, jedná se o šablonu třídy, která slouží ke zpracování událostí. Má pouze jeden parametr a tím je scheduler. Pokud není určeno jinak, tak se používá fifo_scheduler z knihovní implementace. V knihovně plní funkci základní šablony pro všechny ostatní šablony tříd, které zpracovávají událostí a tím pádem i pro šablonu asynchronous_state_machine.

V případě asynchronních automatů se lze dokonce dostat, při využití funkcí, které se dotazu-jí na outermost kontext, kromě metod ve stavovém automatu dokonce i k metodám v šabloně event_processor a to díky aplikované dědičnosti. Tím pádem máme k dispozici i veškeré metody použitého scheduler. V šabloně event_processor jsou implementovány pouze metody pro zpracování události, inicializaci a ukončení stavového automatu. Poté ještě metody, které jsou schopné vrátit objekt scheduleru a objekt processor_handle, který je v knihovně reprezentován pomocí objektu processor_container, kde je uložen objekt event_processor a fifo_scheduler.

4.2.3 Porovnání asynchronních a synchronních automatů

Hlavním rozdílem mezi těmito dvěma druhy je fronta událostí čekajících na zpracování. V případě synchronních automatů úplně chybí naopak u asynchronních taková fronta existuje. V případě synchronních automatů může dojít při zpracování událostí k preempci (zpracování událostí skončí v jiném pořadí než začalo). Někdy nám takové chování vadit nemusí.

Příkladem může být automat, do kterého události přicházejí pouze z venku. Tam dokonce v případě běhu celého programu v jednom vlákně k žádné preempci událostí dojít ani ne-

může. Jakmile existuje více stavových automatů, které spolu komunikují, tak již chování bez preempce zaručeno není.

Naopak v případě synchronních automatů nejsou řešeny problémy souběhu, takže se snižuje při běhu zátěž a časové využití procesor. Informace o náročnosti zamykání kritických sekcí jsou k dispozici v [Gov08].

4.3 Porovnání knihoven

Na úvod této části jsou obecně porovnány knihovny a poté je uvedena ukázka zdrojových kódu pro stejný stav. Na základě těchto kódů je nejprve provedeno porovnání jak z hlediska délky, tak z hlediska názornosti.

Obě knihovny podporují hierarchické umístění stavů. První knihovna ve formě subautomatů a druhá hierarchických stavových automatů. Rozdíly mezi těmito dvěma přístupy jsou popsány v části 2.2 včetně grafické reprezentace takových automatů.

Výhodou knihovny FSM je široká možnost logování všeho, co se ve stavovém automatu děje. Tahle schopnost chybí v knihovně Boost Statechart. Rovněž v ní chybí událost časovače, ale její doplnění je součástí zadání této práce.

Původní knihovna je napsána v jazyce C, ale k jejímu kompletnímu využití je potřeba i jazyk Python, který slouží k definování událostí. Naopak knihovna Boost Statechart využívá pouze jazyk C++. Knihovna je napsána objektovým přístupem pomocí šablon.

Nyní už jsou na řadě ukázky zdrojového kódu z obou knihoven. Pro ilustraci je zvolen jednoduchý stav. Nejprve je uveden zdrojový kód v původní knihovně FSM.

```
FSM_STATE(my_state) {
   switch(FSM_EVENT) {
      case EV_ENTRY:
         /*code*/
         break;
      case EV_ONE:
         FSM_TRANSITION(another_state);
         break;
      case EV_TWO:
         /*code*/
         FSM_TRANSITION(another_state2);
         break;
   }
}
```

Nyní už následuje ukázka zdrojového kódu pro stejný stav, který je ale implementován v nové knihovně Boost Statechart.

```
namespace sc = boost::statechart;
struct my_state : sc::simple_state<my_state, FSM> {
    my_state() {
        /*code*/
    }
    sc::result react(const EV_TWO& ) {
        /*code*/
        return transit<another_state2>();
    }
    typedef Boost::mpl::list<
        sc::transition<EV_ONE, another_state>,
        sc::custom_reaction<EV_TWO> > reactions;
};
```

Z hlediska délky kódu je zdrojový kód nové knihovny, tedy knihovny Boost Statechart, kratší. Jeho hlavní výhoda nejvíce souvisí s jeho větší názorností. U takového stavu je ihned jasné jaký stav je kontextem stavu. U knihovny FSM tohle vůbec patrné není. Dokonce tam není rozdíl mezi stavem v automatu a stavem v subautomatu. Jediná metoda, jak zjistit v jaké hloubce daný stav leží je nalézt stav, ve kterém se do subautomatu vchází nebo do stavu, kde se daný subautomat opouští.

Naopak z hlediska počtu znaků vychází srovnání opačné. Počet znaků v implementaci knihovny Boost Statechart je mnohem vyšší. To snižuje částečně čitelnost tohoto kódu.

Z mého pohledu je také trochu názornější oddělené umístění reakcí na události v metodách než v switch case. Při drobné chybě, kdy programátor neúmyslně zapomene klíčové slovo break, tak může dojít k úplně jinému chování stavového automatu. Na chybějící break neupozorní ani kompilátor, ale naopak na chybějící návratovou hodnotu je programátor upozorněn. Možnost vynechání má i své výhody, kdy lze specifikovat stejný kód pro více událostí. To lze provést i v knihovně Boost Statechart za pomoci polymorfismu. Toho se dá dosáhnout pomocí společné nadtřídy pro více událostí. Bohužel tím se trochu snižuje čitelnost kódu.

Dalším omezením, které přidává knihovna Boost Statechart je chybějící možnost přechodu na vstup či opuštění stavu.

5 Návrh a implementace

Tato kapitola se zabývá návrhem řešení a jeho implementací. Nejprve jsou v sekci 5.1 zmíněny úpravy v knihovně Boost Statechart, v sekci 5.2 jsou probrány úpravy, které byly provedeny ve zbývající části řídicího SW a v polsední sekci 5.3 jsou zmíněny úpravy v nástroji pro vizualizaci stavových automatů.

5.1 Úprava knihovny Boost Statechart

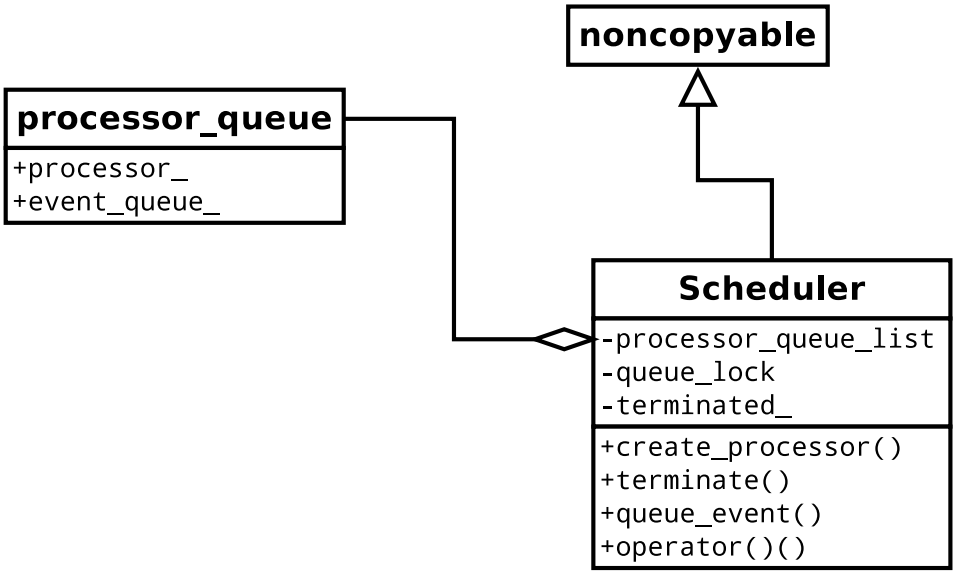
Prvním problémem, se kterým je nutno se vypořádat je chybějící podpora pro časovače. Dalším problémem je závislost na knihovně Boost Thread, která poskytuje programátorovi prakticky stejnou podporu pro vlákna jako knihovna pthread ze standardních POSIX knihoven.

Nejprve se tato část věnuje nahrazení vláken v sekci 5.1.1 a až poté doplnění podpory časovačů 5.1.2 a sloučení obou těchto úprav a zakomponování do knihovny Boost Statechart 5.1.3.

5.1.1 Nahrazení podpory vláken

V knihovně je nutno nahradit šablony fifo_scheduler a fifo_worker. Důvodem pro jejich oddělení v knihovně je to, že pokud je potřeba jiný typ fronty než fifo, například prioritní, tak stačí nahradit fifo_worker. V našem případě mají všechny události stejnou prioritu, takže stačí obyčejná fronta. Tím pádem lze obě třídy sloučit do jedné.

Jelikož nový scheduler plní i úlohu fifo_workeru, bylo nutno přidat i seznam aktivních stavových automatů včetně událostí čekajících na zpracování. Stejně jako v knihovně byla pro tuto funkci použita struktura processor_queue. Tato struktura obsahuje automat ve formě event_processor a frontu událostí čekajících na zpracování. Pro manipulaci je používán ukazatel na tuto třídu, který je vytvořen pomocí typedef na processor_handle. Návrh nové třídy Scheduler je na obrázku 5.1.



Obrázek 5.1: UML diagram návrhu nové třídy Scheduler

Pokud se podíváme na jednotlivé metody, tak funkce queue_event, již dle svého názvu, zajišťuje přidání události do fronty událostí příslušného stavového automatu. Operátor () plní funkci hlavní smyčky, která čeká na poslání události do jakéhokoli automatu. Metoda create_processor vytváří stavový automat a také tento automat vrací ve formě ukazatele na processor_queue, tedy ve formě processor_handle. Poslední metodou je terminate, která ukončí všechny automaty, které patří příslušnému objektu třídy Scheduler a uvolní dynamicky alokovanou paměť.

Jako první jistě bude programátorem použita metoda create_processor. Jedná se o šablonu metody. Šablona má jeden parametr a to je třída, či struktura, stavového automatu, který se má vytvořit. Metoda vytvoří objekt stavového automatu, provede inicializaci automatu (konstruktor počátečního stavu) a spolu s prázdnou frontou na události vrátí ve formě processor_handle. Rovněž si ho ve stejné formě uloží do seznamu aktivních automatů.

Druhou metodou, která ovlivní objekt stavového automatu je terminate. Ta už podle svého názvu ukončí všechny aktivní stavové automaty, které běží v rámci daného objektu třídy Scheduler.

