Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA

Ján Baláž

Interpretr stavových automatů popsaných jazykem SCXML.

Katedra softwarového inžinierstva

Vedúci bakalárskej práce: RNDr. Ondřej Šerý
študijný program: Programovanie

2009

Na tomto mieste by som chcel poďakovať svojmu vedúcemu práce RNDr. Ondřejovi Šerému za konzultácie a pomoc pri vytváraní práce. Rovnako by som chcel poďakovať aj svojmu starému vedúcemu práce RNDr. Jiřímu Semeckému Ph.D. za ponúknutie zaujímavej témy.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a s jej zverejňovaním.

V Prahe dňa 11. 12. 2009 Ján Baláž

Obsah

Názov práce: Interpretr stavových automatů popsaných jazykem SCXML.
Autor: Ján Baláž
Katedra: Katedra softwarového inžinierstva
Vedúci bakalárskej práce: RNDr. Ondřej Šerý
e-mail vedúceho: ondrej.sery@dsrg.mff.cuni.cz

Abstrakt: V súčasnosti vzniká potreba pristupovať ku rôznym programátorským úlohám alternatívnymi postupmi. Jedným z takýchto postupov je zápis pomocou stavových automatov. Ako riešenie sa ukazuje novo vznikajúci štandard W3 Consorcia, nazvaný SCXML. V súčasnosti už existuje niekoľko rôznych implementácii, ktoré však kvôli svojej rozsiahlosti nemusia byť vhodné pre jednotlivcov, alebo malé a stredné firmy. Tieto spoločnosti potrebujú nástroje, ktoré si môžu jednoducho prispôsobovať na svoje vlastné projekty. Cieľom práce je vytvoriť interpreter stavových automatov popísaných jazykom SCXML, ktorý bude spĺňať popísané podmienky. Snahou je implementovať maximálnu časť špecifikácie SCXML. Práca bude vytvorená v jazyku Perl a bude podporovať operačné systémy Windows a Linux.

Klúčové slová: SCXML, stavový automat, interpreter

Title: SCXML state machine interpretter.
Author: Ján Baláž
Department: Department of Software Engineering
Supervisor: RNDr. Ondřej Šerý
Supervisor's e-mail address: ondrej.sery@dsrg.mff.cuni.cz

Abstract: Currently there is a need for programmers to access various tasks with alternative procedures. One such practice is using the state machine entry. As the solution shows a newly emerging standard by W3 Consorcium, called SCXML. By now, there are several different implementation, but because of their robustness may not be suitable for individuals or small to medium businesses. These companies need tools that can easily adapt to their own projects. The aim of this work is to create a state machine interpreter described SCXML language that will satisfy the conditions described. The effort is to implement the maximal part of SCXML specification. Work will be created in Perl and will support the operating systems Windows and Linux.

Keywords: SCXML, state machine, interpreter

Kapitola 1

Úvod

Vďaka modernej technike sa niekedy dôležité faktory ako napríklad rýchlosť aplikácii dostávaju do úzadia a do popredia vystupujú faktory ako je udržovatelnosť, prehľadnosť a rozšíritelnosť softvéru. Vznikajú nové technológie, ktoré umožňujú popisovanie zložitých problémov pomocou jednoduchých konštrukcií a zároveň s veľkou sadou prostriedkov, vďaka ktorým sa programátor nemusí zaoberať kedysi komplikovanými vecami, akými sú rôzne alokácie pamäte, sieťové spojenia atď. V ústrety tomuto trendu sa vyvinula rada nástrojov, ktoré umožnujú pomerne jednoducho a prehľadne zachytiť požiadavky zákazníkov vo forme prijatelnej pre designérov a programátorov. Vyvinula sa tiež rada formátov vhodných pre zápis dát a ich prenos medzi rôznymi platformami, napríklad XML[1]. Ako pomerne jednoduchý zápis niektorých problémov sa ukázal rozšírený stavový automat. Využíva sa na popis aplikácii pre telefónne centrá, taktiež je vhodný pre jednoduché popisovanie rôznych tokov softvéru[6]. V súčasnosti využívaný zápis rozšírených stavových automatov, napríklad v UML 2.0[2], vytvoril matematik David Harel a nazval ho Harel State Tables [1]. Na základe týchto nových technológii začalo v roku 2005 W3C vyvíjať štandard SCXML, alebo *State Chart eXtensible Markup Language*, ktorý je spojením rozšíreného stavového automatu Davida Harela a prenosného formátu XML. V súčasnosti už existuje niekoľko implementácii na rôznych úrovniach splnenia špecifikácie, napríklad Apache Commons SCXML[3] v Jave, Qt SCXML Engine[4] v C++, Intervoice[5]. Keďže SCXML špecifikácia je zatiaľ vo verzii Working Draft a pravidelne sa mení, na dôkladné splnenie špecifikácie treba implementáciu pravidelne aktualizovať a upravovať. U komerčných projektov sa musia užívatelia spoliehať na spoločnosť, ktorá ich vyvíja a niektoré opensource projekty sú pomerne komplikované, teda tiež sa treba spoľahnúť na ich vývojový tým.

