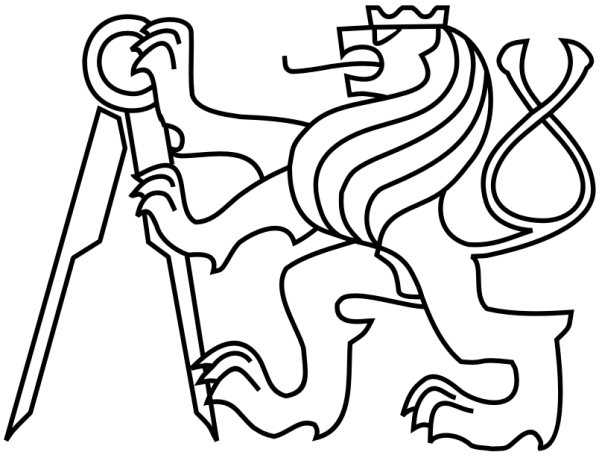
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ

KATEDRA ŘÍDICÍ TECHNIKY



BAKALÁŘSKÁ PRÁCE

Vizualizace stavových automatů pro řízení robota

Autor: Petr Šilhavík

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

Praha, 2011

Poděkování

Chtěl bych poděkovat vedoucímu Ing. Michalu Sojkovi, Ph.D. za jeho vstřícnost a vždy podnětné připomínky a rady. Dále bych chtěl poděkovat svým rodičům a nejbližším za jejich podporu při vypracování této bakalářské práce.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 27.5.2011

Petr Šilhavík

.....

podpis

Abstrakt

Tato práce se zabývá vizualizací stavových automatů. Program umí vizualizovat stavové automaty implementované v C++ pomocí knihovny Boost-Statechart. Výsledkem vizualizace je výstupní soubor ve formátu dot, který je možné vizualizovat a získat tak stavový diagram příslušející danému stavovému automatu. Uživatel dostane k dispozici i přechodovou tabulku, ale pouze vypsanou na standardní výstup. Na závěr jsou ještě vypsány statistické informace o diagnostice během zpracování souboru. Pro realizaci programu bylo použito LLVM a Clang, které umožňují provádět rychlou analýzu zdrojových kódů.

Abstract

This thesis deals with the visualizations of state machines. The program is able to visualize state machines that are implemented in C++ using Boost-Statechart library. The output of the visualization is file that can be visualized with dot and gain this way the state diagram of this state machine. The user gets also the transition table but only printed on standard output. And finally the diagnostical statistics are being printed. To implement the program was used LLVM and Clang that allow rapid source code analysis.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Petr Šilhavík**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **Vizualizace stavových automatů pro řízení robotů**

Pokyny pro vypracování:

1. Seznamte se s řídicím softwarem robotu pro soutěž Eurobot, který je z velké části implementován jako stavové automaty. Dále se seznamte s C++ knihovnami Boost/Statechart a LLVM.
2. Proveďte rešerši existujících open source řešení pro vizualizace stavových automatů.
3. S pomocí knihoven LLVM a clang vytvořte program, který ze zdrojových kódů používajících knihovnu statechart vygeneruje obrázek znázorňující strukturu stavového automatu.
4. Seznamte s vašimi výsledky open source komunitu okolo knihoven Boost.
5. Vše důkladně zdokumentujte a otestujte.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Michal Sojka

Platnost zadání: do konce zimního semestru 2011/2012

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 11. 1. 2011

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tato práce se zabývá vizualizací stavových automatů napsaných v jazyce C++. Cílem práce je tedy vytvořit program, který vygeneruje obrázek znázorňující strukturu daného stavového automatu na základě analýzy zdrojového kódu. Program provede vizualizaci stavových automatů implementovaných pomocí knihovny Boost-Statechart. Výstupem programu je soubor, který lze převést na obrázek pomocí programu dot[1] z projektu Graphviz.

Velkou motivací pro tuto bakalářskou práci je právě absence programu, který by provedl vizualizaci automatů implementovaných pomocí knihovny Boost-Statechart. Tím pádem chybí jednoduchá zpětná vazba, ve formě obrázku, pro programátora. Bez důkladného otestování nemá jistotu, že stavový automat naprogramoval tak, jak chtěl. Pokud tedy potřeboval diagram stavového automatu, musel si ho nakreslit ručně nebo ho právě popsat a vykreslit s pomocí programu dot. Kdyby měl k dispozici jednoduchý nástroj provádějící vizualizaci, mohl by případné chyby ve struktuře odhalit mnohem dříve než ve fázi testování.

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. Určitě je důležité zmínit i robotiku, kde se stavové automaty používají na řízení robotů.

Na základě analýzy existujících nástrojů pro analýzu kódu, bylo rozhodnuto, že pro vytvoření programu bude použito LLVM (Low Level Virtual Machine) a Clang, což jsou nástroje, které umožňují vytvářet nové překladače a tím i provádět s pomocí jejich knihoven rychlou analýzu zdrojového kódu. Právě na základě informací z této analýzy bude vytvořen výstupní soubor.

Práce je strukturována do několika částí. Ve 2. kapitole jsou podrobněji vysvětleny základní pojmy z oblasti stavových automatů, ve 3. kapitole jsou uvedeny výsledky rešerše již existujících řešení, ve 4. kapitole jsou specifikovány základní požadavky pro vizualizaci stavových automatů. V 5. kapitole je jednoduše vysvětlen pojem abstract syntax tree, někdy také nazývaný syntaktický strom, který je použit při analýze zdrojového kódu, v 6. kapitole jsou popsány knihovny použité pro vytvoření programu, v 7. kapitole je podrobně zdokumentován samotný program, 8. kapitola obsahuje výsledky testování a v 9. kapitole lze nalézt krátký výhled do dalšího rozvoje tohoto programu. V 10. kapitole se pak nachází závěr celé práce.

[1] **Dot:** Nástroj na automatické kreslení grafů popsaných jednoduchým kódem

2 Stavové automaty

V této kapitole lze nalézt vysvětlení několika základních pojmů týkajících se stavových automatů a to zejména konečných stavových automatů (2.1). Dále jsou zde zmíněny praktické aplikace stavových automatů, například jejich použití při řízení robotů (2.2).

V závěru této kapitoly jsou zmíněny různé metody pro zobrazování struktur a vizualizaci stavových automatů (2.3).

2.1 Konečné stavové automaty

Aby bylo jasné, odkud se vůbec takový automat vlastně vzal, je vhodné zmínit, že slovo automat pochází z řeckého slova *automatos*, což znamená samohybný. Tento význam si toto slovo zachovalo dodnes [9]. Veškeré dále uvedené informace v této části jsou citovány z [4] 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. Kromě informace o aktuálním stavu nemá potřebu si udržovat další informace, ale tím není tato možnost z principu úplně vyloučena.

Definice: Konečný automat (Finite Automaton, FA) M je pětice $(Q, \Sigma, \delta, q_0, F)$, 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 pro řízení robotů se můžeme obvykle setkat s trochu jiným názvem a to Konečný stavový automat (Finite State Machine = FSM).

Z hlediska HW (hardwarové) struktury automatu lze rozeznávat dva typy automatů a to Mooreův a Mealyho automat. Tyto automaty se liší tzv. výstupní funkcí ω.

1. **Mooreův automat**: $\omega : Q \rightarrow F$ Výstup tohoto automatu je tedy o 1 krok opožděn oproti stejnému Mealyho automatu, protože v počátku neexistuje žádný vnitřní stav.

2. **Mealyho automat**: $\omega : Q\,x\,\Sigma \rightarrow F$ Výstupem je tedy přímo vnitřní stav automatu nebo kombinace vnitřních stavů. Toto zobrazení může být přímo přechodovou funkcí δ.

Toto dělení, ale nemá vliv na zobrazení struktury stavového automatu, protože nás zajímá pouze parciální přechodová funkce. Výstup se neprojeví. Automaty vykonávající stejný úkol lze tedy nejjednodušeji rozpoznat měřením jejich výstupu. Pokud by byl výstup automatu o 1 krok opožděn oproti druhému, pak by se jednalo o Mooreův automat. Druhý automat by patřil do skupiny Mealyho automatů. Podrobnější informace o tomto dělení lze nalézt například v [6].

2.2 Aplikace stavových automatů

Se stavovými automaty se lze setkat téměř ve všech oborech lidské činnosti. I samotné lidské chování by se, v určitých situacích, dalo považovat za stavový automat. Například ucuknutí ruky při dotyku horkého místa. Mozek zareaguje na událost zvýšení tepla oddálením ruky. Samozřejmě jejich nejširší uplatnění najdeme v oblasti průmyslu a informatiky.

Jak jsem již uvedl, stavové automaty jsou matematickým modelem pro výpočetní procesy, a proto se velmi často používají pro zpracování regulárních výrazů[2] v překladačích. Automaty v tomto případě fungují tak, že postupně čtou symboly ze vstupu a přecházejí do stavů, kde událost inicializující přechod je právě přečtený symbol. Pokud automat po přečtení celého slova skončí ve stavu, který patří do množiny koncových stavů, pak automat dané slovo přijal. V opačném případě ho nepřijal. Všechna slova, která je automat schopen přijmout tvoří tzv. regulární jazyk [4].

Další velmi častou aplikací stavových automatů je popis výrobních a technologických procesů, jelikož většinu z nich lze popsat pomocí stavů a přechodů mezi nimi. Automaty se v tomto prostředí velmi rychle rozšířily, protože většina technologických procesů byla v minulosti postavena před automatizaci svého provozu. Tím došlo ke snížení nákladů na lidskou pracovní sílu a zvýšení efektivity výroby. Výroba s pomocí automatizovaných systémů je totiž mnohem rychlejší, efektivnější a velmi často šetří i náklady na výrobní materiál, protože automat pracuje spolehlivěji než člověk.