Další metodou je queue_event. Ta přidá událost do fronty příslušného stavového automatu a upozorní hlavní smyčku na novou událost čekající na zpracování.

Poslední metodou je hlavní smyčka, která je reprezentována pomocí operátoru (), a tedy to nejdůležitější pro chování stavových automatů. Jelikož se jedná o hlavní smyčku, tak se v ní zpracovávají události, které jsou přes metodu queue_event poslány do stavového automatu. Zpracování událostí se provádí přes všechny aktivní automaty. Není tedy zajištěno, že dvě události, které přijdou po sobě do různých automatů jsou zpracovány v pořadí, ve kterém do objektu třídy scheduler přišly. V rámci jednoho automatu je toto samozřejmě zajištěno díky tomu, že se používá fronta. Stejné vlastnosti při zpracování událostí měla i původní knihovna

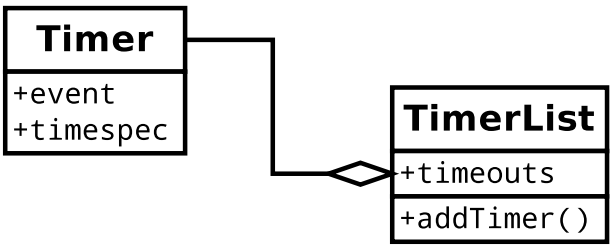
FSM. V jejím případě existovalo omezení na maximální délku fronty, která byla nastavena na 10. Odstranit tento nedostatek při zpracování událostí by šlo pomocí sdílení jedné fronty pro všechny automaty. S ohledem na návaznost na původní knihovnu to ale není třeba.

5.1.2 Doplnění podpory časovačů

Než bylo možno přidat časovače do knihovny, tak bylo třeba vůbec udělat návrh pro časovače. Oproti knihovně FSM bylo rozhodnuto o n2kolika změnách. Počet časovačů v jednom stavu ani v jednom automatu není omezen. Časovače musí umět posílat do automatu i jinou událost než jen evTimer. Druhá podmínka souvisí právě s větším množstvím časovačů.

V souvislosti s těmito úpravami bylo nejprve nutno navrhnout strukturu časovače. U této struktury není třeba mít žádné funkce, ale je důležité, aby všechny jeho atributy byly veřejně přístupné. Důvod je velmi prostý. Časovač je nutné kromě spouštění i zastavovat, takže je potřeba přístup k nastavenému času pro vypršení. Právě proto byla vybrána struktura, u které není třeba řešit a hlídat veřejná přístupová práva, která jsou u ní implicitní. Takový časovač musí obsahovat informaci o události, kterou má do automatu poslat po vypršení a čas kdy přednastavený čas vyprší. Eventuálně by mohl obsahovat informaci o tom, jestli je vůbec aktivní. Pro jednoduchost bylo zvoleno, že neaktivní časovač má nastavenou dobu vypršení na 0.

Pro manipulaci s časy byly použity již vytvořené a používané funkce uvnitř knihovny FSM. Pro časovače byla navrhnuta třída TimerList, která si udržuje přehled o aktivních časovačích pro každý automat. Tedy na jeden objekt stavového automatu připadá jeden objekt TimerList. Návrh třídy TimerList a struktury Timer je na obrázku 5.2. Jelikož vyšší abstrakce je v tomto případě zbytečná nemá třída žádné předky ani není předkem žádné třídy.



Obrázek 5.2: UML diagram návrhu nových tříd pro časovače

Jak je z obrázku patrno, třída obsahuje pouze metodu addTimer. Tato metoda nevytváří objekt časovače, ale pouze nastavuje jeho parametry. Jakmile jsou nastaveny, časovač se přidá do seznamu. Aby se dal jednoduše vyhledávat nejkratší časovač ve všech automatech, tak se seznam vytváří rovnou seřazený. K řazení se využívá algoritmus Insert Sort [Alg07].

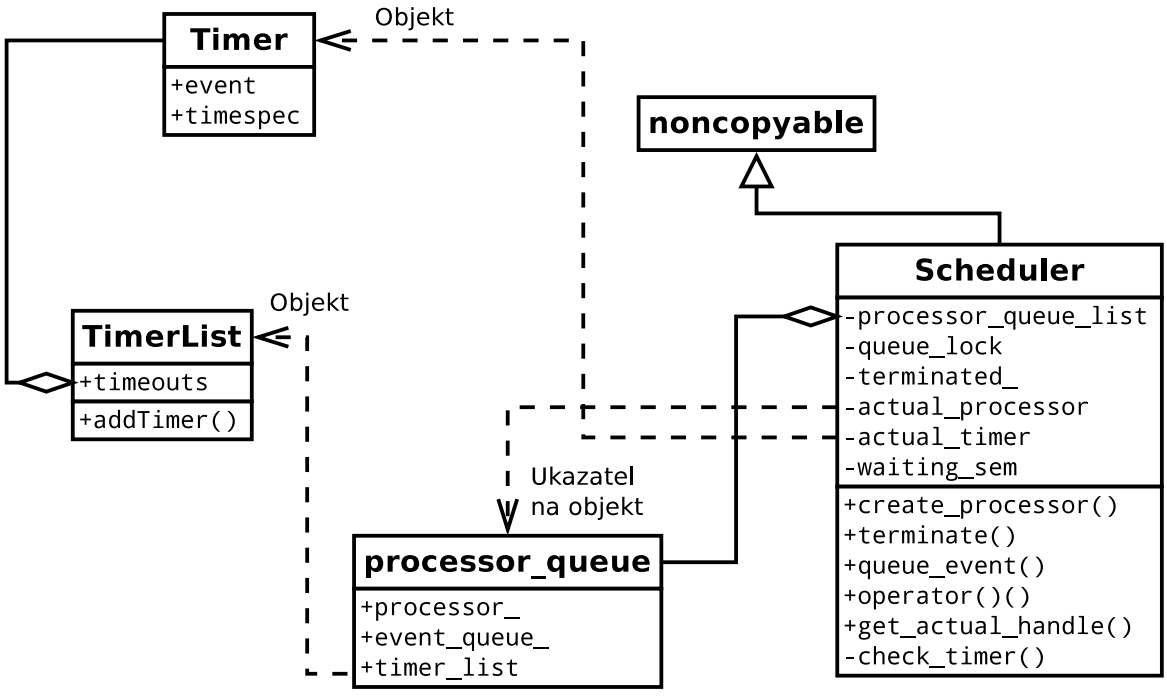
Nepředpokládá se, že ve stavu bude existovat současně více než 10 aktivních časovačů, takže není třeba používat složitější struktury pro ukládání jako jsou stromy.

Vytvářet další metody by bylo v tomto případě zbytečné. Jedinou možností, nad kterou by se dalo uvažovat je zastavení časovače, ale pro umístění této metody byl vybrán přímo stav automatu.

5.1.3 Sloučení obou úprav

Dalším krokem bylo doplnění časovačů přímo do knihovny a tím zkompletování celé úpravy. Po zvážení několika možností se jako nejlepší ukázala volba doplnit časovače do struktury processor_queue. Ta tedy nyní obsahuje kompletní informace o automatu, frontě čekajících událostí a aktivních časovačů.

Po tomto rozhodnutí se ukázalo, že je ještě nutno do třídy Scheduler přidat další funkce. Zejména funkci, která mezi aktivními automaty vyhledá ten, jehož časovač vyprší nejdříve. V souvislosti s touto novou funkcí musely být do třídy scheduler přidány další atributy, které se vztahují právě k časovačům. Na obrázku 5.3 je finální návrh vztahů mezi jednotlivými nově vytvořenými třídami.



Obrázek 5.3: UML diagram nově vytvořených třídy s jejich vztahy

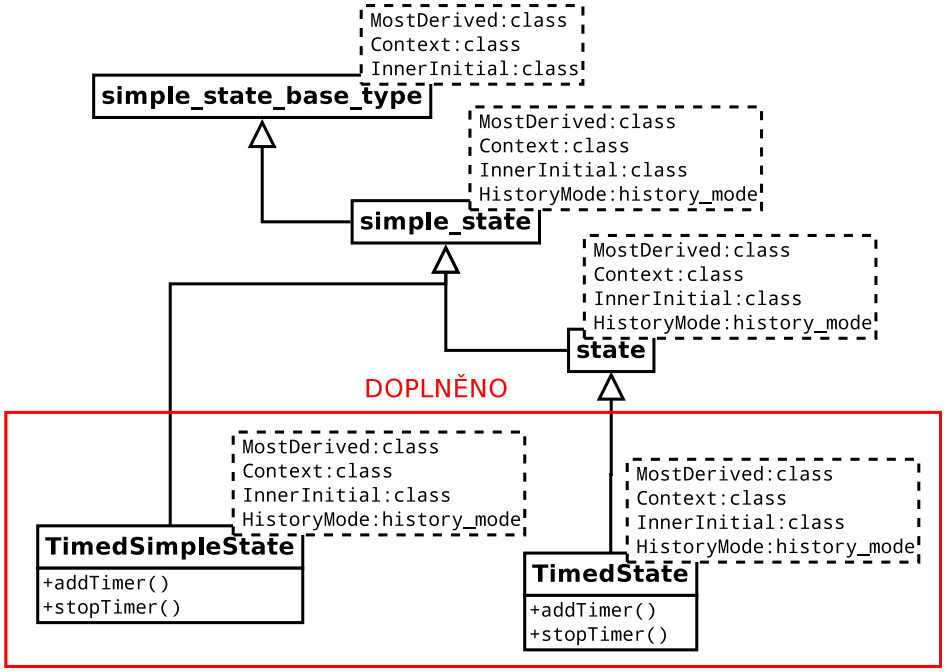
Do třídy Scheduler přibyly metody get_actual_handle a check_timer. Metoda get_actual_handle vrací poslední automat, který zpracoval událost, ve formě processor_handle.

Metoda check_timer je přístupná pouze z ostatních funkcí ve třídě. Jak již název napovídá, vrací informaci o tom, jestli v nějakém automatu, který je vytvořen v tomto objektu, existuje časovač. Pokud nějaký existuje, tak časovač s nejkratším časem vypršení je uložen v proměnné actual_timer a protože musí být uložena i informace o automatu, kterému daný časovač patří,

tak je v proměnné actual_processor uložen processor_handle. Pro hledání se využívá faktu, že časovače jsou ve frontě již seřazeny. Není tedy třeba procházet celou frontu.