[1] eXtensible Markup Language, http://www.w3.org/XML
[2] Unified Modeling Language, http://www.uml.org
[3] http://commons.apache.org/scxml
[4] http://labs.trolltech.com/page/Projects/xml/scxml
[5] http://www.convergys.com/products/contact-center-software/intervoice-voice-portal.php#a2

1.1 Cieľ práce

Cieľom bakalárskej práce je navrhnúť a naprogramovať interpreter SCXML, ktorý splní rozumnú časť špecifikácie SCXML. Tiež bude jednoducho upravovateľný na prispôsobenie rôznym podmienkam konkrétnych spoločností, prípadne aj zmenám v špecifikácii, a zároveň dostatočne modulárny, aby si ho mohli užívatelia nastavovať aj bez zásahu do programového kódu. Pri spracovaní bude kladený dôraz hlavne na navrhnutie vhodných datových štruktúr, ktoré umožnia pomerne jednoduchú rozšíritelnosť softvéru a využívanie prostriedkov, ktoré dovoľujú multiplatformnosť aplikácie v rámci OS Windows a OS Linux.

K vytvoreniu práce ma motivovalo využitie SCXML pre vývojárov v stredných a menších firmách, ktoré si mnohé veci zvyknú upravovať samostatne tak, aby vyhovovali do ich vlastných komerčných riešení, a teda potrebujú jednoducho nastavovateľný, rozšíriteľný a modulárny framework.

1.2 Štruktúra práce

Detailnejší popis jednotlivých častí bakalárskej práce:

Kapitola 1. Úvod Vysvetlenie dôvodov vzniku SCXML štandardu. Špecifikácia cieľa bakalárskej práce. Príčina motivácie.

Kapitola 2. Analýza komponent Rozsiahla analýza problematiky. Vysvetlenie aké komponenty som sa rozhodol použiť. Popis SCXML z môjho pohľadu spojeného zo špecifikáciou.

Kapitola 3. Analýza návrhu Analýza konkrétnych problémov, ktoré vznikli pri návrhu aplikácie. Popis rôznych riešení a zdôvodnenia, prečo som sa pre konkrétne z nich rozhodol.

Kapitola 4. Podrobný návrh Popis vybraných štruktúr z implementácie a niektorých zaujímavostí.

Kapitola 5. Rozdiely oproti špecifikácii SCXML Popis rozdielov, ktoré vznikli oproti špecifikácii SCXML. Uvádzam tu vysvetlenia, prečo som niektoré časti neimplementoval. Tiež uvádzam časti, ktoré softvér poskytuje naviac oproti štandardnej špecifikácii a dôvody, prečo som sa tak rozhodol.

Kapitola 6. Záver Porovnanie očakávaní a výsledkov práce. Zhrnul som tu môj osobný pohľad na prácu a plány na pokračovanie do budúcnosti, po odovzdaní bakalárskej práce.

Kapitola 2

Analýza komponent

2.1 Použité komponenty

2.1.1 Programovací jazyk – Perl

Pri výbere programovacieho jazyka som bral na zreteľ:

1. Sada knižníc, ktorými disponuje.
2. Vhodnosť štruktúr jazyka na implementáciu projektu.
3. Umožnenie multiplatformnosti v rámci OS Windows a OS Linux.
4. Jednoduchosť prípadných úprav konečnými užívateľmi, chcel som hlavne skriptovacie jazyky.
5. Svoje vlastné programátorské možnosti.

Tieto požiadavky som analyzoval nasledujúcim spôsobom:

1. Ako programovací jazyk som si zvolil Perl. Splňuje podmienku veľkej sady knižníc, ktoré sa dajú sťahovať z archívu cpan.
2. Štruktúry má jednoduché a veľmi dobre využitelné. Zvlášť práca s asociatívnym a klasickým poľom je veľmi pekne vyriešená. Bol navrhnutý pre spracovávanie textu[2], teda spracovanie XML zápisu nie je problém. Perl podporuje OOP, čo je vhodné pri návrhu. V softvéri využívam aj vlákna a komunikáciu cez sockety, čo Perl bez problémov zvláda.
3. Perl funguje ako samostatná platforma, teda programy v ňom napísané bežia aj pod OS Linux aj pod OS Windows. Pod OS Linux je Perl štandardnou súčasťou distribúcii. Pod OS Windows existuje distribúcia ActivePerl[1] zdarma stiahnuteľná

[1] http://www.activestate.com/activeperl/

zo stránky spoločnosti ActiveState, ktorá ju vyvíja, prípadne komunitná distribúcia Strawberry Perl[1].

4. Keďže je Perl skriptovací jazyk, úpravy sa dajú robiť veľmi jednoducho a nie je potrebná žiadna kompilácia. Zároveň to zabezpečuje, že softvér ostane opensource.

5. S Perlom som sa predtým stretol, ale programátorské skúsenosti som s ním nemal takmer žiadne. Tento projekt mi poskytol možnosť sa s Perlom viacej zblížiť. Vzhľadom na jeho pomerne jednoduchý základ sa mi celkom rýchlo podarilo do Perlu preniknúť. Ďalším dôvodom bolo doporučenie od môjho prvého vedúceho bakalárskej práce.

Ako alternatíva zo skriptovacích jazykov prichádzal do úvahy Python. Z prekladaných zase C++, ale v ňom by sa vývoj podstatne predĺžil a nespĺňal by podmienku jednoduchej upravovateľnosti.

Keďže vybraný jazyk mal byť skriptovací, rozhodol som sa nasledovať doporučenie svojho vedúceho a využiť Perl.