V neposlední řadě je důležité zmínit řízení robotů, kde se využívá nejčastěji událostmi řízený stavový automat. Senzory na robotu vytvářejí události a robot na ně obvykle reaguje přechody mezi stavy. Pro naprogramování takového automatu stačí vytvořit stavy, události a funkce, které popíší, jak se má na dané události zareagovat. Pro popis obyčejného pohybu v prostoru si programátor vystačí se stavy stop a jízda vpřed. S událostmi otoč vpravo/vlevo, zastav a rozjeď se. Velkou výhodou této metody je tedy přehlednost kódu, jelikož potřebujeme pouze stavy a události. Samozřejmě ještě větší přehlednosti kódu lze dosáhnout použitím

[2] **Regulární výraz** - řetězec popisující celou množinu řetězců

nástrojů na vizualizaci stavových automatů. V neposlední řadě mezi výhody patří možnost jednoduchého testování. Bližší informace o této metodě lze nalézt v [10].

2.3 Zobrazování struktur stavových automatů

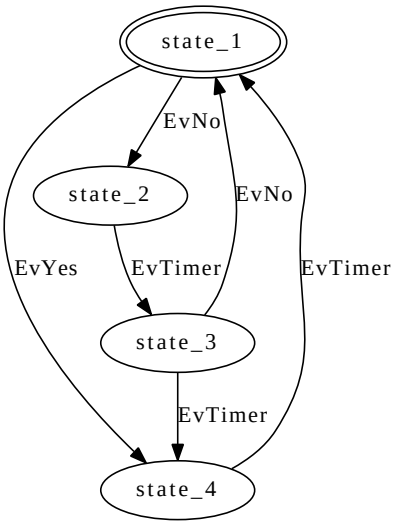
V této části bude ukázáno několik způsobů zobrazení struktur stavového automatu. Pro porovnání těchto metod bude použit jednoduchý stavový automat, jehož matematický zápis vypadá takto: $M = (Q, \Sigma, \delta, q_0, F)$, kde:

- $Q = \{state\_1, state\_2, state\_3, state\_4\}$
- $\Sigma = \{EvYes, EvNo, EvTimer\}$
- $\delta(state\_1, EvYes) = state\_4 \quad \delta(state\_1, EvNo) = state\_2$
  $\delta(state\_2, EvTimer) = state\_3 \quad \delta(state\_3, EvNo) = state\_1$
  $\delta(state\_3, EvTimer) = state\_4 \quad \delta(state\_1, EvTimer) = state\_1$
- $q_0 = state\_1$
- $F = Q$

Jelikož stavový automat je matematický model pro výpočetní procesy, tak prvním zobrazením struktury a přechodů ve stavovém automatu je právě přechodová funkce a počáteční stav. Toto zobrazení je naprosto matematicky korektní, ale nedá programátorovi rychlou a jasnou představu o struktuře stavového automatu. Zobrazením do grafu dostaneme další možnost a to stavové diagramy.

$$q_0 = state\_1$$
$$\delta(state\_1, EvYes) = state\_4 \quad \delta(state\_1, EvNo) = state\_2$$
$$\delta(state\_2, EvTimer) = state\_3 \quad \delta(state\_3, EvNo) = state\_1$$
$$\delta(state\_3, EvTimer) = state\_4 \quad \delta(state\_1, EvTimer) = state\_1$$

Pro zobrazování struktur stavových automatů se asi nejčastěji používá stavový diagram, někdy také nazývaný přechodový diagram. Z něj jsou patrné všechny stavy, ve kterých se automat může nacházet, a přechody mezi stavy inicializované pomocí událostí. Počáteční stav je obvykle vyznačen dvojitou čárou. U šipek, které značí přechody mezi stavy, jsou popisy, které specifikují událost, při které se tento přechod provede. Obrázek znázorňující strukturu stavového automatu je na obrázku 2.1.



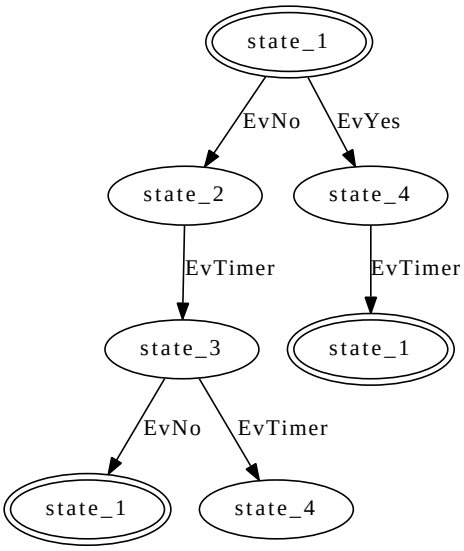
Obrázek 2.1: Stavový diagram vygenerovaný programem

Strukturu stavového automatu lze také zobrazit pomocí přechodové tabulky. V ní se v řádcích nachází jednotlivé stavy a v záhlaví sloupců události. Cílový stav se poté nalezne na křížení sloupce příslušné události a současného stavu. Počáteční stav je obvykle označen s použitím znaku *. Tato metoda patří právě spolu se stavovým diagramem mezi nejpoužívanější.

Tabulka 2.1: Přechodová tabulka

		EvYes	**EvNo**	**EvTimer**
*	**state_1**	state_4	state_2	-
	state_2	-	-	state_3
	state_3	-	state_1	state_4
	state_4	-	-	state_1

Nakonec je vhodné zmínit reprezentaci stavovým stromem. Zde je kořenem stromu počáteční stav, takže není třeba ho označovat. Toto zobrazení se hodí pouze pro automaty bez stavů se subautomaty. Pokud bychom ztotožnili stavy se stejným názvem, dostali bychom stavový diagram. Strom se tedy dá jednoduše vytvořit pouhým odstraněním kružnic ze stavového diagramu. Jediným omezením pro tuto metodu je, že všechny stavy musí ležet ve stejném kontextu neboli nesmí obsahovat subautomaty. Oproti stavovému diagramu má obvykle větší počet uzlů, které reprezentují stavy, ale počet hran reprezentující přechody mezi stavy je totožný. Příklad stavového stromu je na obrázku 2.2.



Obrázek 2.2: Stavový strom

3 Podobné projekty

V této části se nachází popis několika programů, které se zabývají vizualizací stavových automatů.

Na internetu se nenachází téměř žádný program, který by se zabýval automatickou vizualizací stavových automatů. Existuje samozřejmě mnoho programů pokoušejících se o vizualizaci, ale téměř žádný není schopen provést tuto vizualizaci ze zdrojového kódu. Samozřejmě existují i nástroje, které používají opačný model práce, takže umožňují generování kódu z obrázku.

Jelikož se tato práce zabývá právě vizualizací ze zdrojového kódu, proto jsou zde uvedena pouze řešení pro vizualizaci ze zdrojových kódů napsaných v různých programovacích jazycích. A na závěr je uveden nástroj, který umí nejenom generovat kód, ale i vizualizovat stavový automat RT (Real time), tedy během toho, co je daný automat spuštěn. Všechny nástroje zde zmíněné jsou Open Source.

Na základě těchto nalezených řešení a zejména na řešení pro knihovnu Boost-Statechart byl zkonstruován seznam požadavků na tento program.

3.1 XFSM

Typickým příkladem vizualizace z vlastního kódu je XFSM [5]. Tento nástroj je schopný vykreslit stavové automaty v souboru FSM, jenž je napsán v jazyce VHDL. Program bohužel neumí pracovat s jiným formátem souboru. Vývoj tohoto nástroje sice stále pokračuje, ale soubory se stavovými automaty, napsané s pomocí knihovny Boost-Statechart, jsou v jazyce C++. Toto řešení je tedy prakticky nevyužitelné. Jedinou možností, jak ho efektivně využít pro knihovnu Boost-Statechart, by byl překlad z C++ do jazyka VHDL, což by bylo zbytečně komplikované.

3.2 Boost-Statechart viewer

Jediným nalezeným řešením pro automatickou vizualizaci stavových automatů napsaných s pomocí knihovny Boost-Statechart je projekt nabízený v roce 2006 v rámci Google Summer of Code založený na GCC-XML, což je rozšíření pro překladač gcc, které provede převod struktury programu napsaného v jazyce C++ do XML. Informace k tomuto projektu lze nalézt

v [8], nebo na přiloženém CD ve složce emails. Bohužel vývoj tohoto nástroje se zastavil. Poslední vydaná verze 0.6 je dokonce ještě starší, z roku 2004. Vývoj tohoto nástroje byl přímo zastřešen komunitou kolem knihoven Boost, takže některé z požadavků projektu byly zakomponovány do požadavků na program, který je výsledkem této bakalářské práce. Jiný projekt specializující se na knihovnu Boost-Statechart se nepodařilo najít, ale to nevyvrací možnost, že nějaký existuje.

3.3 FSME

Tento nástroj patří sice do skupiny generátorů kódu pro stavové automaty z vytvořeného obrázku. Celý projekt se skládá ze 3 nástrojů. Prvním nástrojem je FSME, které reprezentuje grafický editor stavových automatů. Dále sem patří FSMC, kompilátor, který z XML, které je výstupem FSME, vygeneruje zdrojový kód pro jazyk C++ nebo Python. A jako poslední FSMD, který je schopen provádět RT vizualizaci běžícího stavového automatu. Jako na každém nástroji se i zde dají nalézt nějaké nevýhody a to je hlavně žádná klasická non RT vizualizace, tedy neschopnost vytvoření například stavového diagramu bez běžícího automatu. Oproti tomu RT vizualizací získáte opravdu silný nástroj na debuggování stavových automatů. Generované zdrojové kódy s pomocí FSMC nepotřebují k běhu žádnou speciální knihovnu. Výsledkem vlastního generování jsou hlavičkové soubory s definicí automatu. Poslední dostupná verze 1.0.4 je sice z roku 2006, ale poslední vylepšení zdrojových kódů proběhlo v roce 2009. Bližší informace o tomto nástroji lze nalézt na internetových stránkách programu FSME [7].

4 Požadavky na program

V této části jsou specifikovány základní požadavky na program. Požadavky jsou uvedeny v pořadí, které symbolizuje jejich důležitost pro vizualizaci stavových automatů. U každého požadavku je uveden krátký popis, který vysvětluje, co se pod tímto požadavkem skrývá.

- **Stavy** – nalezení a vykreslení stavů
- **Přechody** – nalezení a vykreslení přechodů mezi stavy
- **Subautomaty** – korektní vykreslení subautomatů
- **Události** – nalezení všech událostí
- **Přechody mezi vlastními reakcemi** – nalezení přechodů mezi vlastními reakcemi na události
- **Ortogonální stavy** – nalezení a vykreslení ortogonálních stavů (stavů, které obsahují více paralelně běžících automatů)
- **Ostatní vlastní reakce** – vykreslení ostatních vlastních reakcí (zahazování událostí, přeposílání událostí, ...)