Do struktury processor_queue přibyl seznam aktivních časovačů ze všech právě aktivních stavů v daném automatu. Ostatní již existující metody byly zachovány v původním rozsahu.

Pro ulehčení práce programátora bylo ještě rozhodnuto o vytvoření těchto dvou šablon tříd: TimedSimpleState a TimedState. Obě tyto šablony jsou závislé pouze na struktuře, která reprezentuje časovač, na ostatních třídách, které jsou zmíněny výše již závislé nejsou. Na obrázku 5.4 je UML diagram těchto nově vytvořených šablon tříd. Stejně jako knihovní šablony podporují i tyto ortogonální stavy.



Obrázek 5.4: UML diagram šablon pro stavy s časovači

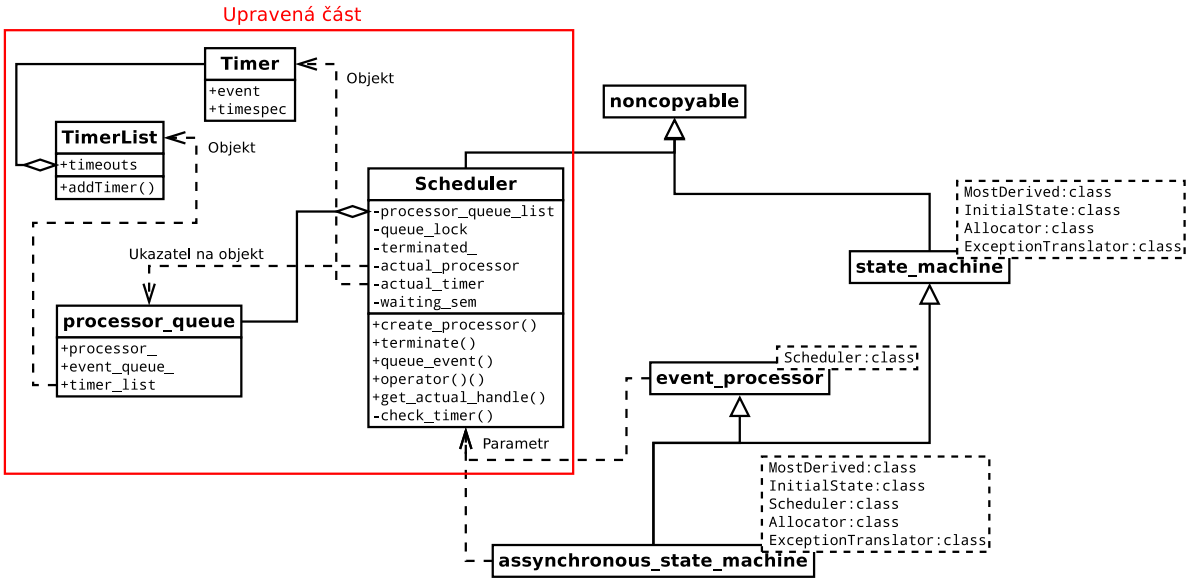
Musí to tedy být šablona třídy, které má čtyři parametry. Kromě všech poděděných metod obsahují metody runTimer a stopTimer. Poslední část, kterou přidávají, je implementace jednoduchého logování aktuálních stavů, aby se daly publikovat do ORTE a přebírat v dalších aplikacích jako je např. robomon (vizualizátor stavu robotu). Aby bylo zajištěno to, že žádné časovače ve frontě nezůstanou po opuštění daného stavu, obsahují tyto šablony i seznam aktivních časovačů ve stavu. Seznam již není seřazený.

Metoda runTimer má stejné parametry jako metoda addTimer ve třídě TimerList. V této metodě je právě funkce ze třídy TimerList volána. V rámci této metody se kromě volání ještě přidá objekt časovače do vnitřního seznamu.

Druhou metodou je stopTimer. Tato metoda časovač zastaví. Jediným parametrem je právě reference na objekt časovače. Tento časovač se poté odstraní z globálního, ale i z vnitřního, seznamu.

V konstruktoru se provádí logování aktuálního stavu pomocí metody třídy Robot. V destruktoru se využije právě seznam časovačů v daném stavu. Postupně se prochází celý seznam a všechny aktivní časovače se odstraňují z globálního seznamu.

Na obrázku 5.5 je UML diagram všech úprav a jejich zakomponování do původní knihovní implementace v knihovně Boost Statechart.



Obrázek 5.5: UML diagram zapojení úprav do knihovny

Další úpravy již nejsou potřeba. Všechny vytvořené úpravy jsou stejně jako celá knihovna pouze hlavičkové soubory. Úprava není vůbec závislá na dalších knihovnách v sadě Boost. Není třeba tedy celou sadu knihoven kompilovat. Celý styl úprav a doplnění se snažil o dodržení stylu knihovny Statechart ve smyslu jednoduchosti a snadného použití. Bohužel došlo k omezení přenositelnosti mezi operačními systémy. Knihovna pthread, která je použita k synchronizaci vláken není oproti knihovně Boost Thread úplně platformově nezávislá.

5.2 Úprava řídicího SW

Kromě stavových automatů do řídicího systému patří i mnoho skupin funkcí, které poskytují programátorovi rozhraní pro ovládání robotu uvnitř stavových automatů. Do této skupiny patří například funkce pro ovládání jednotlivých akčních členů, funkce pro pohyb robotu, funkce pro hlídání soutěžního času a ještě několik dalších.

V úvodu této části jsou obecně popsány metody, které byly použity ke zjednodušení a zpřehlednění zdrojových kódů. Poté jsou konkrétněji popsány jednotlivé nově vzniklé třídy. Na závěr jsou popsány oba stavové automaty, které běží na robotu. Jedná se o automat pro pohyb a hlavní automat obsahující strategii.

V jednotlivých funkcích nebyly provedeny výrazné změny. Některé funkce byly zrušeny, protože mohly být nahrazeny například pouze konstruktorem třídy resp. struktury. Ukázkou může být následující kód. Nejprve jeho varianta v jazyce C.

```
   struct move_target_heading {
      enum move_target_op operation;
      float angle;
      float distance;
   } ;
   static inline struct move_target_heading __target_heading(enum move_target_op op,
float angle, float distance) {
      struct move_target_heading th;
      th.operation = op;
      th.angle = angle;
      th.distance = distance;
      return th;
   }
```

A nyní varianta v jazyku C++ s využitím konstruktorů ve struktuře move_target_heading, která je kratší.

```
   struct move_target_heading {
      move_target_op operation;
      float angle;
      float distance;
      move_target_heading(){ }
      move_target_heading(move_target_op op, float ang, float dist) : operation(op),
angle(ang), distance(dist) { }
   } ;
```

Další změny byly provedeny v souvislosti s odstraněním funkcí, které se staraly pouze o měření času. Celá tato část byla smazána bez náhrady, protože se hlídání soutěžního času přesunulo přímo do stavových automatů.

Také musely být provedeny změny v jiných částech SW, které souvisí s názvy nově vytvořených tříd. Problémem se ukázal název Robot, protože tento název byl již použit v robotickém simulátoru. Aby byl tento problém odstraněn, byl jako nový název pro původní třídu Robot v simulátoru, zvolen RoboSim. Důvodem pro změnu jsou problémy při kompilaci. V rámci původní třídy Robot z robotického simulátoru jsou totiž volány funkce ze třídy Robot na robotu. V původním stavu to nevadilo, protože se původní struktura jmenovala robot.

V obecných částech byly nahrazeny konstrukce s použitím malloc nahrazeny operátorem new. Tato změna si vynutila ještě záměnu dealokace. Tedy výměnu delete za free. U funkcí,

kde to bylo výhodné, byly rovněž nahrazeny ukazatele referencemi. Typickým příkladem jsou get metody, které vrací vnitřní proměnné v třídách. Pro konstanty bylo také použito klíčové slovo const místo define, protože C++ kompilátor dokáže v tomto případě výsledný kód lépe optimalizovat.

Ještě by se zde dalo zmínit odstranění některých klíčových slov. Typickým příkladem může být struct u ukazatelů na struktury nebo enum u výčtových typů.

5.2.1 Třída Robot

V rámci této třídy byly zachovány funkce, které se starají o inicializaci a ukončení celého programu včetně ukončení a startu stavových automatů. Pouze se z nich staly metody. V původním stavu byly i funkce, které pouze vracely polohu robotu a v části move helper byly funkce, které nastavovaly polohu robotu. V nové implementaci jsou tyto funkce z části move helper odstraněny a přesunuty právě do třídy Robot.

Z původní struktury robot byly odstraněny proměnné reprezentující trajektorii, ta je nyní umístěna ve třídě MoveHelper a proměnná, ve které byla umístěna mapa. Ta je nyní ve třídě MapHandling.

Novou funkcí tvoří set_state_name, jejíž prostřednictvím se přistupuje k logování aktuálních stavů do ORTE a je tedy volána v konstruktorech jednotlivých stavů. Kromě konstruktorů stavů se tato funkce volá při přepínání trajektorií, aby byla na displeji u robotu vidět aktuálně zvolená trajektorie.

5.2.2 Třída MapHandling

V této třídě je nyní uložena mapa. Její funkce jsou pouze přesunuty dovnitř třídy a to podle potřeby přístupových práv do skupiny private nebo public. Všechny funkce, které byly pouze přesunuty dovnitř třídy, slouží k zápisu dat do mapy. Do mapy se zapisují překážky, které jsou detekovány s pomocí laserového dálkoměru, jehož data přicházejí skrz ORTE. Vzhledem k tomu, že byl do třídy přidán ukazatel na mapu, byla vytvořena metoda, která mapu vrací, aby se dala používat i jinde. Rovněž musí existovat i metoda, která mapu umožňuje nastavit.

Mimo třídu zůstaly pouze funkce, které se starají o zapomínání překážek. Takto vytvořenou mapu si lze zobrazit v rámci robotického simulátoru, kde lze sledovat i postupné zapomínání překážek.

5.2.3 Třída MoveHelper

Do třídy byl přesunut ukazatel na aktuálně vytvářenou trajektorii. Do třídních metod se přesunuly prakticky všechny funkce, které sloužily dříve. Jedinou změnou z hlediska počtu bylo

odebrání funkcí pro nastavení pozice.