2.1.2 Festival

Festival je pomerne rozšírený opensource nástroj TTS, *Text To Speech*. Využíva sa vo viacerých aplikáciach pod OS Linux, teda je dostatočne otestovaný a spoľahlivý a tiež funguje pod OS Windows, teda spĺňa podmienku multiplatformnosti. Zároveň preň existujú Perlové moduly, čo považujem za obrovskú výhodu, keďže sa to nemusí odznova programovať.

Ako alternatíva prichádza do úvahy OpenTTS, ktoré je menej známe a rozšírené pod rôznymi distribúciami OS Linux.

Rozhodol som sa pre rozšírenejšiu variantu, teda Festival, pretože táto vlastnosť je veľmi žiadanou z užívateľského hľadiska.

2.1.3 CMU Sphinx

CMU Sphinx je framework pre budovanie aplikácii využívajúcich SRE, *Speech Recognition Engine* . Existuje vo štyroch verziách:

- CMU Sphinx 2 – najrýchlejší a zároveň najdlhšie vyvíjaný, ale už sa ukončila jeho podpora, napísaný v C++.
- PocketSphinx – odľahčená verzia CMU Sphinx 2, využíva sa v niektorých aplikáciach pod OS Linux, konkrétne na ovládanie prostredia GNOME.

[1] http://strawberryperl.com/

- CMU Sphinx 3 – pomalší ako CMU Sphinx 2, má viacej možností, napísaný v C++.
- CMU Sphinx 4 – najnovšia a najpodporovanejšia verzia. Napísaná v jazyku Java, teda zaručuje multiplatformnosť.

Rozhodol som sa pre CMU Sphinx 4, pretože je najnovší a bude mať dlhodobú podporu do budúcnosti, zároveň je napísaný v jazyku Java, ktorý zaručuje multiplatformnosť.

2.2 SCXML

SCXML je zaujímavé pre svoj odlišný prístup k popisovaní problémov. Umožňuje jednoducho a prirodzene vytvárať stavové automaty s niektorými rozšírenými funkciami. Základom SCXML je dokument napísaný v XML formáte. Táto sekcia je mojim pohľadom na SCXML a časť materiálu na jej tvorbu som prevzal z literatúry [1].

2.2.1 Element scxml

Na začiatku dokumentu sa definuje SCXML element, ktorý určuje ako sa bude parsovať a interpretovať nasledujúca SCXML aplikácia. Najdôležitejšie sú parametre `profile` a `exmode`.

Parameter profile. Určuje, ktorý prednastavený profil SCXML sa použije. Podľa špecifikácie musí každá SCXML implementácia povoliť `minimum` profil a môže podporovať `ecmascript` profil, alebo `xpath` profil. Profil určuje aké moduly budú pri interpretovaní použité a podľa jednotlivých modulov sa zároveň určuje sada povolených elementov a ich význam pri interpretácii. V mojej implementácii, nie je podporovaná kompletná sada elementov a parametrov, ktorú prikazuje špecifikácia, vysvetlenia k jednotlivým nepodporovaným súčastiam sa nachádza v Kapitole 4. Rozdiely oproti SCXML špecifikácii. Konkrétne do parametru profil je možné zadávať iba hodnotu `perl` , ktorá umožní interpretovanie elementov z *core module*, *data module* a *script module*, ktorý využíva Perl ako skriptovací jazyk.

Parameter exmode. Určuje správanie sa interpretu v situácii, keď narazí na element, ktorý v danom profile nie je podporovaný. Povolené hodnoty sú podľa špecifikácie `lax` a `strict`. Pričom v móde `lax` interpret ticho ignoruje nepovolené elementy a v móde `strict` je interpret povinný vyhlásiť internú chybu `error.unsupportedelement`. Pri hodnote `lax` sa interpret správa korektne, avšak pri hodnote `strict` je xml dokument validovaný pred prvým behom aplikácie narozdiel od špecifikácie, podľa ktorej validácia prebieha až počas interpretácie aplikácie.

2.2.2 Druhy výrazov v SCXML a systémové premenné

SCXML špecifikuje druhy výrazov, ktoré je možné používať v rôznych miestach. Medzi tie zaujímavé patria názvy premenných, v špecifikácii *Location expression*, a hodnotové výrazy, v špecifikácii *Value expression*.

Názvy premenných. *Location expression* v SCXML vyjadrujú umiestnenie premennej v dátovom modele, ktorý je aktuálnym profilom využívaný. Využívajú sa pri priradeniach premenných, pomocou elementu assign, či pri špecifikácii premenných pre posielanie, napríklad v elemente send. Zaujímavá je tu flexibilita, ktorú SCXML poskytuje pri definovaní týchto výrazov. Totiž ich tvar vôbec definovaný nie je. O túto záležitosť sa má postarať konkrétny profil, ktorý si programátor môže nastaviť v elemente `scxml`. Definované sú tieto výrazy len pre doporučené profily `ecma` a `xpath`. Keďže vo svojej implementácii ani jeden z týchto profilov nepodporujem a zavádzam nový profil `perl`, sú *location expressions* definované podľa mňa.