5 Abstract syntax tree

Tato část vysvětluje pojem z oblasti teoretické informatiky a to abstract syntax tree (5.1), česky též nazývaný syntaktický strom. Tento strom je použit při analýze zdrojového kódu a vyhledávání stavů a přechodů.

V závěru jsou zmíněny některé z aplikací AST (Abstract syntax tree) a to nejen při analýze a překladu zdrojových kódů (5.2).

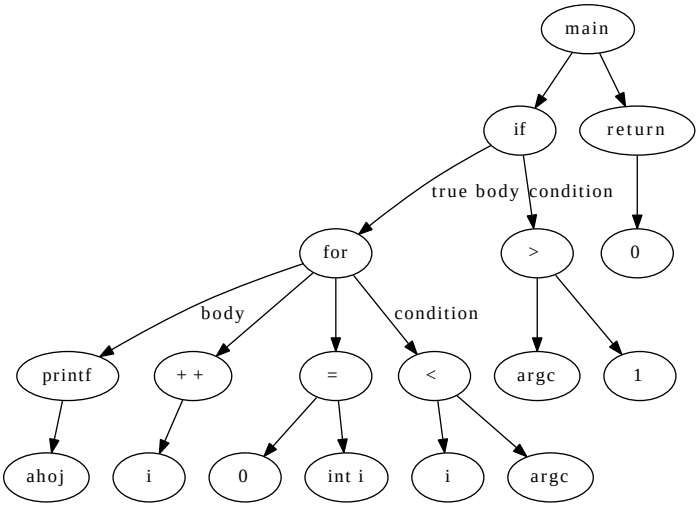
Jelikož by v práci došlo ke kolizi stejných zkratek pro AST jako abstract syntax tree a *AST* jako knihovny Clang, je název knihovny *AST* vždy odlišen kurzívou.

5.1 AST

AST je strom, který je produktem syntaktického analyzátoru. Zobrazuje syntaktickou strukturu zdrojového kódu. Každému programu patří právě jeden AST, takže neplatí, že by každý soubor měl svůj vlastní AST. Z hlediska struktury se jedná o strom, kde kořenem je program a v listech jsou proměnné nebo konstanty. Hlavním rozdílem od parse tree je to, že AST zobrazuje pouze strukturu, nenalezneme v něm například závorky. To je také důvod, proč se nazývá abstraktní. Například pro if v AST nalezneme 3 podstromy: podmínku, true větev a false větev. V případě parse tree bychom na počátku všech podstromů našli závorky, ale zde dostaneme přímo kód, který se nachází uvnitř závorek.

Rovněž v AST nalezneme informace o konverzích proměnných a samozřejmě z hlediska funkcí i o parametrech a jejich návratových hodnotách. Bohužel z hlediska struktury AST nelze přímo rozeznat např. dědičnost pro případ objektově orientovaných jazyků. Takové informace lze reprezentovat pomocí vlastností jednotlivých uzlů. Ukázku AST včetně příslušného zdrojového kódu lze nalézt na obrázku 4.1.

```
int main(int argc, char **argv)
{
    if(argc>1)
    {
        for(int i = 0; i<argc; i++)
        {
            printf("ahoj\n");
        }
    }
    return 0;
}
```



Obrázek 5.1: Zdrojový kód a jemu příslušející AST

5.2 Aplikace AST

V úvodu této části jsem zmínil, že hlavní aplikaci pro AST najdeme v syntaktické analýze a optimalizaci překladu zdrojových kódů. Jak jsem již uvedl, AST je výstupem syntaktického analyzátoru. Takový AST se postupně doplňuje informacemi o konverzích a s pomocí všech těchto informací provádí optimalizátor v překladači optimalizaci před překladem do strojového kódu.

Další oblastí pro aplikaci AST lze uvést optimalizaci při interpretaci. Kdy s pomocí AST si může interpret ulehčit práci tím, že nemusí např. vyhodnocovat celý booleovský výraz, když jeho část je vždy pravdivá. Rovněž při interpretaci udržuje informace o struktuře programu a vztazích mezi různými konstrukcemi. Tím udržuje mnohem podrobnější informace o programu než obvykle používaný bytekód. V případě použití Just-in-Time[3] interpretů umožňuje provádět i mnohem podrobnější a jednodušší analýzu a optimalizaci za běhu programu.

[3] **Just-in-Time:** Interpretace s použitím mezikódu (nativní strojový kód). Úsek kódu se interpretuje až tehdy, když je opravdu potřeba. Odtud název just in time.

6 Použité knihovny

V této části budou postupně představeny tři základní knihovny, které jsou v této práci použity. V prvé řadě se jedná o knihovnu Statechart (6.1), která umožňuje realizovat stavové automaty v jazyce C++. Dále o nástroj LLVM (6.2), který poskytuje infrastrukturu pro překladač a nakonec Clang (6.3), což je překladač, pro jazyk C/C++ umožňující vytváření vlastních překladačů. Tyto nástroje jsou v programu reprezentovány jednotlivými knihovnami.

Při vytváření programu je nejvíce používán právě Clang, jehož knihovny umožňují rychlou a snadnou analýzu kódu. Nejdůležitější knihovnou pro tuto analýzu bude *AST*, která dává programátorovi k dispozici silné nástroje pro analýzu kódu pomocí AST.

6.1 Boost-Statechart

Knihovna Statechart je jednou z mnoha knihoven, které se ukrývají v sadě s názvem Boost. Umožňuje implementovat stavové automaty v jazyce C++. Je vyvíjena komunitou sdružující se kolem celé sady knihoven Boost. Samotná knihovna Statechart je rozdělena do mnoha hlavičkových souborů, takže paměťová náročnost jednotlivých stavových automatů je omezena téměř na minimum. Programátor tedy opravdu využívá pouze tu část, kterou potřebuje. Další informace v této části jsou volně citovány z internetové dokumentace [1].

Aby bylo celé toto teoretické vysvětlení lépe pochopitelné, jsou průběžně vloženy v textu krátké ukázky aplikace popsané teorie do kódu.

Stavový automat napsaný v této knihovně vypadá z hlediska syntaxe asi takto. Nejprve je nutno definovat jednotlivé stavy automatu a události. V první části je nutné jednotlivé stavy pouze vypsat. Z pohledu datové reprezentace jsou stavy reprezentovány jako struktury nebo třídy. Pro všechny stavy je výhodnější použít struktury, protože zde není nutno nastavovat přístupová práva na public. Události jsou rovněž struktury, které dědí od template třídy event z knihovny statechart. Jediným parametrem takto vytvořené události je její identifikátor. Ještě než dojde na deklaraci jednotlivých stavů, je nutno nastavit počáteční stav automatu. To se provede definováním celého automatu, který je definován rovněž strukturou nebo třídou. Tato deklarace je potomkem template třídy state_machine, kde prvním parametrem je název automatu a druhým pak jeho počáteční stav.

```
namespace sc = boost::statechart;
// definice stavů
struct Active;
struct Stopped;
struct Running;
// definice automatu
struct StopWatch : sc::state_machine< StopWatch, Active > {};
```

V případě, že počátečním stavem je stav, který obsahuje také automat, tzv. subautomat. Jako název stavu se zadává stav, který je na vnější straně. Dále je nutno nastavit počáteční substav. To se provede až při deklaraci vnějšího stavu jednoduchým přidáním třetího parametru do deklarace. V případě, že automat přejde do tohoto stavu, bude automaticky spuštěn tento definovaný počáteční substav.

Nyní je již možno přikročit k deklaraci jednotlivých stavů. Pro každý stav je definována pouze reakce na vstup do stavu reprezentovaná konstruktorem a na výstup ze stavu umožněná pomocí destruktoru. Důvodem pro toto použití je vytvoření objektu při vstupu do stavu a jeho zrušení při opuštění stavu. Jednotlivý stav je potomkem třídy simple_state nebo state, jejíž parametry jsou v tomto pořadí: název stavu, název kontextu stavu neboli název nadstavu. Eventuelně pro případ stavu se subautomatem je 3. parametrem název počátečního stavu subautomatu. Jak již bylo zmíněno je možno stav reprezentovat jako strukturu nebo jako třídu.

```
struct Active : sc::simple_state< Active, StopWatch, Stopped >
{
   Active() {} //konstruktor
   ~Active() {} //destruktor
};
```

Ve funkci main už pouze následuje vytvoření objektu pro stavový automat a spuštění celého automatu metodou initiate(). Tím se spustí počáteční stav.

Takto definovaný automat bohužel neumožňuje provádět přechody mezi stavy. K tomu slouží jednotlivé již definované události na počátku souboru. Pro jednoduchý přechod mezi stavy stačí přidat jednotlivé přechody řízené událostmi do příslušných stavů. Pokud potřebujeme, aby ze stavu existovalo více přechodů (cílové stavy by byly rozdílné), musíme všechny přechody dát do seznamu, který využívá knihovnu mpl, která patří rovněž do skupiny knihoven Boost. Použití tohoto seznamu je podmínkou, takže jiný seznam například ze STL nelze použít.

```
struct EvStartStop : sc::event< EvStartStop > {}; //definice události
struct Running : sc::simple_state< Running, Active >
{
   typedef sc::transition< EvStartStop, Stopped > reactions; //přechod mezi stavy
   Running() {} //konstruktor
   ~Running() {} //destruktor
};
```

Další možností přechodů je definování vlastních reakcí na události. Na to je potřeba vytvořit metodu, jejíž parametrem je reference na danou událost. Do této metody může programátor umístit například podmínkově závislý přechod, tedy přechod závisející na stavu nějaké proměnné. Tím vzniká další možnost pro přechod mezi stavy.

```
struct Stopped : sc::simple_state< Stopped, Active >
{
   typedef sc::custom_reaction< EvStartStop> reactions;
   Stopped() {} //konstruktor
   ~Stopped() {} //destruktor
   sc::result react( const EvStartStop & ) //metoda pro danou událost EvStartStop
   {
      return transit<Running>();
   }
};
```

Knihovna samozřejmě umožňuje vytvářet i složitější automaty, které obsahují posílání událostí mezi stavy, ortogonální stavy, uchovávání stavových informací, zahazování událostí, asynchronní automaty a mnoho dalšího.