Uvnitř původních funkcí nebyly provedeny žádné funkční změny. Pouze došlo ke sloučení funkcí, pokud kód, který vykonávaly, byl stejný. To se týkalo například funkcí robot_moveto_notrans a robot_goto_notrans. U této dvojice se kód lišil pouze v jedné proměnné, která určovala, zda se má použít plánovač trajektorie anebo ne.

Mimo třídu zůstaly funkce, které převáděly úhel a souřadnice na základě startovní barvy. Do třídy nebyly rovněž přesunuty funkce, které převáděly koncovou akci u trajektorie např úhel, pod kterým se má přijet nebo úhel otočení v koncovém bodě. Tyto funkce zůstaly pro jednodušší použití schovány za instrukcemi preprocesoru.

5.2.4 Třída Actuators

Uvnitř třídy je uložen ukazatel na ORTE. Důvodem pro jeho uložení je přímé posílání pokynů do ORTE pro jednotlivé akční členy. Ve třídě jsou metody pro pohyb pacek a výtahu. Dále také metoda pro nastavení ukazatele na ORTE. Poslední sadou jsou metody, které vrací informaci o posledním příkazu pro jednotlivé akční členy. Tyto metoda slouží pouze k přenosu informací do robomonu.

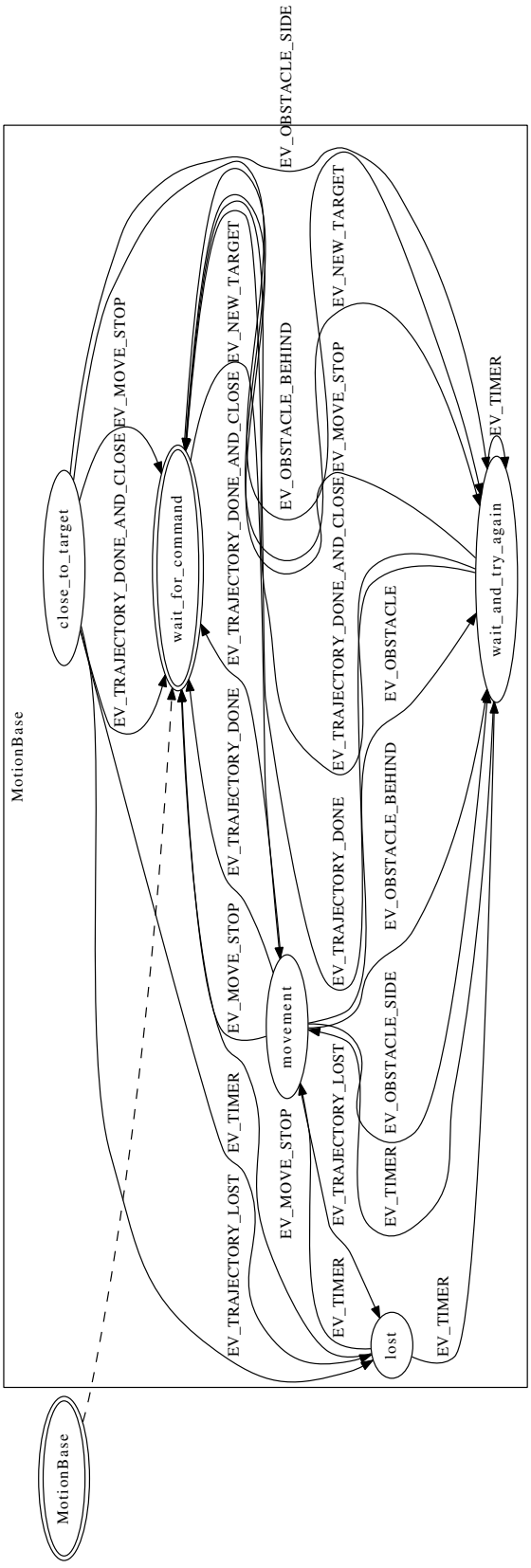
Informace o posledním pokynu pro aktuátory se ukládají do vnitřních proměnných ve třídě. V případě pacek nelze nastavit jiný povel pro pravou a jiný pro levou. Obě musí vykonávat stejnou akci. V rámci inicializace jsou nastaveny packy na otevřené, protože se vždy při bootování robotu otevírají.

5.2.5 Stavové automaty

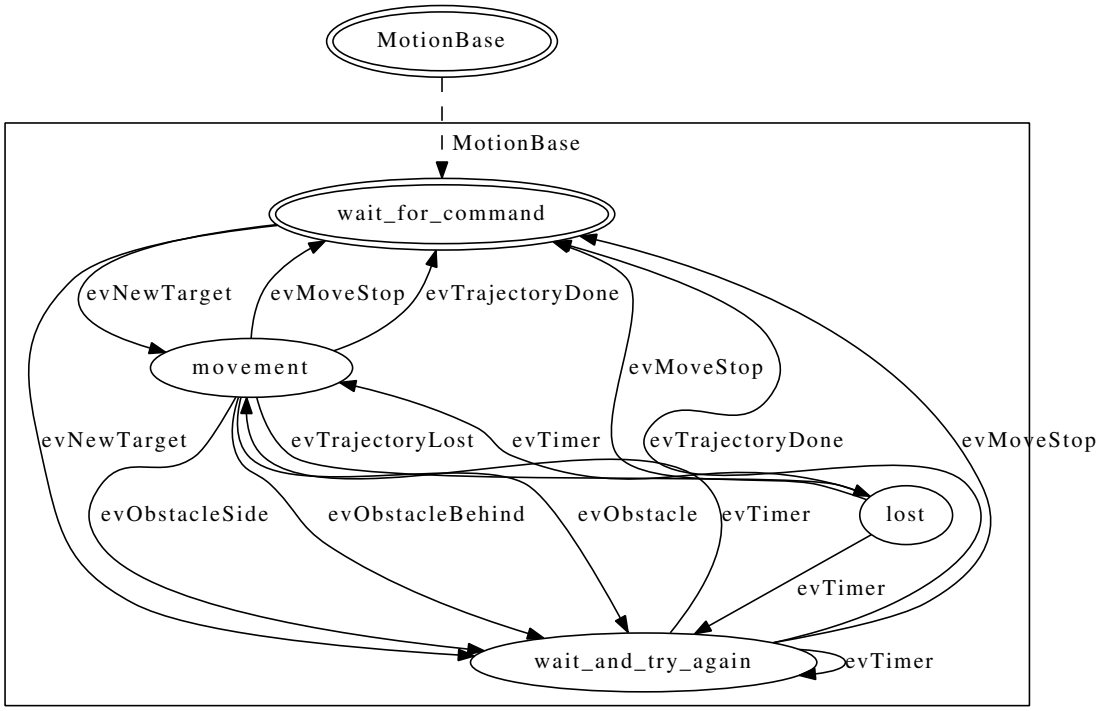
U obou stavových automatů, které běží v rámci robotu, došlo k využití hierarchických stavů. Za jejich pomocí se podařilo zjednodušit některé funkce.

V případě automatu, který se stará o pohyb, je celý automat umístěn do dvou vrstev. V jedné vrstvě je stav, který se stará o logování nezpracovaných událostí. Tím pádem se tato funkce již nemusí řešit v jednotlivých stavech. V první fázi, kdy došlo k pouhému přepisu automatu do nové knihovny stylem 1 ku 1, byl analyzován stavový diagram. Tento stavový diagram je na obrázku 5.6. Ve stavovém diagramu je již využito zjednodušené logování.

Z obrázku je patrno, že ze stavu close_to_target se sice přechází, ale do něj žádný přechod již nevede a ani se nejedná o počáteční stav. Takový stav již nemusí v automatu vůbec existovat. Další analýzou bylo zjištěno, že na dvě definované události EV_MOTION_DONE a EV_MOTION_DONE_AND_CLOSE se reaguje stejně. Je tedy zbytečné mít tyto události oddělené a může se zachovat pouze jedna z nich. Stavový diagram po těchto úpravách je na obrázku 5.7.



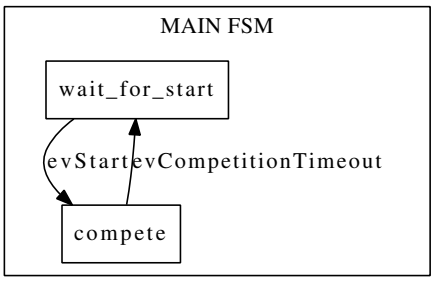
Obrázek 5.6: Původní varianta stavového automatu pro pohyb



Obrázek 5.7: Nový stavový automat pro pohyb

Z porovnání stavu před úpravami a po úpravách je jasné, že došlo k výraznému zjednodušení a omezení počtu přechodů mezi stavy. Původní stavový automat měl 6 stavů a 23 přechodů naopak nový stavový automat má jen 5 stavů a pouze 15 přechodů. Inicializace objektu stavového automatu je obsažena uvnitř metody init ve třídě Robot.

Další část textu v této části se věnuje hlavnímu automatu. V jeho případě není popis tak konkrétní, protože na každou soutěž je potřeba jiný stavový automat. Na závěr je ukázka soutěžního automatu ze soutěže Eurobot 2012. V případě hlavního automatu byla zvolena architektura, která je zobrazena na obrázku 5.8. V případě hlavního automatu již není inicializace automatu umístěna uvnitř třídy Robot, takže bude vždy umístěna ve funkci main.



Obrázek 5.8: Architektura soutěžních automatů

Základní myšlenkou tohoto umístění stavů je možnost restartu celého automatu bez nutnosti znovuspouštění celé aplikace. Druhým faktem, který architekturu ovlivnil, je potřeba měření soutěžního času. Obě tyto funkce umožňuje právě hierarchické uspořádání stavů. V pří-

padě stavu wait_for_start robot čeká na povel ke startu a jsou v něm rovněž nastavovány startovní pozice robotu a startovní poloha všech aktuátorů. Naopak ve stavu competing probíhá měření soutěžního času. V případě více trajektorií existuje také více stavů competing odlišených podle názvu trajektorie. Na základě zvolené trajektorie se poté přechází do příslušného stavu při odstartování. Další variantou by byl pouze jeden stav competing a v něm jeden obecný stav init a až z něj by se podle zvolené strategie přecházelo do jiného stavu. Stavový diagram by se stal trochu více nepřehledným. V obou případech by logování bylo umístěno ve stavu competing. Na obrázku 5.9 je reálný stavový automat ze soutěže Eurobot 2012. Tento automat byl již přepsán s navrženou architekturou pro soutěžní automaty. Pro implementaci byla využita první metoda, tedy více stavů competing, kdy každý zahrnuje pouze jednu trajektorii.