Na začiatku som mal ako povolené hodnoty perlové výrazy, čo celkom fungovalo. Pri priradeniach stačilo Perlu vnútiť interpretovanie vzniknutého výrazu a on sa už o priradenie postaral sám. Napríklad v premennej `$name` bol uložený perlový výraz ako cesta k premennej a v premennej `$value` bol výraz, ktorý som chcel priradiť.

```
$name = '$_data{'osoba'}->{'krstne'}';
$value = 'Jano';
eval($name . '=' . $value);
```

Tento výraz zabezpečil priradenie hodnoty do správnej premennej. Príkaz `eval` interpretoval reťazec ako perlový kód všetko sám zabezpečil. Problém však nastal pri vymenovaní premenných pre posielanie. Programátor má podľa špecifikácie napríklad v elemente `send` možnosť cez parameter `namelist` vymenovať zoznam premenných, ktoré sa majú odoslať. Teda pri perlových výrazoch ako *location expression* mohol programátor písať:

```
<send ... namelist="$_data->{'osoba'}->{'krstne'}" />
```

Nevymenoval som všetky parametre, iba `namelist`, pretože ostatné nie sú momentálne dôležité. V takomto prípade mala správa doraziť, tak aby jej dáta mali rovnakú štruktúru ako sa posielala. Teda programátor, ktorý chcel prečitať danú premennú písal perlový výraz:

```
$_event->{'data'}->{'osoba'}->{'krstne'}
```

V tejto chvíli nastal problém, keďže výraz v parametri `namelist` som nevedel previesť do tvaru štruktúry. Pre prikaz `eval` výraz predstavoval hodnotu, ale ako sa ku nej dostal a cez ktoré referencie prechádzal, mi bolo neznáme. Nevedel som z reťazca perlového výrazu jednoducho rekonštruovať štruktúru, ktorú predstavoval.

Preto som nakoniec zvolil jednoduchší zápis *location expression*, ktorý by mi umožňoval reprezentovať všetky potrebné štruktúry a zároveň sa dal rozumne parsovať. Ako štruktúry prichádzali do úvahy reťazce, obyčajné polia a asociatívne polia. Reťazce môžu reprezentovať čísla a znaky, polia sú nutné, a asociatívne polia reprezentujú sami seba a štruktúry. Ako zápis som zvolil jednoduché názvy premenných oddelené pomocou znaku /, ktorý sa v názve premennej nesmie vyskytovať a jeho výskyt je v programovacích jazykoch na tomto mieste značne neobvyklý. Vyššie uvedený príklad by sa dal prepísať ako:

```
<send ... namelist="osoba/krstne" />
```

V prípade, že štruktúra je komplikovanejšia napríklad:

```
# Definícia štruktúry:
$_data = { 'osoba' =>
{ 'krstne' => 'Jano', priezvisko => 'Balaz' },
bydliska => [ 'Praha', 'Presov'], vek => 23 };

# Odošle celú štruktúru meno.
<send ... namelist="osoba" />
# Prístup sa ku premennej:
$_event->{'data'}->{'osoba'}

# Odošle premennú krstne.
<send ... namelist="osoba/krstne" />
# Prístup ku premennej:
$_event->{'data'}->{'osoba'}->{'krstne'}

# Odošle prvok poľa bydliska.
<send ... namelist="osoba/bydliska/[0]" />
# Prístup ku premennej:
$_event->{'data'}->{'osoba'}->{'bydliska'}->[0]

# Odošle premennú krstne a prvý prvok poľa bydliska.
<send ... namelist="osoba/bydliska/[0] osoba/krstne" />
# Prístup ku nim podľa predpokladu:
$_event->{'data'}->{'osoba'}->{'krstne'}
$_event->{'data'}->{'osoba'}->{'bydliska'}->[0]
```

Takýto spôsob umožnuje jednoducho naparsovať meno premennej a zostaviť odpovedajúcu štruktúru, tým pádom odosielať len vymenované premenné.

Hodnotové výrazy. *Value expressions* v SCXML vyjadrujú povolené hodnoty. Pri *value expressions* je rovnako zaujímavá voľnosť ako pri predchádzajúcich *location expressions*. Existujú tu dve časti tohto zápisu. *Data expression* , teda správny zápis dát, a *Value expression*, teda správny zápis výrazu, pričom výsledok interpretácie každého *value expression* musí byť správny *data expression*, inak je interpret povinný zahlásiť udalosť `error.illegalvalue`. Použitý dátový modul definuje povolené *data expression* a na základe toho je potom profil povinný definovať povolené *value expression*. Ako je možné vidieť štandart necháva aj tu vývojárom intrepretu aj programátorom v SCXML veľkú voľnosť. Keďže implementujem profil perl, ako *data expression* som zvolil všetky perlové premenné a ako *value expression* som zvolil zase všetky perlové výrazy. Teda väčšinu práce tu môžem nechať odvádzať Perl namiesto mňa, stačí pre každú interpretáciu zavolať perlový príkaz `eval`, ktorý interpretuje akýkoľvek perlový skript, pričom vráti hodnotu posledného riadku, alebo uloží chybový výpis. *Value expression* a *data expression* sa zapisujú do parametrov `cond` a `expr` pri rôznych scxml elementoch a sú v dôsledku jednoriadkové, preto príkaz `eval` ideálne vyhovuje stanoveným podmienkam.

Systémové premenné. SCXML špecifikácia definuje takzvané systémové premenné, ktoré programátorovi sprístupňujú niektoré funkcie. Systémové premenné sa vyznačujú tým, že ich názov začína znakom `_`. V súčasnej špecifikácii sa vyskytujú tri takéto premenné.