Jediným větším nedostatkem knihovny je absence časovače a k němu připojení události, která by se do automatu poslala s vypršením časového limitu. Toto však není žádný velký problém. Velmi jednoduše to jde vyřešit například vytvořením speciálního vlákna pro tento časovač. Po spuštění se vlákno okamžitě uspí na požadovanou dobu a po opětovném probuzení vlákna je do automatu poslána událost o vypršení požadovaného časového intervalu.

6.2 LLVM

LLVM (Low level virtual machine) poskytuje infrastrukturu pro překladače. LLVM knihovny jsou psány v jazyce C++. S pomocí knihoven LLVM, jak již samotný název napovídá, lze velmi jednoduše vytvořit virtuální stroje. LLVM poskytuje infrastrukturu pro mnoho dalších projektů, mezi nimiž je i projekt Clang, který poskytuje alternativní překladač k překla-

dačům skupiny GCC, ačkoli projekt LLVM byl původně zamýšlen jako nadstavba právě pro GCC.

Projekt knihoven LLVM odstartoval v roce 2000 na University of Illinois at Urbana Champaign. Původně byl projekt zamýšlen pro výzkum možností dynamické kompilace statických[4] a dynamických[5] programovacích jazyky. V roce 2005 se jádro projektu přestěhovalo do společnosti Apple a LLVM se stalo nedílnou součástí vývojářských nástrojů pro Mac OS.

LLVM používá překlad programu do byte kódu a poté tento byte kód přeloží do kódu instrukcí pro jednotlivé procesory. Program v byte kódu lze poté jednoduše interpretovat na různých architekturách. S pomocí LLVM lze tedy kód jednoduše optimalizovat a poskytnout programátorům celý backend (část překladače závislá na architektuře počítače) pro vývoj nových překladačů.

V současné době se pod hlavičkou LLVM vytvořilo několik různých projektů. Za zmínění určitě stojí projekt LLDB (Low level debugger), který poskytuje vysoce výkonný debugger pro různé programovací jazyky. Dále existují projekty dragonegg a llvm-gcc 4.2, které využívají standardní kompilátor GCC, ale nahrazují v něm optimalizátor a generátor kódu za standardní, který je v LLVM. V neposlední řadě sem patří již krátce zmíněný Clang, kterému je věnována celá další část.

6.3 Clang

Clang je frontend (část překladače závislá na programovacím jazyku) kompilátor pro skupinu programovacích jazyků C (C, C++, Objective-C a Objective-C++). Cílem vytvoření tohoto kompilátoru je nabídnout alternativu ke standardnímu kompilátoru GCC, která bude poskytovat lepší přístup k jednotlivým funkcím frontend části a bude poskytovat podrobnější informace uživateli pro diagnostiku chyb ve zdrojovém kódu.

Projekt Clang byl zahájen až v roce 2007. Z počátku podporoval pouze jazyk C, ale nyní umožňuje téměř kompletní práci se všemi výše zmíněnými programovacími jazyky a vytváření vlastních překladačů pro téměř jakkoliv definovaný programovací jazyk.

Celý překladač je členěn do několika knihoven, z nichž každá poskytuje pouze určitou část funkcí pro vytvoření překladače. Nejvíce používanými knihovny pro analýzu kódu v rámci této bakalářské práce budou knihovny *AST* (6.3.1), Driver (6.3.2) a Frontend (6.3.3).

6.3.1 Knihovna *AST*

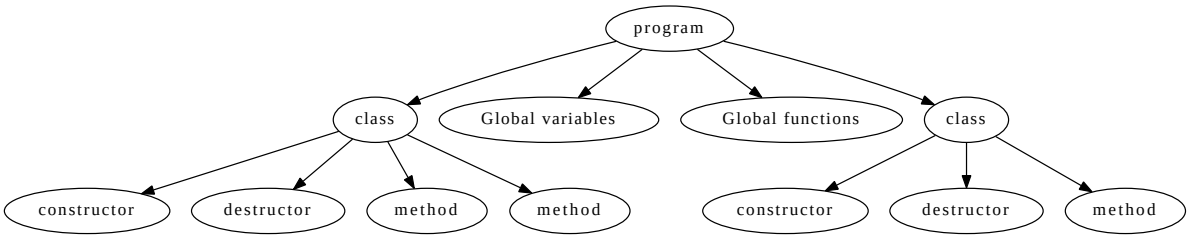
V našem programu je nejvíce použita knihovna *AST*, která poskytuje programátorovi funkce pro úpravu a zevrubnou analýzu AST. Nejdůležitější je možnost vytvoření vlastní třídy,

[4] **Statický(staticky typovaný):** datové typy jsou známy již v průběhu kompilace
[5] **Dynamický(dynamicky typovaný):** datové typy se určují až za běhu

která je potomkem ASTConsumer poskytující rozhraní pro práci s AST.

Celkový AST se dá z pohledu knihovny rozdělit na 2 části.

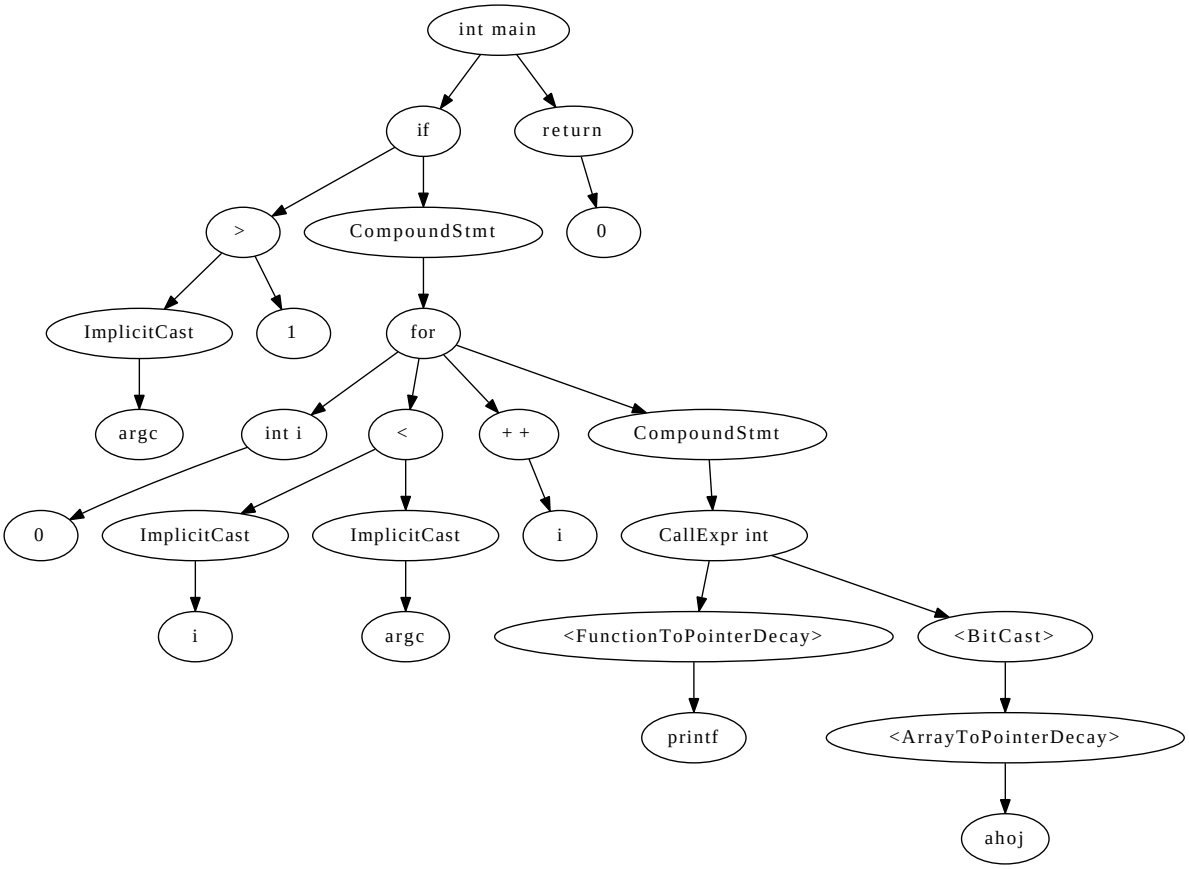
1. **Globální část** – kořenem tohoto stromu je soubor. Jednotlivé uzly jsou tvořeny třídami, jejich metodami a globálními deklaracemi. Tato část je reprezentována třídami, jejichž předkem je třída Decl. Zde se využívá kontextu jednotlivých deklarací pro hledání potomků deklarací. Ukázku této části lze nalézt na obrázku 6.1.



Obrázek 6.1: AST z pohledu jednoho souboru

2. **Jednotlivé funkce** – v kořenu stromu leží funkce. Uzly tvoří jednotlivé výrazy v kódu rozebrané ze syntaktického pohledu. V této části AST se používají třídy, které mají za hlavní třídu Stmt. Pro zjednodušení práce je v každé třídě implementována metoda children, která vrací všechny potomky daného výrazu (uzlu v grafu). Obrázek 6.2 reprezentuje tuto část AST přesně tak, jak je reprezentována v rámci programu.

```
int main(int argc, char **argv)
{
    if(argc>1)
    {
        for(int i = 0; i<argc; i++)
        {
            printf("ahoj\n");
        }
    }
    return 0;
}
```



Obrázek 6.2: Zdrojový kód a jemu příslušející AST vygenerovaný s pomocí Clang

6.3.2 Knihovna Driver

Tato knihovna pomáhá programátorovi se zpracováním příkazové řádky a řízením procesu kompilace. Dá se říci, že obecně reprezentuje právě předzpracování zdrojového kódu před kompilací.

Celý proces její funkce začíná zpracováním argumentů příkazové řádky. Poté, co je celá příkazová řádka zpracována, je vytvořen seznam dalších úkonů. Právě s pomocí zpracovaných argumentů z příkazové řádky lze vytvořit instance pro kompilace. V našem případě instanci třídy CompilerInvocation. Z ní lze poté dostat veškeré informace pro další zpracování. Tato třída již není součástí knihovny Driver, ale knihovny Frontend.