Stavy competing tvoří strategy_homologation a strategy_central_buillon. V případě druhé strategie je vidět i její členění na jednotlivé úkoly, které se v průběhu strategie provádějí. Pokud je potřeba prohodit jednotlivé úkoly, tak stačí prohodit přechody ve vrstvě, kde úkoly leží. Se samotnými stavy, které leží uvnitř úkolů se nic neprovádí. Přepínání mezi strategiemi je řešeno v rámci stavu wait_for_start.

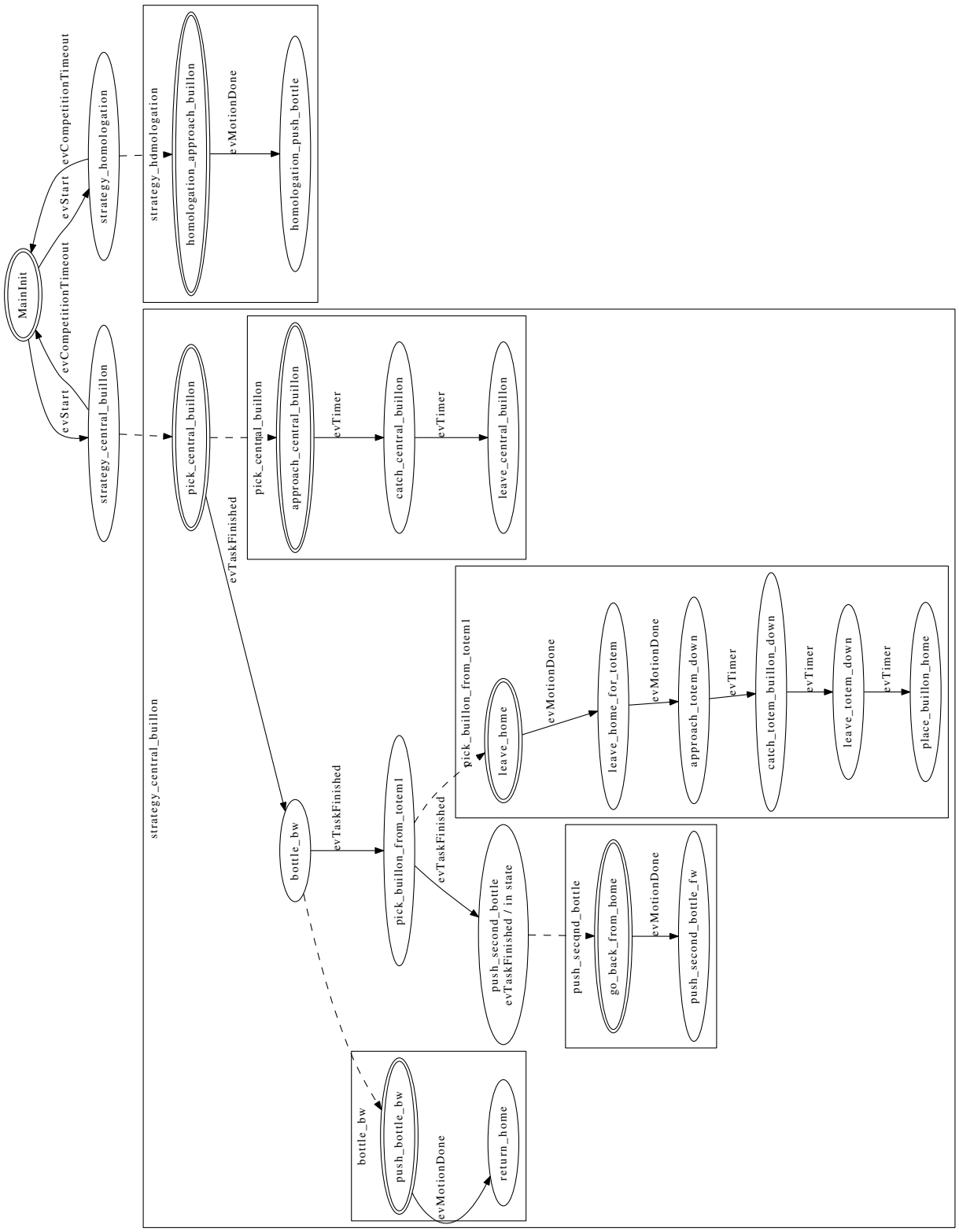
5.3 Úprava vizualizačního nástroje

Úpravy vizualizačního nástroje vychází z porovnání s již existujícími nástroji, které se také zabývají vizualizací stavových automatů. Popisy jiných nástrojů pro vizualizaci jsou v části 2.3.4 až 2.3.7. Porovnání se soustředí pouze na smach_viewer, který zobrazuje také hierarchické automaty a rovněž používá jazyk dot. Schopnosti obou nástrojů z hlediska škály automatů, které dokáži vizualizovat, jsou prakticky stejné.

Samozřejmě, že výhodou, kterou smach_viewer má, je GUI, ale to není cesta, kterou se vývoj a úpravy budou ubírat. Vizualizace aktivních stavů je velmi dobrá věc, ale tato vlastnost je již na robotu implementována, sice pouze v textovém režimu v rámci robotického simulátoru nebo na displeji, který je umístěn na robotu. Tam se informace dostávají pomocí ORTE, ale i to je dostačující pro testování korektního chování. Cílem nástroje totiž je pouze jednoduchá zpětná vazba o stavovém automatu.

Mnohem zajímavější jsou funkce programu xdot, který je schopen při kliknutí na daný stav odkázat na příslušné místo do zdrojového kódu. Tahle vlastnost je jedním z cílů, kterého také chceme dosáhnout.

Oproti tomuto nástroji se rovněž vývoj bude soustředit na kontrolu chyb, které může programátor vytvořit a tím mu zjednodušit jejich hledání. Výhodou je v tomto případě to, že program je pluginem do běžného kompilátoru, který má velmi dobře propracovanou diagnostiku.



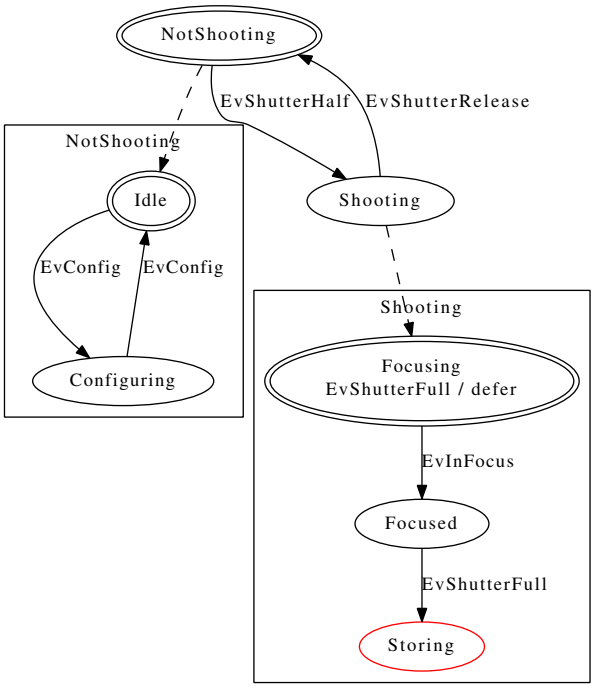
Obrázek 5.9: Automat ze soutěže Eurobot 2012

5.3.1 Úpravy a jejich implementace

V souvislosti s úpravami celé knihovny, aby šla použít na robotu a splňovala naše požadavky, musí dojít i k drobným úpravám v nástroji pro vizualizaci. Úpravy souvisí naštěstí pouze s tím, že musí být ignorovány šablony TimedState a TimedSimpleState, které nevytvářejí samotné stavy. Tohle je způsobeno tím, že cílem úprav bylo, co nejméně ovlivnit samotnou knihovnu. V programu je již ignorována šablona state, protože stavy se vyznačují tím, že jako předka mají šablonu simple_state.

Další úpravy, které jsou součástí zadání diplomové práce, se soustředí na zpětnou vazbu pro programátora. První úpravou z této skupiny je kontrola typedef na datový typ reactions, ve kterém musí být definovány všechny reakce. I přestože tenot typedef chybí, tak je automat kompilovatelný a kompilátor nenahlásí žádnou chybu. Samozřejmě, že v diagramu nejsou z takového stavu žádné přechody, ale i přesto se hledání, v případě složitějšího stavového automatu, může protáhnout. Takový stav je ve stavovém diagramu vyznačen červenou barvou, ale kromě této grafické zpětné vazby je vypsáno pomocí diagnostiky překladače varování. Jedná se pouze o nestandardní stav a nikoliv o chybu.

Pro ilustraci této funkce se dá použít stavový automat, který je distribuován jako příklad ke knihovně. Jedná se o stavový automat reprezentující chování fotoaparátu, jehož stavový diagram je na obrázku 5.10.



Obrázek 5.10: Stavový diagram s chybějícím typedef reactions

V případě, že typedef na datový typ reactions existuje, tak mohou nastat dvě možnosti. První je případ, kdy událostí v tomto typedef reactions je více než příslušných metod. Opět je

program kompilovatelný a chyba je odhalitelná jen testováním nebo důsledným porovnáním přechodů ve stavového diagramu s návrhem. Postupně jsou procházeny události a pro ty, ke kterým neexistuje příslušná react metoda, je vypsána chyba.

Poslední možností je to, že v typedef na datový typ reactions jsou i události, pro které neexistuje příslušná metoda. V takovém případě již automat nelze zkompilovat a kompilátor nahlásí chybu. Taková chyba se přece jen diagnostikuje jednodušeji díky varování překladače. Události se párují s react metodami, které jsou vytvořené ve stavu. Na konci analýzy stavu je u metod, ke kterým chybí událost v typedef reactions, vypsáno varování.

Další přidanou funkcí je hlídání definovaných událostí. Program v tomto případě hlídá, aby všechny definované události byly použity. V případě, že nejsou všechny použity, tak dostane programátor informaci přes výstup kompilátoru. Informace se vypisuje na konci běhu vizualizačního pluginu až jsou zanalyzovány všechny zdrojové soubory.

Mezi další úpravy se dá zařadit doplnění podpory událostí, které jsou odloženy a zpracovány v dalším stavu (tzv. defered events). Tyto události jsou označeny ve stavu, kde jsou definovány (vypisují se do stavového diagramu).