- `_name` označuje názov SCXML session, teda aktuálneho spustenia a hodnota sa jej priradzuje cez parameter name v elemente scxml. Jej hodnota je prístupná vo všetkých skriptoch a *value expressions* ako `$_data->{'_name'}`.
- `_data` označuje dátový model a pomocou nej môže programátor v podmienkach a skriptoch pristupovať k premenným uloženým v dátovom modele. V mojej implementácii je táto premenná hash mapou a je prístupná vo všetkých skriptoch a *value expressions* ako `$_data`.
- `_event` označuje štruktúru spätú s poslednou spracovávanou udalosťou. Táto premenná je nedefinovaná, pokiaľ nepríde prvá udalosť. Túto premennú som implementoval ako hash mapu.V podmienkach a v prechodoch je prístupná ako`$_event` a v ostatných skriptoch ako`$_data->{'_event'}`.

2.2.3 Stavy

V SCXML sa vyskytujú tri druhy elementov určujúcich stav v automate. Sú to stavy typu `state`, `parallel` a `final`. Spoločnými znakmy pre tieto stavy je možnosť do nich vchádzať a opúštať ich pomocou prechodov, v špecifikácii nazývaných `transition`, prípadne definovať spustitelný obsah, ktorý bude vykonaný pri vojdení do stavu, v špecifikácii

`onentry`, prípadne pri opustení stavu, v špecifikácii `onexit`. Každý element typu stav, musí mať parameter `id`, pomocou ktorého sa stav v rámci aplikácie identifikuje a programátor sa naňho pomocou `id` odkazuje, s hodnotou unikátnou pre celú SCXML aplikáciu.

Element state. Klasický stav známy zo stavových automatov je stav typu `state`. V SCXML tento stav môže zastupovať stav jednoduchý, v špecifikácii nazývaný `atomic`, v prípade, že nemá žiadne podstavy. Tiež môže zastupovať stav zložený, v špecifikácii sa volá `compound`, v prípade, že má nejaké ďalšie podstavy. Podstavom sa rozumie v XML strome priamy potomok typu stav.

Element parallel. Pomocou stavov typu `parallel` sa v SCXML podporuje paralelizmus. Tento stav môže mať potomkov typu `parallel`, alebo `state` a nesmie mať potomka typu `final`. Interpret zareaguje tak, že počas jedného kroku vojde do všetkých potomkov. Ak sú potomkovia `parallel` alebo zložené `state`, vojde rekurzívne aj do ich potomkov. Teda v jednom kroku interpret vloží do novej konfigurácie naraz všetkých svojich priamych potomkov typu `state`, alebo `parallel`, čím im umožní reagovať na prichádzajúce udalosti. Teda vzniká efekt paralelizmu, keď automat môže mať v rovnakej hĺbke spracovania viacej aktívnych stavov.

Element final. Stav typu `final` označuje výstupný/prijímací stav automatu. Ak interpret narazí na takýto stav, vygeneruje udalosť `done.state.parentid`, pričom `parentid` nahradi `id` svojho rodičovského stavu.

2.2.4 Udalosti

Interpret SCXML zavádza typ udalosť, v špecifikácii sa nazýva `event`. Každá udalosť musí mať názov a môže mať data, ktoré povoľujú akúkoľvek štruktúru. Slúži na prechody medzi jednotlivými stavmi, prípadne presúvanie dát. V mojej implementácii sa rovnaké udalosti ako spracováva SCXML interpreter, používajú aj na komunikáciu, medzi modulmi v projekte. V SCXML existujú dva druhy udalostí a tými sú `event` a `error`. `Event` je obyčajná udalosť signalizujúca nejaký stav, alebo dej, na ktorý môže automat reagovať. `Error` je špeciálny podtyp udalosti, na ktorú interpret reaguje rovnakým spôsobom ako na `event`, ale narozdiel od `event` slúži na signalizáciu chyby interpreteru, ktorú má automat možnosť odchytiť a spracovať. Medzi takéto chyby patria napríklad nesprávne výrazy pre hodnoty premenných, prípadne pre názvy premenných, alebo elementy, ktoré automat nepozná, či nevyhodnotiteľné podmienky. Podľa špecifikácie automat nie je povinný spracovať všetky udalosti, teda nemusí ani odchytávať chyby. Väčšina týchto chybových hlásení je v aktuálnej implementácii zahrnutá. Nie sú využívané len chyby, ktoré vyplývajú z neimplementovaných častí SCXML.

Zápis udalostí. Udalosti, ako už je vyššie spomenuté, sú dvoch typov a to buď `event`, alebo `error`. Zapisujú sa vo formáte `error.typchyby.ďalšie špecifiká`. Teda jednotlivé položky názvu su od seba oddelené pomocou bodky. Rovnaké pravidlá platia aj pre udalosti typu `event`.

Porovnávanie mena udalosti. Aby došlo ku reakcii na udalosť, musia sa zhodovať mená udalostí. Nech u1 je názov prichádzajúcej udalosti a u2 je názov udalosti, na ktorú reaguje prechod medzi stavmi. Potom názov udalostí sa zhoduje, ak u2 je prefixom u1, alebo u2 je rovnaké ako u1, alebo sa zhodujú za použitia wildcardu `*`, ktorý je povolený v u2. Wildcardy sú povolené a implementované len v `event` parametri u prechodov, ak sa použijú v iných situáciach pri zadávaní názvu udalosti, berú sa ako obyčajný znak a nezastúpia svoju špeciálnu funkciu.