6.3.3 Knihovna Frontend

Knihovna Frontend, již dle svého názvu, reprezentuje část závislou na programovacím jazyku. Hlavním úkolem třídy CompilerInvocation je uchovávat informace o kompilaci pro jejich pozdější použití. V rámci konstruktoru jsou aplikovány zpracované argumenty a vytvořena veškerá nastavení potřebná pro kompilaci. Samozřejmě, že v knihovně Frontend není pouze třída CompilerInvocation, ale například i třída CompilerInstance, která je schopna kromě vlastností držet i další objekty, které jsou potřebné pro kompilaci. V našem programu je použita pouze třída CompilerInvocation, protože většinu těchto objektů potřebujeme pouze jednou a v programu reprezentujeme také pouze jednu část kompilátoru a to syntaktickou analýzu. Použití takto komplexního objektu by tedy bylo zbytečné.

7 Popis aplikace

V této části je zdokumentován celý program, který je výsledkem této bakalářské práce. Celý program je podrobně zdokumentován přesně tak, jak jsou jeho části postupně využívány v průběhu vizualizace stavových automatů.

Nejprve jsou uvedeny přípravy, které jsou nutné před tím, než program dostane k dispozici vytvořený AST (7.2). Do této části například patří zpracování argumentů příkazové řádky a vytvoření klienta pro diagnostiku. Dále je zmíněna část, která se zabývá hledáním stavů, událostí a samotného stavového automatu (7.3). Následuje krátká kapitola o hledání přechodů a obecně reakcí na události (7.4). Podrobněji je popsán postup hledání vlastních reakcí (7.5), mezi kterými se mohou také nacházet přechody. Závěrem je zmíněn postup pro výpis do souboru (7.6), jednoduché statistiky o automatu (7.7) a ukázka toho, jak vůbec takový výpis programu na standardní výstup vypadá (7.8).

7.1 Vývoj programu

Celý vývoj probíhal pod OS Linux a byl průběžně testován nejen pod 32 bitovou, ale i pod 64 bitovou verzí. Podmínkou pro správné fungování je nainstalované LLVM a Clang ve verzi 2.9. Rychlost vývoje aplikace byla zpočátku ovlivněna hlavně nedostatečnou dokumentací k tomu, jak programy vytvořené s pomocí Clangu kompilovat. Pro první seznámení s těmito nástroji byla tedy zvolena úprava ukázkových příkladů přiložených k distribuci Clangu. Zejména jednoho pluginu pro Clang, který pracoval s AST. Jakmile se podařilo tyto problémy překonat, celý vývoj se urychlil a schopnosti aplikace začaly vzrůstat.

Na počátku celého vývoje nebyly pevně stanoveny všechny atributy stavových automatů, které má být aplikace schopna vizualizovat. Seznam s požadavky dle jejich priority lze nalézt v kapitole 4.

7.2 Příprava na analýzu zdrojového kódu

Tato část je v kódu reprezentována funkcí main. Nachází se v ní všechny potřebné implementace před tím, než je možné provést samotnou analýzu kódu.

Patří sem tedy oblast diagnostiky (7.2.1), zpracování vstupních parametrů (7.2.2) a z nich získání zejména informací o vstupním souboru, výstupním souboru a umístění hlavičkových

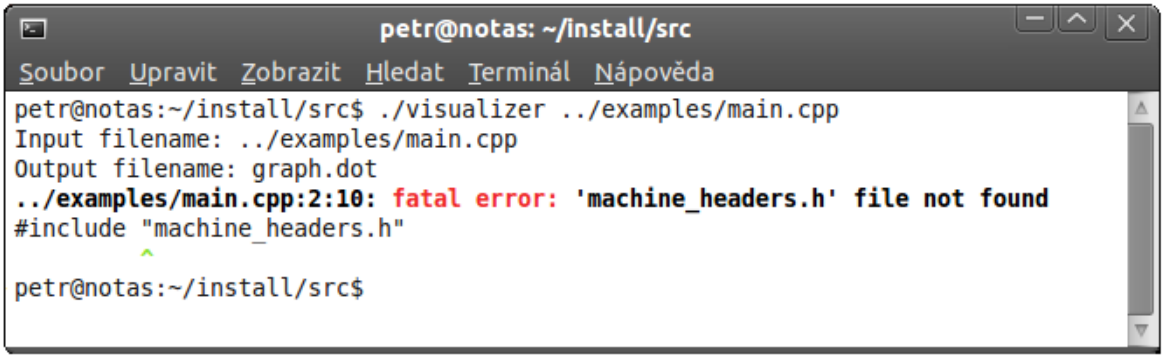
souborů. Dále sem patří příprava a nastavení vlastností před parsováním souboru a vytvořením AST (7.2.3) a jako poslední krátký úvod do parsování souboru s pomocí *AST* (7.2.4).

7.2.1 Diagnostika

Nejdříve je třeba vytvořit diagnostického klienta, který provádí výpis diagnostických informací a v případě chyby ukončí celý program. Využíváno je rozhraní TextDiagnosticPrinter, které umožňuje vlastní implementaci reakcí na specifické druhy chyb, varování a poznámek. Výpis zajišťuje původní knihovní implementace, která se zavolá při každém volání vlastní implementace metody HandleDiagnostic, ale pouze v případě chyby je program ukončen. Varování a poznámky nevadí při zpracování souboru. Tento klient běží ve vlastním vláknu, takže k výpisu diagnostických informací dochází průběžně při zpracovávání zdrojových souborů a to až při procházení a vyhledávání v rámci AST.

Každé takovéto volání se projeví v diagnostických statistikách, které se zobrazí na konci práce programu, tedy i v případě, že je program předčasně ukončen kvůli chybě.

Výstup diagnostiky je zobrazen na standardní chybový výstup přímo v klasickém stylu, jaký používá Clang při překladu. Aby byl výpis diagnostiky lépe odlišitelný pro uživatele od standardního výstupu programu, tak je barevně označen, čímž je docíleno jednoznačného rozdílu mezi standardním výstupem programu a výstupem diagnostického klienta. Ukázku jak se takový barevný chybový výstup zobrazí uživateli je vidět na obrázku 7.1.



Obrázek 7.1: Ukázka barevného výpis diagnostického klienta

7.2.2 Zpracování vstupních parametrů

Pro zpracování vstupních parametrů je využito třídy Driver, která provádí nejen zpracování vstupních parametrů, ale umožňuje řídit celý překlad programu. Driver zpracuje celou příkazovou řádku a všechny parametry umístí do příslušných proměnných reprezentujících atributy pro další třídy. Nejdůležitějším parametrem pro hledání stavových automatů je -I, což reprezentuje umístění hlavičkových souborů. Ručně je třeba nastavit umístění standardních

hlavičkových souborů jazyku C, protože Clang potřebuje používat své vlastní kopie, které jsou vytvořeny při překladu a instalaci Clangu. To je snadné, protože celá adresa jejich umístění se dá vytvořit s pomocí konstant ze systémových souborů LLVM.

Jediným parametrem, který nelze takto získat je parametr -o (s pomocí tohoto parametru je možné definovat název výstupního souboru), protože by bylo nutné posunout celé zpracování do poslední fáze při překladu programu, což je vytváření výstupního spustitelného souboru. Jelikož by toto vyžadovalo komplikovanější řešení, byla zvolena jednoduchá metoda na prohledání vstupních parametrů přímo z argumentů příkazové řádky. Jakmile je tento parametr nalezen, tak je celé procházení zastaveno.

7.2.3 Ostatní nastavení před počátkem parsování

Nejprve je třeba získat veškeré vlastnosti a nastavení. K tomu je potřeba vytvořit třídu CompilerInvocation právě z argumentů příkazové řádky zpracovaných třídou Driver. CompilerInvocation je určen právě jako nositel nastavení a vlastností při samotné kompilaci. Poté s pomocí funkcí typu getOpts() získáme nastavení pro preprocesor, vstupní jazyk, frontend a další. Vzhledem k tomu, že Driver rozpoznává jazyk pouze z hlavního vstupního souboru a vůbec se nestará o hlavičkové soubory, proto je třeba nastavit možnosti jazyka na plnou verzi objektového C++. Po získání těchto nastavení dojde k výpisu informací o vstupním a výstupním souboru.

Mezi další nutná nastavení a přípravu před samotným parsování patří jednoznačně inicializace preprocesoru, protože je nutné ještě před začátkem procházení AST zpracovat příkazy pro preprocesor. Sem patří například #include, #define a #if. K preprocesoru je třeba ještě nastavit používání builtin funkcí, aby nedocházelo k chybám při používání programu.

Dále je potřeba nastavit vstupní soubor ve třídě SourceManager, která slouží k udržení informací a samotných souborů při cachování do paměti během kompilace. Každý soubor je unikátně identifikován pomocí FileID. S touto třídou lze tedy provádět identifikace umístění jednotlivých deklarací v rámci souboru a celého programu. Třída ASTContext spolu s vlastním rozhraním pro hledání, třídou FindStates, která je potomkem ASTConsumer, jsou poslední dvě třídy, které je před parsováním nutné inicializovat.

Posledním krokem je zapnutí diagnostického klienta, které informuje o tom, že začalo zpracování hlavního souboru. Od této doby je tedy spuštěna diagnóza a její výpis. Klient je po ukončení parsování vypnut, protože dále již není potřeba.

7.2.4 Parsování souboru

Tato část je reprezentována funkcí ParseAST, kde jsou nejdůležitějšími parametry instance třídy ASTContext, Preprocesor a FindStates, která má za nadtřídu ASTConsumer. AST

je zparsován a metodě HandleTopLevelDecl je předána skupina globálních deklarací. Jedná se o globální funkce, proměnné, třídy, struktury, jmenné prostory, v nichž se hledají stavy, události. Nyní je vše připraveno pro vyhledávání v souboru.

7.3 Hledání stavových automatů

Stavy, události i stavový automat mohou být z hlediska datového typu struktury nebo třídy. Jediný rozdíl je v tom, že v případě použití třídy je nutné změnit přístupová práva na public kvůli přechodům mezi stavy.