Podobným případem jsou události, při jejichž příchodu se pouze zavolá nějaká funkce z jakéhokoli aktivního stavu či samotného stavového automatu (in_state_reaction). Opět jsou tyto události označeny v příslušném stavu ve stavovém diagramu.

S ohledem na provedené úpravy byla rovněž aktualizována internetová prezentace tohoto projektu. V současném stavu je projekt schopen korektní funkce s LLVM a Clang verze 3.0 a vyšší.

5.3.2 Budoucnost nástroje

Již v současné době je nástroj používán několika dalšími vývojáři. Pro zjednodušení komunikace byl z tohoto důvodu zřízen mailing list.

S ohledem na vývoj LLVM a Clang bude nutno vždy mezi jednotlivými verzemi pravděpodobně nástroj vždy mírně upravit.

Budoucí směřování vývoje tohoto vizualizačního nástroje vychází z toho, na jaké problémy během řešení narazíme a jaké další funkcionality by byly pro programátora užitečné. Jak již bylo zmíněno výše, tak jednou z takových je navigace do zdrojového kódu při kliknutí na stav v diagramu.

6 Testování

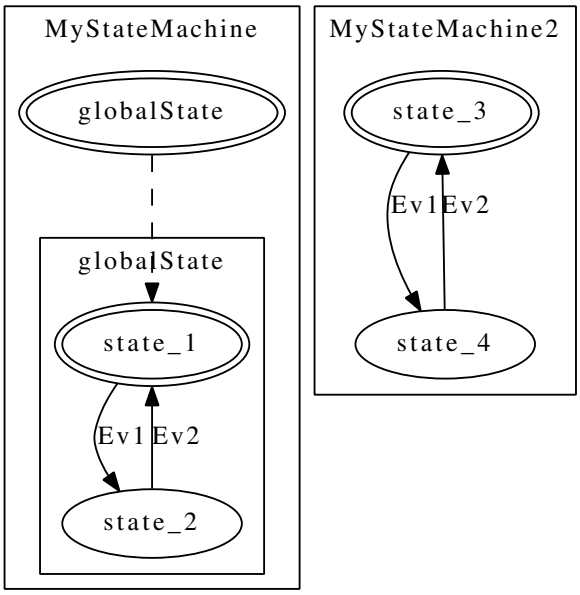
Testování všech SW součástí diplomové práce se dá rozdělit do dvou hlavních částí. První je testování úpravy knihovny včetně testování úprav řídicího SW a druhou je testování upraveného nástroje pro vizualizaci. Obě tyto části se nedají od sebe jednoznačně oddělit, protože při přepisu stavových automatů do nové knihovny, se přirozeně pro kontrolu využíval nástroj pro vizualizaci.

I přes tuto provázanost, bude v této kapitole práce nejprve popsáno testování pouze upravené knihovny včetně přidaných časovačů na jednoduchém stavovém automatu 6.1 a až poté testování včetně upraveného řídicího SW 6.2. V závěru této kapitolz bude v sekci 6.3 ještě krátce zdokumentováno testování nástroje pro vizualizaci.

6.1 Testování upravené knihovny

Pro tuto fázi testování byl vytvořen stavový automat, ve kterém se kromě úvodní události, která nezávisela na časovačích. Ostatní události byly navázané na časovače. V úvodní fázi se jednalo o automat, který měl pouhé dva stavy, mezi kterými se přecházelo, a jeden úvodní stav, který celý automat odstartoval.

Jakmile tento automat fungoval správně, tak byl vytvořen podobný automat, který se do testování přidal. Tímto způsobem se testoval běh více automatů v jednom vlákně a v jednom objektu Scheduler. Díky existenci dvou automatů šlo otestovat i posílání událostí mezi nimi. Po dokončení této fáze testování se mohlo přejít do další části vývoje, kterou byly úpravy řídicího SW a testování na automatech, které mají reálný podklad. Pro ilustraci této části je na obrázku 6.1 diagram obou stavových automatů. Obrázek nebyl vygenerován přímo nástrojem pro vizualizaci, protože jeho omezením je podpora pouze jednoho stavového automatu v souboru.



Obrázek 6.1: Ukázka stavového diagramu testovacích automatů

Jelikož tato dvojice automatů otestovala veškeré přidané a upravené schopnosti knihovny Boost Statechart, nebylo nutné vytvářet další stavové automaty. Stejně bude tato část průběžně testována v dalších fázích testování.

6.2 Testování řídicího SW

Tato fáze zahrnuje nejdelší část, která se věnovala testování. Z hlediska přístupu se dá rozdělit na dvě fáze. První je testování na robotickém simulátoru a druhá je testování na reálném robotu. Důvodem pro rozdělení na dvě části je to, že pokud bude vše fungovat v simulátoru, je to dobrým předpokladem pro to, aby vše fungovalo na robotu.

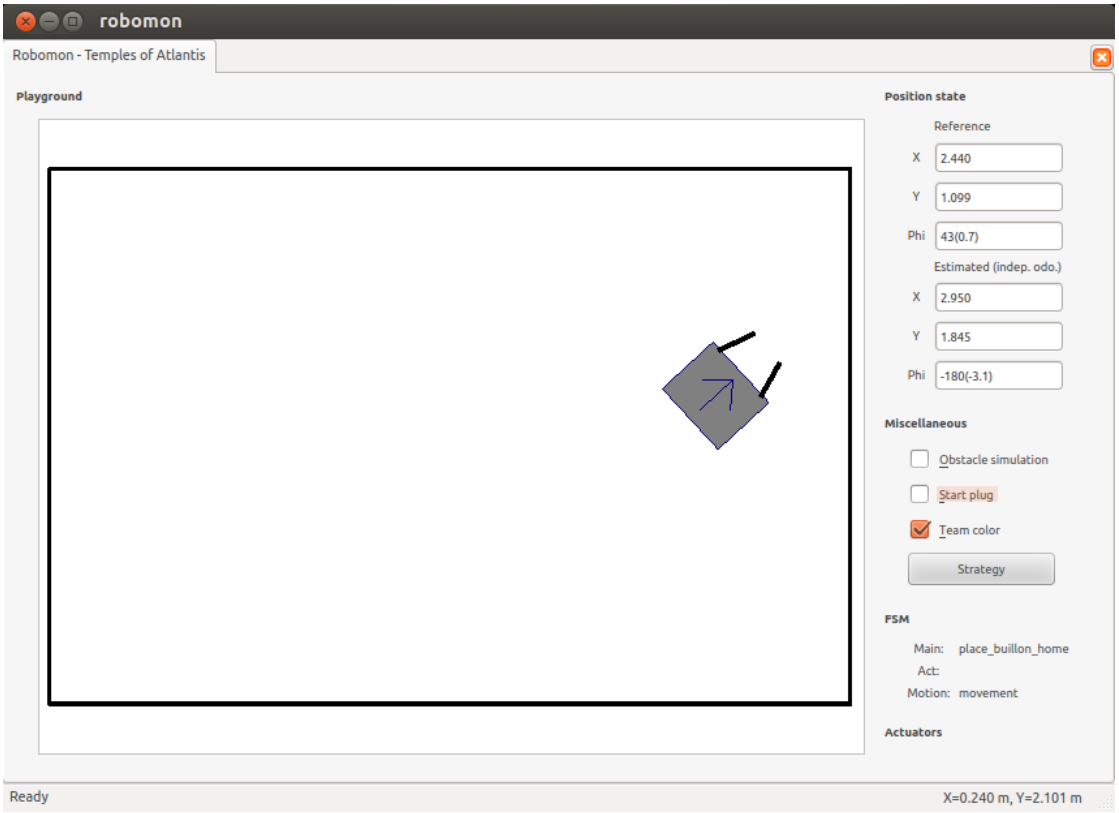
Všechny provedené úpravy v řídicím SW, včetně nahrazení knihovny stavových automatů, byly integrovány do standardního sestavovacího nástroje, který je používán v rámci robotického týmu Flamingos. Jedná se o nástroj OMK.

6.2.1 Testování v simulátoru

Aby mohlo být spuštěno testování v simulátoru, musely být vytvořeny stavové automaty pro testování. Jedná se o hlavní stavové automaty. Pro tuto fázi byly podle původních testovacích automatů, které vykonávají například pohyb po čáře a pohyb do obdélníku, vytvořeny jejich varianty pro testování s novou knihovnou stavových automatů.

Robotický simulátor robomon umožňuje kromě simulace pohybu také online sledování pohybu robotu včetně porovnávání referenční pozice s pozice podle obou odometrií (nezávislé

i odometrie z motorů). Tato schopnost je důležitá při testování na reálném robotu. V okně simulátoru se rovněž dají přepínat barvy, podle kterých se přepočítávají body pro trajektorie, ovládat startovací tlačítko a samozřejmě i přepínat strategie před odstartováním. V simulátoru se dají rovněž vytvářet umělé překážky pro robot, aby se kompletně dal otestovat stavový automat pro pohyb. Poslední důležitou schopností pro testování je zobrazení aktuálního stavu ve všech běžících automatech. Na obrázku 6.2 je screenshot robotického simulátoru.



Obrázek 6.2: Screenshot obrazovky robotického simulátoru

Kromě stavových automatů určených pro klasické testování, byly v rámci simulátoru vyzkoušeny i soutěžní automaty ze soutěže Eurobot 2012. Při všech prováděných testováních byly simulovány překážky, kterým se robot musel vyhýbat.

V případě demo robotu, který provádí hledání předmětu (plechovky) v prostoru, byla také prováděna standardní simulace. Komplikací bylo, že kvůli kompletnímu otestování v simulátoru musel být upraven zdrojový kód pro tento automat. Problémem byla například chybějící zpětná vazba od kamery, která také provádí identifikaci plechovky nebo chybějící zpětná vazba od aktuátoru, kterým se plechovka nakládá. V momentě, kdy byly tyto problémy odstraněny, šlo již tento automat dobře otestovat ještě před použitím na reálném robotu.

Bohužel zde nešlo simulovat překážky, protože původní simulátor překážek zde slouží k simulaci umístění plechovky v prostoru. Naštěstí to není problém, protože oba stavové automaty, které slouží pro pohyb, jsou stejné.