2.2.5 Prechody

Prechody sú v SCXML reprezentované elementom `transition`. Každý prechod patrí práve jednému stavu, ktorý nazývam rodičovským stavom, v špecifikácii `parent state`. Prechody umožnujú presúvať sa medzi stavmi ako v obyčajných stavových automatoch. Tiež môžu mať spustiteľný obsah, v špecifikácii `executable content`, ktorý sa spustí po opustení rodičovského stavu a pred vojdením do cieľových stavov. Majú tri dôležité parametre: `event`, `cond` a `target`. Parametre `event` a `cond` sa vyhodnocujú a v prípade, že sa ich obe podarí kladne vyhodnotiť považuje sa prechod za aktivovaný a spustí sa proces opustienia rodičovského stavu, vykonania spustiteľného obsahu a vojdenia do stavov definovaných parametrom `target`. Celý proces sa deje iba v prípade, že rodičovský stav daného prechodu patrí do aktuálnej konfigurácie automatu.

Parameter event. Označuje názov udalosti, na ktorú prechod reaguje. Povolená hodnota je jedine reťazec zložený z textových znakov. Hodnota reťazca sa porovnáva s názvom prichádzajúcej udalosti. Zhoda týchto hodnôt je nutnou podmienkou pre aktivovanie prechodu. Ak parameter `event` nie je špecifikovaný, prechod sa stáva bezudalostným, čo v špecifikácii sa nazýva `eventless transition`. Pre tento prechod platí, že sa aktivuje vždy, ak sú splnené ostatné podmienky.

Parameter cond. Parameter `cond` sa využíva na špecifikovanie podmienky, ktorá musí byť splnená, aby sa prechod mohol aktivovať. Podmienka musí byť v tvare hodnotového výrazu, v špecifikácii nazývaného *value expression*. Tento parameter nie je povinný a teda, ak nie je uvedený považuje sa vždy za splnený.

2.2.6 Spustiteľný obsah

Spustiteľný obsah je špeciálna oblasť SCXML, ktorá sa vyskytuje v elementoch `onentry`, `onexit` a `transition`. V špecifikácii sa nazýva `executable content` a má za úlohu pracovať s klasickými programovacími nástrojmi známymi zo štrukturálneho programovania a zároveň využíva nástroje potrebné v stavových automatoch. Prvky spustiteľného obsahu sú elementami SCXML. Spracujú sa sekvenčne v poradí v akom sú zapísané v SCXML dokumente. V prípade, že sa počas spracovanie niektorého z prvkov spustiteľného obsahu vyskytne chyba, interpret vytvorí príslušnú udalosť typu `error` a uloží ju do zoznamu interných udalostí a ukončí spracovávanie aktuálneho bloku spustiteľného obsahu, teda ďalšie prvky sa už nespracujú.

- Elementy `if`, `elseif`, `else` sa využívajú rovnako ako v štruktúrovanom programovaní.
- Element `assign` slúži ako priradenie *value expression* premennej určenej pomocou *location expression*.
- Element `send` slúži na generovanie udalostí a ich odosielanie do ostatných modulov, alebo do vlastnej SCXML session, kde sa môžu umiestniť buď do fronty na externé, alebo interné udalost, podľa parametrov.
- Element `raise` zastupuje funkciu elementu `send`, keď má nastavené parametre na ukladanie do fronty na interné udalosti interpretu SCXML.
- Element `log` sa používa na vypisovanie logovacích výpisov, ktoré je platformne závislé. V tejto implementácii som mu dal ešte jednu jednoduchú vymoženosť, a to keď sa neuvedie parameter `level`, výpis je poslaný na štandardný výstup aby bolo možné vytvoriť napríklad aplikáciu *Hello world*.

Element script. Ako najzaujímavejší z elementov v spustiteľnom obsahu hodnotím element `script`. Prináša do SCXML aplikácii možnosť využívať skriptovací jazyk definovaný skriptovacím modulom, v špecifikácii uvedený ako *script module*. Vďaka nemu je v SCXML umožnená v podstate akákoľvek funkčnosť programovacieho jazyka, ktorého skriptovací modul použijeme. Tento element nemá povolených žiadnych potomkov, iba textový obsah, ktorý je zároveň programom napísaným v podporovanom skriptovacom jazyku. Vďaka nemu je teoreticky možné vynechať všetky ostatné elementy spustiteľného obsahu, ak skriptovací modul implementuje ich funkčnosť vlastnými metódami. V mojej implementácii využívam vyššie zmienený perlový skriptovací modul. Rozhodol som sa pre Perl z podobných dôvodov ako pri hodnotových výrazoch. Pomocou perlového príkazu `eval` je možné interpretovať reťazec ako perlový kód a v prípade chyby odchytiť o akú chybu išlo.

V prípade implicitne dostupného perlového skriptovacieho modulu, v podstate programátor môže využívať všetky vlastnosti *executable content* elementov priamo zo svojho perlového skriptu. Záleží teda na voľbe programátora, či využije SCXML prostriedky, alebo bude radšej programovať v Perl-i.

2.2.7 Algoritmus interpretácie

Algoritmus bol vytvorený w3c konzorciom k presnému popisu postupu, ako sa bude správať každý interpret SCXML pri spracovávaní SCXML aplikácii. Popisuje jednotlivé kroky interpretácie aplikácie napísanej v SCXML jazyku, vďaka čomu je možné vytvárať programy jednoducho prenositeľné medzi rôznymi SCXML platformami. Bohužiaľ SCXML je momentálne vo verzii working draft, s čím prichádza možnosť zmien v budúcnosti a zároveň je pravdivé, že nepopisuje dostatočne jasne všetky situácie, ktoré môžu nastať. Ako prechádza medzi jednotlivými verziami working draft, sú niektoré jeho časti nedostatočne okomentované, prípadne neaktuálne. Je nutné pomerne veľké množstvo vecí vymýšľať z vlastnej hlavy tak, aby to bolo pre programátorov, ktorí budú používať tento interpreter, pohodlné a praktické. Nebolo to jednoduché a nedovolím si tvrdiť, či som vystihol smer, ktorým sa bude tento dynamický štandard v budúcnosti uberať. Predpokladám, že v tejto časti interpretu bude ešte pomerne veľa zmien, kým SCXML dosiahne stabilnú verziu 1.0. Popis návrhu som umiestnil do Kapitoly 3. Analýza návrhu.