Pouhým prohledáváním globálních deklarací by mohlo dojít k nenalezení některých stavových automatů a to včetně jejich stavů a událostí. Pokud by byl automat umístěn uvnitř namespace, nebyl by nalezen vůbec. Taková deklarace by totiž nebyla tímto způsobem dosažitelná, takže je v případě namespace provedeno prohledání jeho vnitřních deklarací. Zanořování do hloubky tímto způsobem není omezeno. Lze tedy nalézt opravdu všechny stavové automaty.

Jelikož jsou všechny stavy, události, ale i samotný stavový automat reprezentovány z hlediska datového typu strukturou nebo třídou, tak je pro jejich vyhledávání použit stejný postup. Hlavním rozdílem je to, že každá skupina těchto deklarací má jinou nadtřídu. Pro každou deklaraci samozřejmě existují určitá specifika, která celé prohledávání zjednodušují.

Pro případ stavového automatu platí, že program předpokládá existenci pouze jednoho stavového automatu. Takové hledání se provádí pouze do prvního úspěchu.

Stavy mohou být deklarovány až po stavovém automatu, takže se s jejich hledáním začíná až tehdy, když je stavový automat úspěšně nalezen.

Události se mohou nacházet kdekoli, takže se hledají po celou dobu prohledávání, tady bohužel neexistuje žádné omezení, které by snížilo počet testování.

7.4 Hledání přechodů

Tato část je zaměřena na hledání typedefs uvnitř tříd. Tímto způsobem jsou definovány veškeré reakce automatu. Aby mohl program provést toto hledání, je třeba nejprve získat vnitřní kontext třídy nebo struktury, v tomto případě dostat kontext o úroveň níže v hierarchii AST (metody, proměnné, ...).

Dále je celý kontext procházen s pomocí knihovních iterátorů. Postupně jsou veškeré deklarace testovány na svůj typ a hledá se právě typedef. Pro testování je použito číslování typů deklarací v rámci Clangu. Jméno deklarace je vždy transition. V této deklaraci se může nacházet více přechodů, v takovém případě jsou přechody ještě umístěny v template třídě mpl::list. Proto je nejprve proveden test na počet členů.

V případě transition se jedná přímo o přechod mezi stavy, takže jsou zde dva parametry a to název události a cílový stav. V tomto případě program pouze získá parametry, které jsou

poté v hlavní třídě uloženy do seznamu. Na výstup jsou pak ještě upraveny, aby byly korektně vykresleny.

Custom_reaction si zaslouží speciální pozornost, protože po takové události nemusí nutně následovat přechod. V tomto případě má pouze jediný parametr a to název události. Reakce na tuto událost je poté popsána ve speciální metodě, která je součástí třídy.

7.5 Hledání speciálních reakcí

Do této oblasti patří zpracování custom_reactions a tedy metod, kde jsou jednotlivé události parametry. Tyto metody se nachází uvnitř tříd nebo struktur, takže jsou hledány v rámci hledáni typedef pro přechody, ale v případě jazyka C++ se mohou uvnitř třídy nacházet pouze prototypy a samotné deklarace úplně jinde. Jelikož potřebujeme nutně znát tělo takové metody, je nutné hledat tyto metody i mimo třídy.

Při hledání mimo třídu se používá stejných metod jako pro hledání stavů, událostí nebo stavového automatu. Opět je využita identifikace pomocí číslování typů deklarací. Dokonce je díky tomu možné vynechat funkce a soustředit se pouze na metody ze tříd. Dále je postup úplně stejný jako v případě deklarace uvnitř třídy.

Nejprve se zpracuje třída, ve které se metoda nachází a také její parametr. S pomocí klasického prohledávání do hloubky se projde celé tělo metody, protože návratových hodnot může být více než jedna. Program prohledává pouze celé výrazy, tedy neprochází dopodrobna obsah výrazů. Například i=3+4; není pro něj atraktivní, a tedy není dále rozpracováno na konstanty, operátory, ... Mezi zajímavé konstrukty například patří if, do, while, for, switch. Nejdůležitějším je ovšem návratová hodnota.

Jak jsem již uvedl, mezi návratovými hodnotami mohou být i další reakce kromě přechodů. Asi nejčastěji se zde kromě přechodu vyskytuje zahození události. Při nalezení return dojde k testu na návratovou hodnotu. V případě, že se jedná o template class transit, pak je nalezen přechod. Třída má pouze jeden parametr a to je cílový stav. Tento stav je vrácen a vytvořen standardní zápis do seznamu přechodů.

7.6 Výpis do souboru

Pro tuto část je ve zdrojových kódech vytvořena speciální třída s názvem IO_operations nacházející se v souboru iooper.h. Výpis probíhá do souboru tak, aby byla zachována hierarchická struktura automatu, což znamená správné vykreslení subautomatů.

V prvním kroku se vypíší všechny stavy, které leží v kontextu stavového automatu. Aby bylo možno jednoduše kontrolovat, že vše bylo vypsáno, tak se daný stav smaže ze seznamu stavů. Výjimku tvoří stav se subautomatem, který se sice vypíše, ale nesmaže. Vždy po dokon-

čení průchodu seznamem stavů se změní kontext pro výpis na první stav, který obsahuje subautomaty. V případě tohoto programu je tento stav vždy na vrcholu seznamu, protože by jinak automat nebyl zkompilovatelný a tím by neprošel diagnostikou. Takto se celý proces opakuje, dokud není seznam se stavy prázdný.

Při výpisu přechodů se již nehledí na to, že některé stavy neleží ve stejném kontextu. Dot nakreslí čáry od stavu ke stavu bez ohledu na to, jestli leží v jiných subgrafech. Tím je celý výpis do souboru dokončen. Dále je uvedena ukázka okomentovaného výstupního souboru pro jednoduchý stavový automat.

```
digraph DU { //vytvoř graf s názvem DU
//výpis stavů
state_1;
state_2;
state_3;
state_4;
state_1 [peripheries=2] ; //počáteční stav
//výpis přechodů
state_1->state_4[label="EvA"]; //ze stavu state_1 do stavu state_4 na událost EvA
state_1->state_2[label="EvN"];
state_2->state_3[label="EvTimer"];
state_3->state_1[label="EvN"];
state_3->state_4[label="EvTimer"];
state_4->state_1[label="EvTimer"];
}
```

Až je celý tento výstup dokončen, je vypsána přechodová tabulka, která byla průběžně vytvářena během příprav na výpis do souboru. Tabulka obsahuje samozřejmě všechno, co obsahuje obrázek generovaný z výstupního souboru. Počáteční stavy jsou v ní označeny s pomocí znaku * v prvním sloupci.

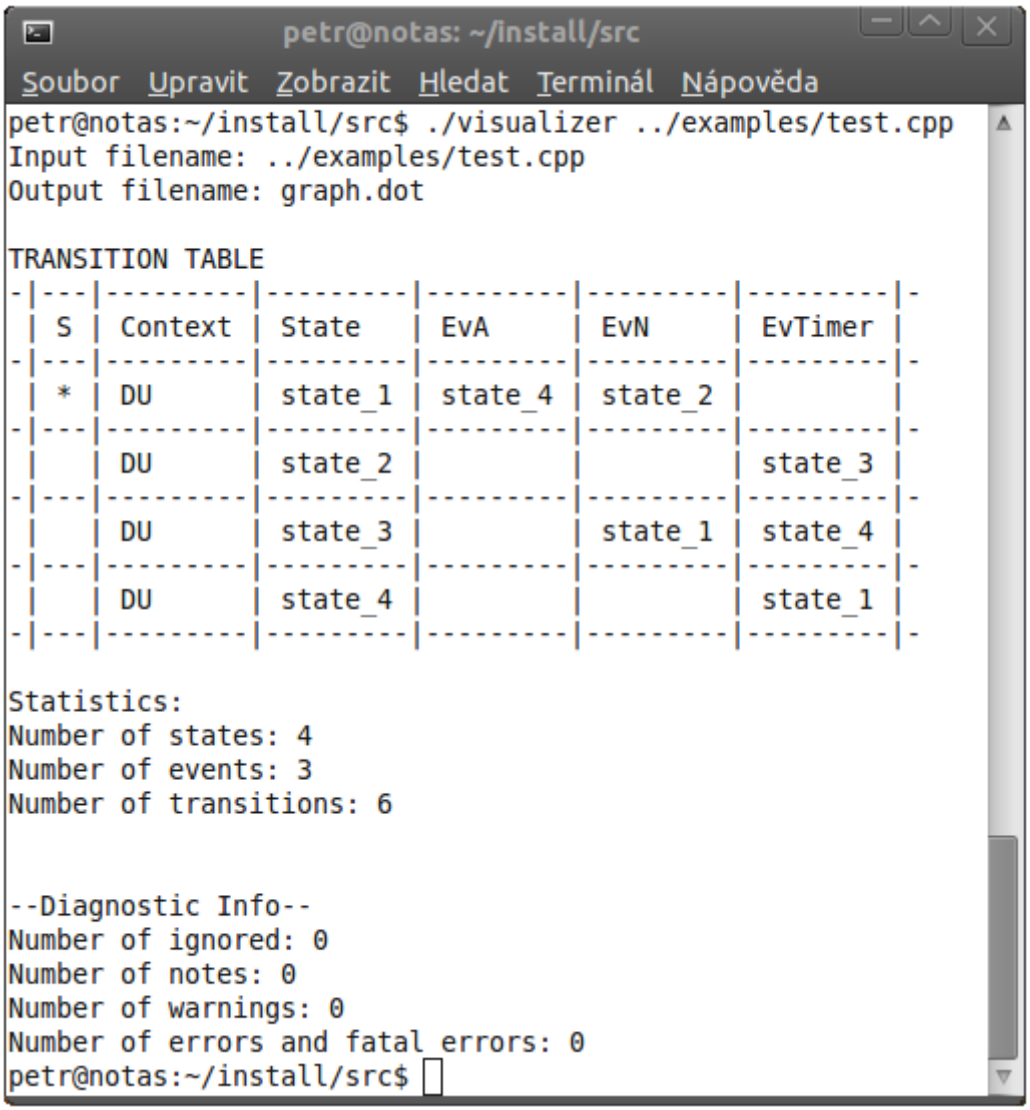
7.7 Statistiky

Na závěr běhu celého programu jsou vypsány krátké statistiky obsahující počet stavů, událostí a přechodů. Z hlediska diagnostiky je vypsán rovněž počet varování během prohledávání zdrojového souboru. Tyto statistiky slouží jako jednoduchá kontrola správnosti kódu pro programátora.