6.2.2 Testování na reálném robotu

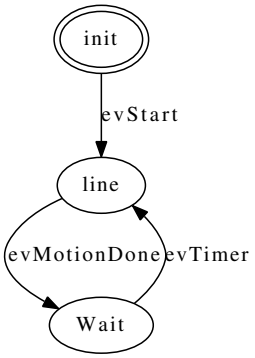
Stavové automaty byly testovány na obou robotech současně. Na demo robotu byl pouze otestován automat, který hledá plechovku v prostoru 5x5 metrů. Naopak na soutěžním robotu bylo provedeno testování pouze s testovacími automaty a ne se soutěžními. To není praktický problém, protože i v rámci těchto jednoduchých automatů se dá otestovat pohyb různými rychlostmi vpřed, pohyb vzad nebo zatáčení.

V případě demo robotu se vyskytlo několik problémů. První byl způsoben kompilací pro notebook a nikoli pro PowerPc. Ve zdrojovém kódu se nacházely části, které byly závislé na cílové platformě. Tím pádem nemohl být předmět korektně identifikován. Díky tomu se podařilo otestovat ostatní část stavového automatu, která se stará o náhodné prohledávání v prostoru. Díky tomu, že testování probíhalo v místnosti se stoly a mnoha dalšími překážkami, tak se podařilo úspěšně otestovat i vyhýbání překážkám. Po odstranění tohoto problému se ještě vyskytl problém s kamerou, kdy kamera nevracela žádná data a tím pádem nemohl být předmět řádně identifikován.

Po odstranění i tohoto problému byl úspěšně otestován celý automat. Robot opakovaně našel v prostoru plechovku a úspěšně ji i naložil. Na obrázku 6.3 je stavový diagram automatu pro demo robot, který provádí hledání plechovky. Obrázek je pochopitelně vygenerován nástrojem pro vizualizaci.

V případě soutěžního robotu se již žádné větší problémy nevyskytly, ale to bylo určitě způsobeno tím, že jako první bylo provedeno testování na demo robotu. V rámci této fáze se pro testování požívaly tedy pouze jednoduché automaty. V počátečních krocích byla z bezpečnostních důvodů nastavena výrazně pomalejší rychlost pohybu, aby nemohlo dojít k chybám. Postupně při úspěšných pokusech byla rychlost zvyšována.

Pro ilustraci je uvedena ukázka z logování událostí a stavů, které bylo používáno při testování. Úvodní výpisy jsou pro větší názornost ještě okomentovány. Ve finální verzi SW již tyto výpisy nejsou k dispozici. Logování odpovídá stavovému automatu, který implementuje strategii pro pohyb po čáře. Nejprve vpřed a poté vzad. Na obrázku 6.4 je stavový diagram tohoto automatu.



Obrázek 6.4: Hlavní automat z demo robotu

```
Processor creation. // vytvoření hlavního stavového automatu
FSMMain // zavolán konstruktor hlavního stavového automatu
Processor creation. // vytvoření stavového automatu pro pohyb
We are VIOLET! // informace o nastavené soutěžní barvě
FSMMain - init // aktuální stav hlavního automatu
FSMMain event evStart // událost, která přišla do hlavního automatu
FSMMotion - wait_for_command // aktuální stav pohybového stavového automatu
FSMMain - line // aktuální stav hlavního automatu
FSMMain event evMotionDone // událost, která přišla do pohybového automatu
FSMMotion event evNewTarget
FSMMotion - movement
FSMMotion event evTrajectoryDone
FSMMotion - wait_for_command
FSMMain - Wait
FSMMain event evTimer
FSMMain - line
FSMMain event evMotionDone
FSMMotion event evNewTarget
FSMMotion - movement
FSMMotion event evTrajectoryDone
FSMMotion - wait_for_command
FSMMain - Wait
```

I v tomto případě se při testování objevila chyba, která byla způsobená rozdílnými chybami levé a pravé odometrie. Z tohoto důvodu robot mírně zatáčel vpravo v případě pohybu rovně vpřed. S touto chybou se dá jednoduše vypořádat korekcí dat z odometrie (vynásobením potřebnými konstantami). Jelikož se tento problém na robotu vyskytuje, tak nebylo třeba tyto konstanty hledat. Po korekci se robot již pohyboval správně. Kontrola nad pohybem se prováděla nejen vizuálně, ale i s pomocí robotického simulátoru, kde se daly porovnávat naměřená odometrická data s referenční pozicí.

6.3 Testování nástroje pro vizualizaci

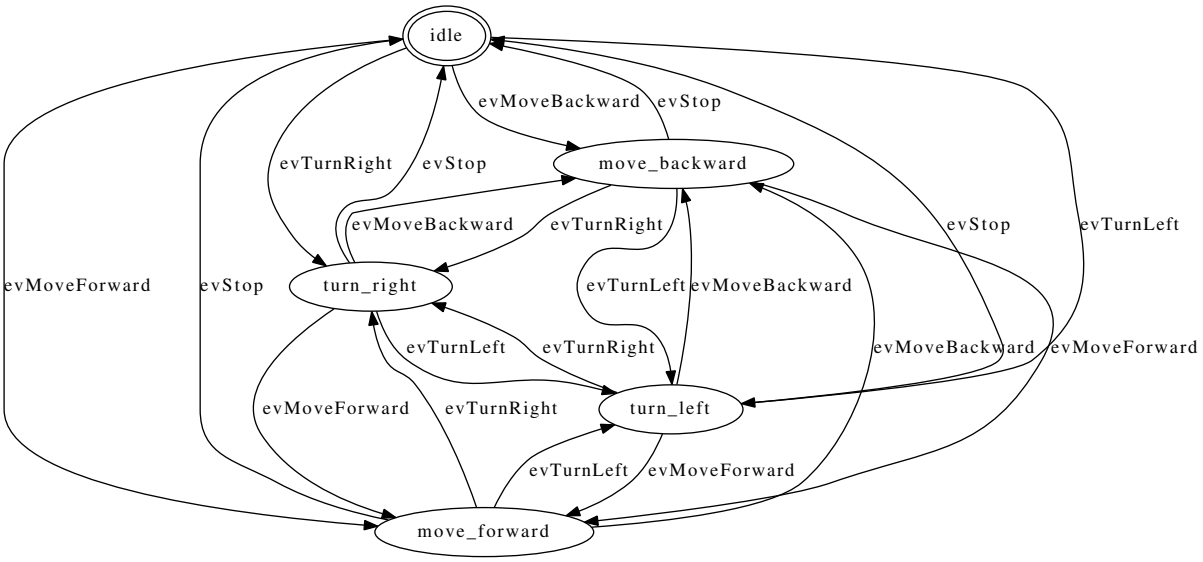
Tato část byla prováděna průběžně, protože při přepisu stavových automatů byly průběžně generovány jejich stavové diagramy pro kontrolu správnosti přechodů. Samozřejmě se tímto způsobem nedají otestovat přidané schopnosti. Takže pro tyto účely byly ve stavových automatech vytvořeny chyby tak, aby se otestovala jejich detekce.

Při testování vizualizačního nástroje se nevyskytovaly žádné problémy, které by znemožňovaly správné fungování programu. Ukázky výstupu programu jsou viditelné v průběhu celé

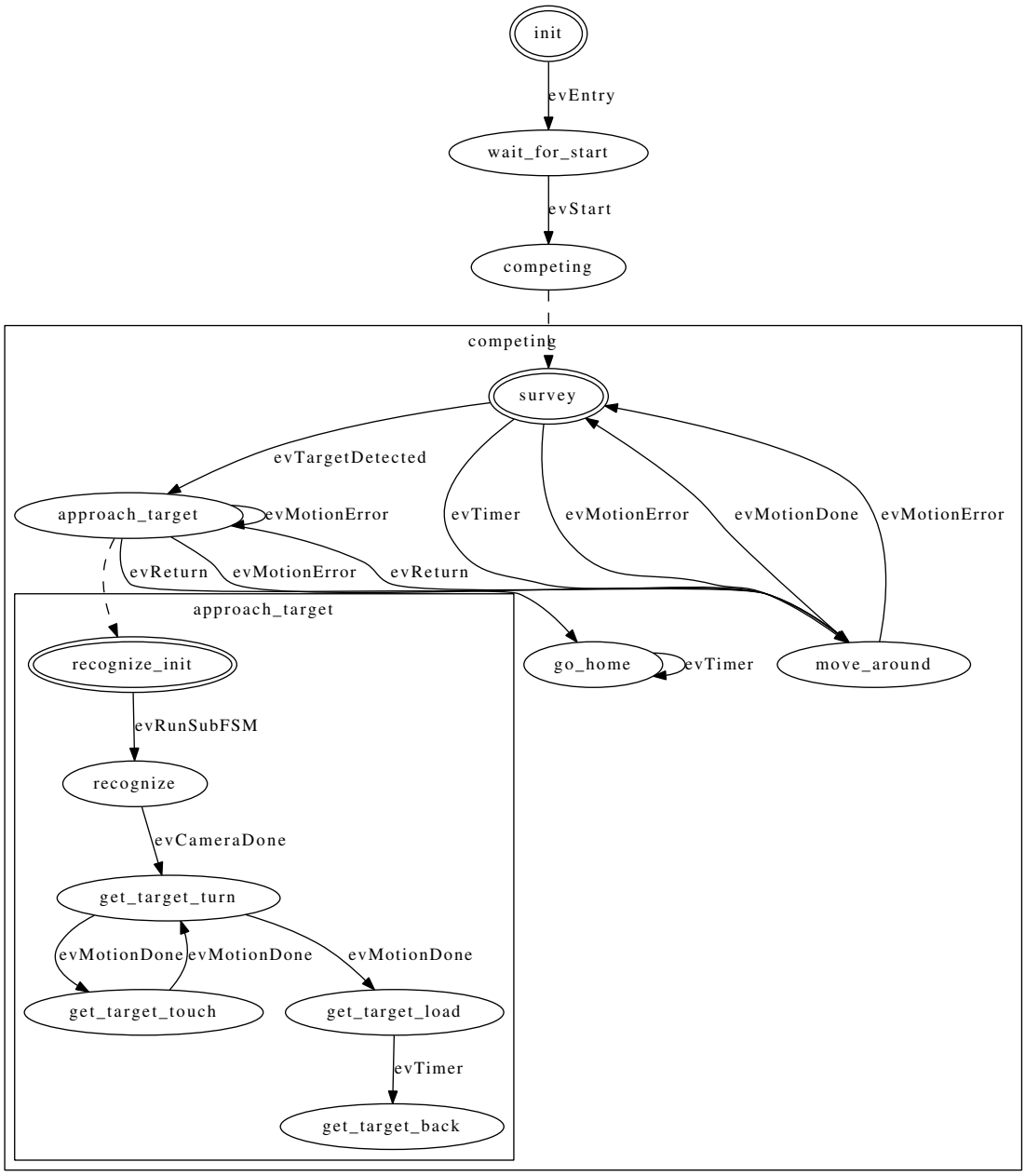
diplomové práce. Speciálně pro testování vizualizačního nástroje byl vytvořen stavový automat, s jehož pomocí lze demonstrovat prakticky veškeré chyby, které je program schopen úspěšně detekovat. Samozřejmostí bylo i testování na zdrojových souborech, kde se nenacházel žádný stavový automat. I v tomto případě se program choval přesně tak, jak měl a nenahlásil žádnou chybu.