Kapitola 3

Analýza návrhu

3.1 Požiadavky na aplikáciu

Na začiatku projektu sa bolo treba rozhodnúť, akým spôsobom sa bude uberať ďalší vývoj. K tomu sa museli zhodnotiť všetky dostupné požiadavky na túto aplikáciu.

Aplikácia musí spĺňať:

- Dostatočne modulárna, aby bolo jednoduché pridávať nové možnosti, ktoré by sa dali využívať prostredníctvom SCXML aplikácie. To nie sú konkrétne SCXML elementy, ale moduly voľne pripojiteľné do projektu, ktoré rozširujú jeho celkovú funkčnosť. Napríklad syntéza reči a vstupy z klávesnice.
- Jednotlivé časti aplikácie sú na sebe nezávislé do tej miery, že môžu vykonávať svoju prácu oddelene a teda sa bude používať multithreading.
- Aplikácia musí bežať na platformách OS Windows a OS Linux a teda sa nesmú použiť žiadne prostriedky závislé na jednej platforme, kde by Perl nevedel zabezpečiť použitie aj na druhej platforme. Táto podmienka platí len pre implementáciu interpreteru SCXML, iné moduly samozrejme môžu byť platformne závislé, aby programátori neboli ochudobnení o špeciálne možnosti konkrétnej platformy.

3.2 Rozbor problémov

Rozbor konkrétnych problémov s ohľadom na vyššie zmienené požiadavky.

Problém modelu modulov. Hneď na začiatku sa rozhodlo, že moduly musia byť pripájateľné a odpájateľné nezávisle na zvyšných moduloch a to bez zásahov do zdrojového kódu projektu, čím sa myslí kód bakalárskej práce. Zároveň každý modul musí vykonávať svoju činnosť tak aby nebrzdil ostatné moduly. Zvažovali sa dve možnosti.

1. Každý modul bude vykonávať svoju činnosť vo vlastnom vlákne v rámci jednej aplikácie, čo zabezpečí podmienky na využívanie systémových zdrojov pre všetky moduly rovnaké. Keďže podpora práce s vláknami je priamo na úrovni jazyku Perl, uvedený spôsob je teda platformne nezávislý a prijateľný ako riešenie. Vlákna sú pravdepodobne jedinou možnosťou ako zabezpečiť rovnocenný beh viacerých procesov súčasne v rámci jednej aplikácie.

2. Ďalším riešením by mohlo byť rozdeliť aplikáciu na viacej častí a nechať každý modul vystupovať ako samostatnú aplikáciu. V tomto prípade by neboli potrebné vlákna. Avšak zbytočné rozdeľovanie jedného procesu do viacerých, zabezpečenie ich spustenia a činnosti v sebe nesie rizika závislosti na konkrétnej platforme. Okrem tohto riešenie vychádza ako celkom komplikované na implementáciu oproti prvej možnosti, keďže podpora multiplatformných vlákien je zabudovaná priamo v jazyku Perl.

Po uvedených zdôvodneniach som sa rozhodol pre postup podľa prvej možnosti.

Problém pripájania modulov. Ostal problém s pripájaním modulov do aplikácie bez zmeny zdrojového kódu projektu. Počítalo sa s tým, že každá SCXML aplikácia bude potrebovať nejakú konkrétnu sadu modulov zabezpečujúcu požadovanú funkčnosť. Pritom programátori takejto aplikácie vedia, ktoré moduly bude potrebovať na spustenie. Tu vyvstáva ďalšia dôležitá podmienka, a to že jednotlivé aplikácie sa musia dať konfigurovať aby sa zabezpečila dostatočná pružnosť bez zásahov do zdrojových kódov.

Jedna možnosť by bola nechať pripojené všetky dostupné moduly a pre každý projekt vytvoriť konfiguráciu, kde by boli nastavenia jednotlivých modulov pre aktuálne potreby SCXML aplikácie. Táto možnosť by však v závislosti na implementácii konkrétnych modulov mohla využívať množstvo systémových prostriedkov, ktoré by značne prevyšovalo potreby bežiacej aplikácie. Bolo by teda vhodné, buď presne špecifikovať, ako sa daný modul musí implementovať, aby nezaberal zbytočné prostriedky, alebo obmedziť množstvo pripojených modulov.

Ako riešenie sa ukázalo postačujúce do konfigurácie ku každej aplikácii pridať zoznam modulov, ktoré budú pripojené pri spustení aplikácie. Projekt ide s modulárnosťou tak ďaleko, že samotný SCXML interpreter je modulom, ktorý sa musí pripojiť, aby bolo možné spúšťať SCXML aplikácie. Týmto sú problémy sprístupnenia modulov a ich súbežnej nezávislej práce vyriešené.