Statistiky ohledně počtu nalezených chyb, varování a poznámek jsou zobrazeny i v případě, že dojde k předčasnému ukončení práce programu. Přesně tak, jak je tomu u obvyklých kompilátorů.

7.8 Výstup programu

Na závěr této části celé práce je na obrázku 7.2 uveden ukázkový výstup z programu při zpracování vstupního souboru. Z přechodové tabulky jsou patrny seznamy stavů, událostí a přechodů. Samozřejmostí je označení počátečního stavu *. V případě, že automat obsahuje i subautomaty, jsou zde rozpoznatelné rovněž jejich počáteční stavy.



Obrázek 7.2: Výstup programu

8 Testování

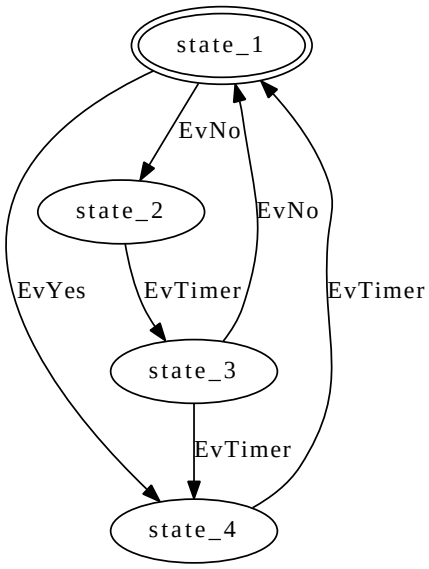
Samozřejmostí pro vývoj takovéto aplikace je testování. K tomuto účelu jsem používal několik vlastnoručně zhotovených automatů a některé příklady, které jsou přiloženy ke zdrojovým kódům a tutorialu ke knihovně Boost-Statechart. Jeden z bodů v zadání bakalářské práce je seznámení komunity Boost s výsledky vývoje, takže jsem se snažil využít i této možnosti pro otestování. Zajímal mě také jejich názor na tento program.

Nejprve jsou tedy uvedeny výsledky mého vlastního testování (8.1) a poté ještě názory vývojářů knihovny Boost-Statechart a vůbec celé komunity, která se zabývá vývojem knihoven Boost (8.2).

8.1 Vlastní testování

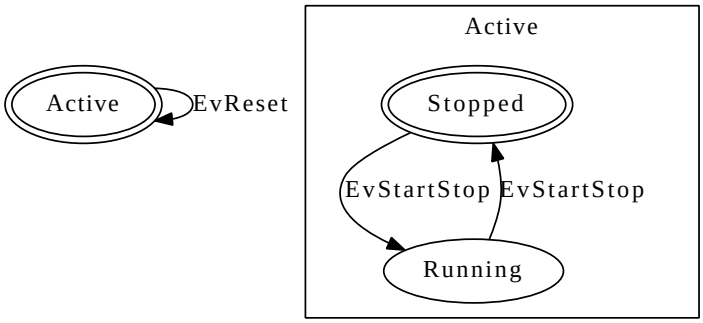
Program byl testován v průběhu celého vývoje za pomoci čtyř automatů, které byly průběžně upravovány, aby byly schopny otestovat všechny možnosti a schopnosti programu. Poté, co je každý testovací soubor popsán, je zobrazena jeho struktura ve formě stavového diagramu. Veškeré zdrojové kódy pro testování lze nalézt na přiloženém CD ve složce examples, kde se, kromě již zmíněných zdrojových kódů, nacházejí také vygenerované soubory ve formátu dot a také obrázky vytvořené z těchto souborů.

První automat neobsahoval žádné subautomaty. Obsahoval pouze 4 stavy a 6 přechodů zapsaných pomocí transitions a tedy neobsahoval ani žádné vlastní reakce na události. S pomocí tohoto automatu byly rovněž ukázány možnosti pro zobrazování struktur stavových automatů uvedené v této práci. Stavový diagram tohoto automatu vygenerovaný programem je na obrázku 8.1.



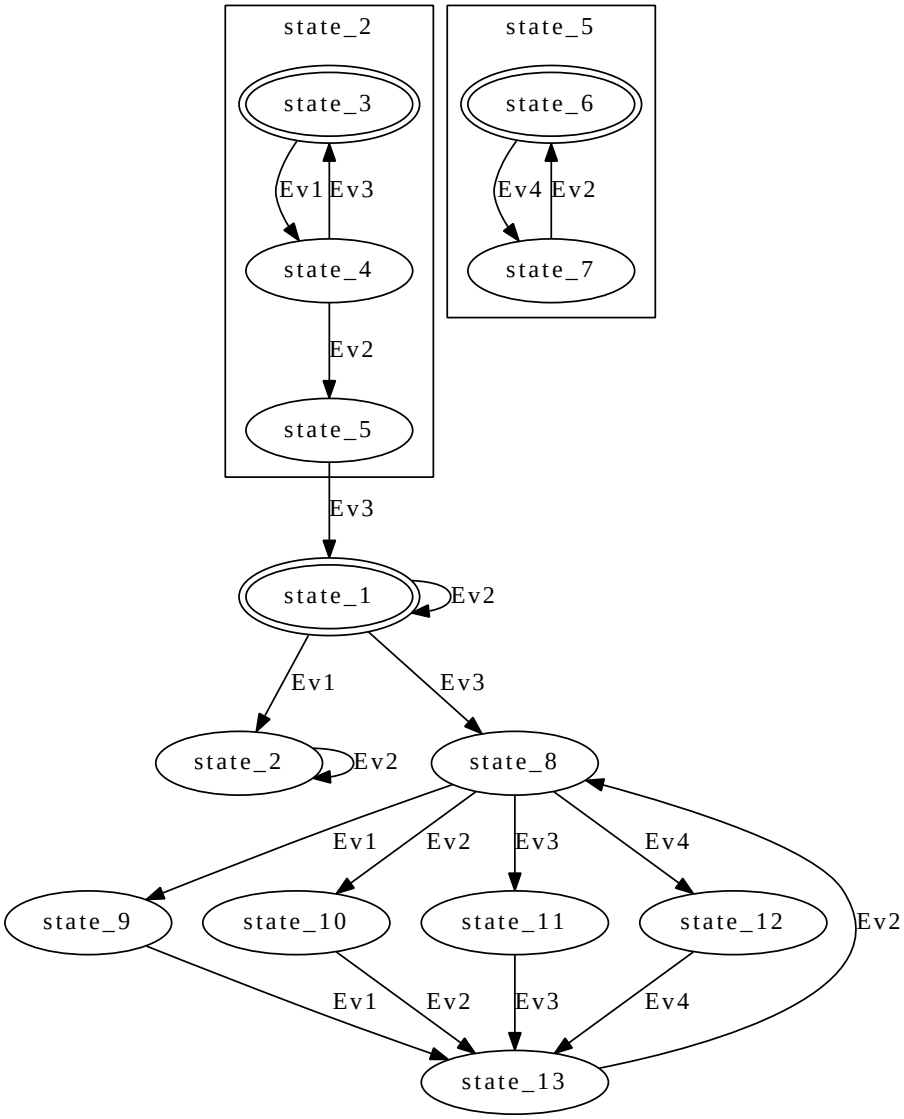
Obrázek 8.1: První stavový automat pro testování

Druhý automat byl již ze skupiny příkladů přiložených ke knihovně Boost-Statechart. Z hlediska struktury simuloval chování stopek a obsahoval nejen samotnou simulaci, ale prováděl i měření času. Obsahoval subautomat a jeho stavy měly více nadtříd, tato vlastnost právě umožňuje provádět měření času. Přechody byly definovány opět pomocí transitions a tedy bez vlastních reakcí. Stavový diagram, který jsem vygeneroval s pomocí programu, je na obrázku 8.2.



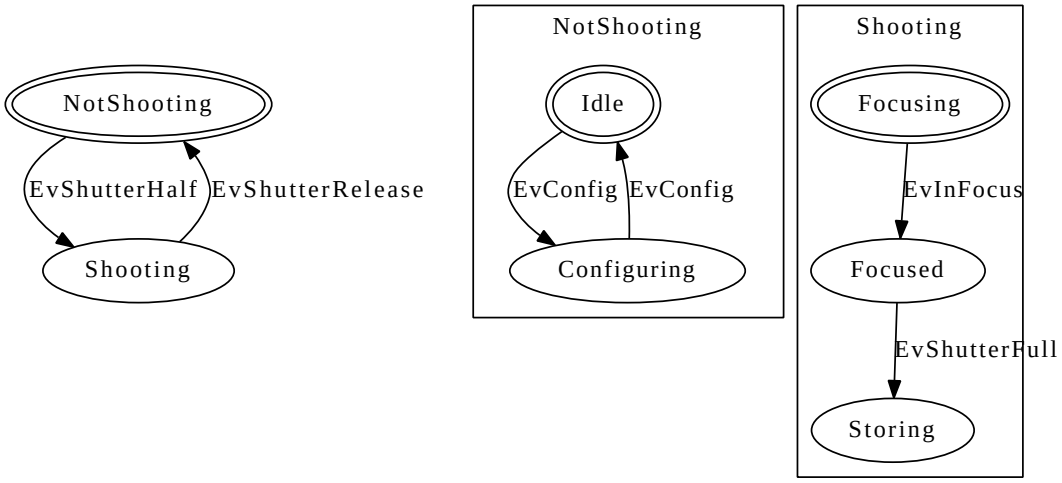
Obrázek 8.2: Druhý stavový automat pro testování

Další automat vytvořený pro testování obsahoval více vnořených subautomatů, aby bylo možno otestovat správné hierarchické vykreslování subautomatů. Obsahoval celkem dvanáct stavů a z hlediska svého rozsahu byl nejrozsáhlejší. Samotný zdrojový kód obsahuje pouze strukturu automatu a tedy žádné ostatní metody. Dalším důvodem pro použití tohoto automatu bylo to, že se celý nenacházel v hlavním souboru, ale v hlavičkovém souboru. Tím bylo ověřeno správné prohledávání i mimo hlavní soubor. Vygenerovaný stavový diagram příslušející tomuto automatu je pak na obrázku 8.3.