Podařilo se otestovat některé stavové automaty, které jsou distribuovány jako příklady spolu s knihovnou. Obrázky jejich stavových diagramů lze nalézt na webových stránkách nástroje nebo na přiloženém CD. Jedná se pouze o stavové automaty, které využívají synchronní stavové automaty z knihovny Boost Statechart. Příklad, který obsahuje asynchronní automaty, obsahuje také dva stavové automaty, které si mezi sebou posílají události a reprezentují tak zápas ve stolním tenisu. Vizualizátor neumí vizualizovat více stavových automatů v jednom souboru. Z tohoto důvodu byl vytvořen jednoduchý příklad, který využívá knihovní implementaci asynchronních automatů. Stavový diagram je na obrázku 6.5.

Uvnitř tohoto stavového automatu se neposílají události. Ty se do automatu dostávají pouze z vnějšku. Sice je v tomto případě použití asynchronních stavových automatů zbytečné a lze tento automat naprogramovat i s pomocí synchronních automatů a oba se budou chovat naprosto stejně. Stavový automat reprezentuje stav pohybu robotu. Ale pokud se použije stavový automat pro kompletní řízení robotu, tak tento automat nebude jediným a objeví se v něm i posílání událostí o vykonaném pohybu do ostatních stavových automatů.



Obrázek 6.5: Asynchroní stavový automat



Obrázek 6.3: Hlavní automat z demo robotu

7 Závěr

V rámci diplomové práce bylo provedeno nahrazení knihovny stavových automatů používané pro řízení robotů v robotickém týmu Flamingos. Kromě knihovny stavových automatů se úpravy týkaly i ostatních částí řídicího SW.

V souvislosti s tím, že v rámci bakalářské práce jsem vytvářel nástroj pro vizualizaci stavových automatů pro knihovnu Boost Statechart, byla právě tato knihovna vybrána jako náhrada původní. Součástí diplomové práce byly také úpravy v nástroji na vizualizaci.

Do knihovny Boost Statechart jsem navrhl a posléze doimplementoval podporu pro časovače a upravil řešení problémů souběhu a zpracování událostí. Veškeré změny jsou plně kompatibilní s ostatními částmi knihovny a používají se naprosto stejně. Je tedy na programátorovi, pro kterou variantu se nakonec rozhodne.

Navrhl jsem úpravy ve struktuře řídicího systému a implementoval je. Jednalo se pouze o změny ve struktuře. V samotném kódu moc změn nebylo provedeno, aby se zachovala původní funkcionalita. Při přepisu byly využity možnosti, které přináší jazyk C++ oproti jazyku C.

Do nástroje pro vizualizaci stavových automatů byly doplněny zejména funkce soustřeďující se na analýzu zdrojového kódu ve smyslu logiky ve stavových automatech. Informaci o aktualizaci nástroje a doplněných funkcích jsem přidal i na webové stránky nástroje. Nástroj byl již v rámci mé bakalářské práce šířen i mezi další vývojáře používající knihovnu Boost Statechart.

Testováním nejprve prošly úpravy knihovny Boost Statechart. Po jejich úspěšném otestování došlo na kompletní řídicí SW a to nejprve v simulátoru a poté úspěšně i na reálných robotech. V průběhu celého testování řídicího SW byly vytvářeny stavové automaty, takže testování nástroje na vizualizaci bylo prováděno také průběžně. Kromě těchto automatů byly pro testování používány automaty, ve kterých byly úmyslně vytvořeny chyby.

Příloha A –Obsah CD

/text	elektronická verze této práce ve formátu pdf
/src	zdrojové kódy pro vizualizátor a zdrojové kódy rozšiřující knihovnu Boost Statechart. Ostatní zdrojové kódy, které jsou součástí řídicího SW, jsou umístěny v repozitáři robotického týmu Flamingos v mé personální větvi personal/silhape2/boost. Přístup k repozitáři lze získat u vedoucího práce.
/test_examples	ukázkové příklady stavových automatů včetně vygenerovaných obrázků a souborů Makefile
/docs	dokumentace upravených částí knihovny v pdf. Rovněž je zde umístěn soubor s návodem, jak používat rozšiřující část knihovny. Návod i dokumentace jsou psány v Anglickém jazyce.
/UML_diagrams	soubory s UML diagramy, které jsou použity v této diplomové práci.

Literatura

[Alg07] Algoritmy.net. *Insertion Sort*. navštíveno: 07.04.2013. 2007. URL: http://www.algoritmy.net/article/8/Insertion-sort.

[Apa13] Apache. *Commons SCXML*. navštíveno: 02.04.2013. 2013. URL: http://commons.apache.org/proper/commons-scxml/guide.html.

[Boh10a] J. Bohren. *Dokumentace ke knihovně smach*. navštíveno: 20.03.2013. 2010. URL: http://www.ros.org/wiki/smach/.

[Boh10b] J. Bohren. *Dokumentace programu smach_viewer*. navštíveno: 20.03.2013. 2010. URL: http://ros.org/wiki/smach_viewer/.

[Cha07] A. J. Champandard. *Hierarchické automaty nebo subautomaty*. navštíveno: 14.03.2013. 2007. URL: http://aigamedev.com/open/article/hierarchical-or-nested-fsm/.

[Dar03] A. Darovsky. *Dokumentace projektu FSME*. navštíveno: 20.03.2013. 2003. URL: http://fsme.sourceforge.net/.

[Dic13] Dictionary.com. *Online Etymology Dictionary*. navštíveno: 01.05.2013. 2013. URL: http://dictionary.reference.com/browse/automaton.

[Dou99] B. P. Douglass. „UML statecharts“. In: *Embedded systems programming* 12.1 (1999), pp. 22–42. URL: http://www.eetimes.com/ContentEETimes/Documents/Embedded.com/1999/9901/f-dougla.pdf.

[Dre11] D. Drelll. *Visual State Chart Editor*. navštíveno: 02.04.2013. 2011. URL: http://www.davidwdrell.net/wordpress/?page_id=81.

[DP04] A. Drumea et al. „Finite state machines and their applications in software for industrial control“. In: *Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 2004. 27th International Spring Seminar on*. Vol. 1. 2004, 25–29 vol.1. DOI: 10.1109/ISSE.2004.1490370.

[Gov08] D. Gove. *Cena mutexu*. navštíveno: 30.04.2013. 2008. URL: https://blogs.oracle.com/d/entry/the_cost_of_mutexes.

[Hee13] F. Heemn. *Dokumentace projektu StateForge*. navštíveno: 20.03.2013. 2013. URL: http://www.stateforge.com/.

[HD07] A. Huber Dönni. *Dokumence ke knihovně Boost Statechart*. navštíveno: 13.03.2013. 2007. URL: http://www.boost.org/doc/libs/1_53_0/libs/statechart/doc/index.html.

[Jar10] F. Jareš. „Implementace stavových automatů pro soutěž Eurobot 2009“. Bakalářská práce. 2010. URL: http://rtime.felk.cvut.cz/~sojka/students/Bp_2010_jares_filip.pdf.

[Kub10] J. Kubias. „Hardware robota pro soutěž Eurobot“. Diplomová práce. 2010. URL: http://rtime.felk.cvut.cz/~sojka/students/Dp_2010_kubias_jiri.pdf.

[Kr06] M. Křetínský et al. „Formální jazyky a automaty I“. In: (2006). navštíveno: 01.05.2013. URL: http://is.muni.cz/elportal/?id=703389.

[Lr13] Ch. Lattner et al. *LLVM a Clang dokumentace*. navštíveno: 02.05.2013. 2013. URL: http://llvm.org.

[Ně10] M. Němec. *UML diagramy tříd*. navštíveno: 02.05.2013. 2010. URL: http://www.milosnemec.cz/clanek.php?id=199.

[Qt12] Qt. *Qt SCXML engine*. navštíveno: 02.04.2013. 2012. URL: http://qt.gitorious.org/qt-labs/scxml.

[Rox13] J. Roxendal. *PySCXML*. navštíveno: 02.04.2013. 2013. URL: https://github.com/jroxendal/PySCXML#readme.

[Sal+10] D.O. Sales et al. „Vision-Based Autonomous Navigation System Using ANN and FSM Control“. In: *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*. 2010, pp. 85–90. DOI: 10.1109/LARS.2010.26.

[She11] M. Shead. *State Machines – Basics of Computer Science*. navštíveno: 02.05.2013. 2011. URL: http://blog.markshead.com/869/state-machines-computer-science/.

[Smo+12] P. Smolík et al. *Dokumentace k projektu ORTE*. navštíveno: 03.03.2013. 2012. URL: http://orte.sourceforge.net/orteman.pdf.

[Soj11] M. Sojka. *Dokumence ke knihovně FSM*. navštíveno: 13.03.2013. 2011. URL: http://rtime.felk.cvut.cz/dragons/doc/group__fsm.html.

[Vok12] M. Vokáč. „Demonstrační robotická platforma“. Diplomová práce. 2012. URL: http://rtime.felk.cvut.cz/~sojka/students/Dp_2012_vokac_michal.pdf.

[W3C12] W3C. *State Chart XML - verze z 6.12.2012*. navštíveno: 02.04.2013. 2012. URL: http://www.w3.org/TR/2012/WD-scxml-20121206/.

[Ša11] J. Šach. „Vývoj elektroniky pro mobilní roboty“. Diplomová práce. 2011. URL: http://rtime.felk.cvut.cz/~sojka/students/Dp_2011_sach_jaroslav.pdf.

[Ši11] P. Šilhavík. „Vizualizace stavových automatů pro řízení robotů“. Bakalářská práce. 2011. URL: https://support.dce.felk.cvut.cz/mediawiki/images/5/55/Bp_2011_silhavik_petr.pdf.