Obrázek 8.3: Třetí stavový automat pro testování

Poslední automat pro testování je opět z ukázkových automatů u knihovny Boost-Statechart. Simuluje chování fotoaparátu při fotografování. Obsahuje dva subautomaty, ale jeho hlavní vlastností je použití vlastních reakcí pro přechody. Z hlediska struktury není rozsáhlý, ale umožňuje právě otestovat správné vyhledávání vlastních reakcí. Pro potřeby testování v průběhu vývoje byl několikrát upravován, aby bylo možno testovat různé varianty umístění vlastních reakcí v kódu. Jelikož je tento automat nejsložitější a to hlavně tím, že jsou zde použity i vlastní reakce na události, je zde kromě vygenerovaného obrázku 8.4 i ukázka přechodové tabulky tak, jak se pro tento stavový automat zobrazí uživateli, obrázek 8.5.



Obrázek 8.4: Čtvrtý stavový automat pro testování

S	Context	State	EvShutterHalf	EvShutterFull	EvShutterRelease	EvConfig	EvInFocus
*	Camera	NotShooting	Shooting				
	Camera	Shooting			NotShooting		
*	NotShooting	Idle				Configuring	
	NotShooting	Configuring				Idle	
*	Shooting	Focusing					Focused
	Shooting	Storing					
	Shooting	Focused		Storing			

Obrázek 8.5: Přechodová tabulka vygenerovaná programem

8.2 Názor autorů knihovny Boost-Statechart

Jelikož součástí zadání bakalářské práce je informování komunity kolem knihovny Boost-Statechart, byla vytvořena jednoduchá internetová stránka http://rtime.felk.cvut.cz/statechart-viewer/, která prezentuje to, co tento nástroj umí. Na této stránce lze nalézt i jednoduchý návod na používání a specifikované požadavky na verze LLVM a Clang. Po vytvoření této webové stránky byl odeslán e-mail do mailing listu komunity kolem knihoven Boost.

Reakce na tento e-mail byly pozitivní. Vývojáři dokonce vyjádřili zájem o rozšíření podpory pro další knihovnu, která umožňuje vytvářet stavové automaty. Jedná se o knihovnu Boost-MetaStateMachines, dále v práci zkracovanou jako Boost-MSM. Výsledky prezentované na webových stránkách je zaujaly.

Mezi reagujícími byl i hlavní vývojář Boost-Statechart, Andreas Huber. S ním probíhala podrobnější komunikace, protože chtěl program podrobně otestovat. To se bohužel nepodařilo z důvodu používání jiných hlavičkových souborů v OS Windows a v programu Vi-

sual Studio 2010. Jelikož byl program vyvíjen pro OS Linux, bylo nejprve nutné provést jednoduchou úpravu programu pro OS Windows, která zahrnovala například použití zpětných lomítek. Ve Visual Studiu se rovněž nedá použít Makefile, takže bylo nutno celý vygenerovaný příkaz pro kompilaci odeslat emailem. Kompilace programu se sice zdařila, ale během analýzy docházelo k chybám při použití hlavičkových souborů Visual Studia 2010. Krátký popis nejen tohoto problémů, ale celé komunikace s komunitou Boost, lze nalézt na přiloženém CD ve složce emails.

9 Budoucí vývoj

Nyní bych rád uvedl několik nejbližších kroků v rozvoji tohoto nástroje pro kreslení stavových automatů napsaných s pomocí knihovny Boost-Statechart. Tyto kroky by měly být poté učiněny v rámci rozšíření podpory pro stavové automaty.

Prvním krokem v dalším rozvoji programu je jednoznačně přidání podpory pro ortogonální stavy, tedy pro stavy, kde může paralelně běžet více stavových automatů. To by neměl být v současné struktuře programu žádný problém. Jediná větší úprava by byla nutná při vykreslování automatů do výstupního souboru. Po dokončení této části by již podpora stavů měla být kompletní a program by si měl dokázat poradit se všemi možnými stavy.

Dalším bodem je rozšíření podpory událostí. V současné době program umožňuje s pomocí událostí realizovat pouze přechody, ale využití událostí je mnohem širší. Na vrcholu důležitosti mezi těmito ostatními je určitě zahazování událostí v jednotlivých stavech a předávání událostí mezi stavy, ale i automaty. Tato část se bude zobrazovat v přechodové tabulce, ale některé reakce se promítnou i do generovaného souboru s popisem struktury stavového automatu.

Další možností je rozšíření podpory i pro druhou knihovnu od komunity Boost zabývající se stavovými automaty a to knihovnu Boost-MetaStateMachines. Vývojáři vyjádřili zájem o rozšíření podpory právě pro tuto knihovnu, když se v rámci plnění úkolu seznámit s výsledky komunitu Boost, dozvěděli o existenci tohoto programu. Vhodnější variantou by v tomto případě bylo asi vytvoření nového programu, který by ale využíval poznatky již existujícího řešení pro knihovnu Boost-Statechart.

Dokonalou verzí tohoto programu by byla verze s GUI (Graphical user interface), která by uměla z obrázku vygenerovat zdrojový kód. Dále také umožnit automatické úpravy kódu v závislosti na změně stavového diagramu.

10 Závěr

Cílem této práce bylo navrhnout a implementovat program pro vizualizaci stavových automatů pro řízení robotů. Prvním z předpokladů bylo to, že stavové automaty budou napsány s pomocí knihovny Boost-Statechart.

Nejprve bylo nutné zvolit vhodný nástroj pro analýzu zdrojových kódů. Pro tuto analýzu jsem zvolil LLVM a Clang, protože s pomocí jejich knihovních funkcí se dají analyzovat zdrojové kódy. Jako metoda analýzy byl zvolen AST. Tato volba se ukázala v průběhu vývoje aplikace jako velmi dobrá, protože s pomocí AST lze zdrojový kód velmi jednoduše procházet.

Provedl jsem rešerši existujících řešení pro vizualizaci stavových automatů. A to pro metodu vizualizace ze zdrojového kódu pro různé knihovny. Zmínil jsem rovněž jeden program, který umožňuje generovat zdrojové kódy z vytvořených obrázků.

Naprogramoval jsem program, který jsem otestoval na několika příkladech. Oproti původnímu plánu, kdy se počítalo pouze se zobrazením struktury automatu do grafu, je ve výsledném programu i vygenerovaná přechodová tabulka. Ta je připravena i pro zobrazování další práce s událostmi, která se bohužel zatím nepovedla implementovat. V přechodové tabulce jsou vnitřní stavy v rámci jednotlivých kontextů umístěny pod sebou.

Program jsem otestoval na několika stavových automatech. Mezi nimi nechyběly ani ukázky stavových automatů distribuovaných společně s knihovnou Boost-Statechart. Tyto příklady byly průběžně upravovány, aby se otestoval celý program. Výsledky testování byly úspěšné. Program dokázal vizualizovat všechny atributy stavových automatů, které měl.

Vytvořil jsem jednoduché internetové stránky programu a seznámil jsem s výsledky komunitu Boost. Ohlasy byly pozitivní. Dokonce se objevil požadavek na podporu další knihovny, která umožňuje vytváření stavových automatů a to knihovnu Boost-MSM. Bohužel se nepodařilo program zprovoznit pod OS Windows, takže nedošlo k důkladnému otestování hlavním vývojářem knihovny Boost-Statechart. Kompilace byla pod tímto OS úspěšná, ale při spuštění došlo k chybám při použití hlavičkových souborů Visual Studia 2010.

V závěru jsem se ještě pokusil nastínit další kroky v rozvoji programu.

Použitá literatura

[1] Andreas Huber Dönni, Internetová dokumentace knihovny Boost-Statechart, dostupné online, 20.04.2011
http://www.boost.org/doc/libs/1_46_1/libs/statechart/doc/index.html

[2] LLVM Team, Internetová dokumentace projektu LLVM, dostupné online, 20.03.2011
http://www.llvm.org

[3] LLVM Team, Internetová dokumentace překladače Clang, dostupné online, 20.03.2011
http://clang.llvm.org

[4] I.Černá, M.Křetínský a A.Kučera: Automaty a formální jazyky I, FI MU, 2002, dostupné online, 16.06.2010
http://is.muni.cz/elportal/estud/fi/js06/ib005/Formalni_jazyky_a_automaty_I.pdf

[5] "Architecture des Systèmes intégrés et Micro électronique" department, Dokumentace programu XFSM, dostupné online, 17.04.2011
http://www-asim.lip6.fr/recherche/alliance/doc/design-flow/tools.html

[6] Jiří Bayer, Zdeněk Hanzálek, Richard Šusta: Logické systémy pro řízení, 1. vyd. Praha : ČVUT, 2000, dostupné online, 17.01.2010
http://dce.felk.cvut.cz/lor/prednasky/skripta/

[7] Alexander Darovsky , Dokumentace projektu FSME, dostupné online, 08.05.2011
http://fsme.sourceforge.net/

[8] Informace o požadavcích na vizualizátor Boost-Statechart, dostupné online, 09.09.2010
http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Google_Summer_Of_Code_2006

[9] Dictionary.com, "automaton," in Dictionary.com Unabridged. Source location: Random House, Inc., 06.05.2011
http://dictionary.reference.com/browse/automaton

[10] Filip Jareš, Implementace stavových automatů pro soutěž Eurobot 2009, Bakalářská práce ČVUT 2010
http://support.dce.felk.cvut.cz/mediawiki/images/5/50/Bp_2010_jares_filip.pdf

Příloha A – Obsah CD

/text	elektronická verze této práce ve formátu pdf
/src	zdrojové kódy pro aplikaci včetně Makefile pro jejich kompilaci. Hlavní soubor `visualizer.cpp`, soubor pro úpravu deklarací `stringoper.h` a soubor s třídou pro generování přechodové tabulky a stavového diagramu `iooper.h`.
/examples	ukázkové příklady stavových automatů včetně vygenerovaných obrázků
/emails	obsah emailové korespondence s komunitou Boost
/docs	dokumentace zdrojových souborů v pdf