Problém inicializácie a komunikácie modulov Vzniká otázka ako budú moduly inicializované a ako bude prebiehať ich komunikácia. Sú zase dve možnosti.

1. Každý modul bude mať kontakt na všetky ostatné moduly a teda takto spolu budú môcť komunikovať.
2. Bude centrálna autorita, ku ktorej budú všetky moduly pripojené a ona zabezpečí komunikáciu medzi modulami. Každý modul bude komunikovať iba s centrálnou autoritou a oznámi jej názov modulu, ktorému má dáta posielať, o čo sa už postará sama centrálna autorita.

Druhé riešenie vychádza značne prehľadnejšie, pretože informácie o moduloch sa sústreďujú na jednom mieste a nie je potrebné ich distribuovať medzi všetky moduly. Ak napríklad nejaký modul neodpovedá, existuje autorita, ktorá má možnosť danú situáciu riešiť, zároveň sa táto autorita môže starať aj o inicializáciu modulov. Pri distribuovanom riešení by takáto autorita neexistovala, pretože spadnutie modulu by si mohlo uvedomiť niekoľko iných súčasne a bolo by potrebné rozhodnúť, ktorý sa o situáciu postará. To by zároveň znamenalo, že moduly musia implementovať funkčnosť, ktorá s ich poslaním priamo nesúvisí.

Problém komunikácie modulov a centrálnej autority. Ostáva definovať, akým spôsobom budú komunikovať moduly s centrálnou autoritou. Prvé riešenie, ktoré padlo vyzeralo nasledovne:

Centrálna autorita sa nazýva `MessagePool`. Komunikácia prebieha presne pomocou správ s presne definovaným formátom. Moduly musia implementovať dohodnuté rozhranie, ktorým je metóda s parametrom typu správa a vracia hodnotu typu zoznam správ. Táto metóda má predpísané, že musí trvať krátku dobu, aby nebrzdila beh ostatných modulov. Ak modul potrebuje dlhší čas na spracovanie správy, musí si sám vytvoriť vlákno, kde spracovanie pobeží. `MessagePool` si uchováva pre každý modul zoznam správ, ktoré mu prišli a ešte ich nespracoval. Potom pravidelne obieha všetky moduly, u každého volá rozhranie, ktoré ako parameter dostane najstaršiu správu čakajúcu vo fronte pre daný modul. Metóda po skončení má zanechať parameter vymazaný v prípade, že správu modul spracoval a teda `MessagePool` ju môže vymazať zo svojho zoznamu pre daný modul. Inak ju ponecháva na vrchole a v ďalšej iterácii ju predá znovu ako parameter. Vo výsledku metóda vracia zoznam správ, ktoré chce modul odoslať. `MessagePool` spracuje tento zoznam a roztriedi ho medzi zoznamy správ čakajúcich na spracovanie pre jednotlivé moduly.

Na nedostatky tohto riešenia poukázal vedúci práce.

- Pri iteráciach cez všetky moduly, je potrebné sa vždy na nejakú krátku dobu zastaviť, inak proces zaberie sto percent času cpu. Čakanie spôsobuje nežiadané spomalenie aplikácie.

- Moduly by si nemali vlákna vyrábať samé, mal by sa o to postarať pri inicializácii už MessagePool.

Oba problémy som riešil súčasne. Keďže obe platformy OS Linux aj OS Windows podporujú sockety a tie sú zároveň súčasťou perlových modulov dostupných v distribúciach pre obe platformy, rozhodol som sa ich využiť. Sockety podporujú blokovací režim, v ktorom čakajú na prichádzajúce dáta, čím sú schopné uspať vlákno po dobu, kým ho nezobudia prichádzajúce dáta. Myšlienka bola nasledovná. `MessagePool` bude autoritou, ktorá vytvorí server, potom vytvorí všetky potrebné moduly vždy v novom vlákne. Tie sa na tento server pripoja cez vlastných klientov. Architektúra klient-server zabezpečí dohľad `MessagePool`-u nad modulami a vďaka blokujúcej implementácii socketov na oboch podporovaných platformách je automaticky zabezpečené, že `MessagePool` pri čakaní na správy od modulov je uspaný a zároveň moduly pri čakaní na správy od ostatných modulov sú uspané. Ak chce programátor modulu iné správanie, musí si ho do modulu naprogramovať, štandardne moduly počas nečinnosti spia. Bolo potrebné navrhnúť vhodného abstraktného predka pre modul, ktorý by implementoval metódy na prácu so socketmi, aby nebola práca programátorov modulov zbytočne komplikovaná. Po odvodení potomka od abstraktného predka všeobecného modulu, má programátor k dispozícii jednoduché rozhranie na pristupovanie k udalostiach obsiahnutých v prichádzajúcich správach a zároveň rozhranie, ktoré automaticky zabalí udalosť do správy a odošle k `MessagePool`-u na spracovanie.

Riziko deadlocku. Existuje tu riziko deadlocku, keď všetci čakajú na správu.

1. Túto situáciu si musia riešiť programátori jednotlivých modulov sami.
2. Ďalšou možnosťou by bolo vytvoriť jedno kontrolné vlákno, ktoré by vždy na určitý čas zaspalo. Po prebudení by zaslalo budiacu správu `MessagePool`-u, ktorá by ho zobudila a ak by bol už dlhší čas nečinný, pokúsil by sa prebudiť pomocou budiacich správ všetky moduly.

Ostalo sa pri prvej možnosti, pretože druhá zjavne poukazuje na chybu v logike naprogramovanej aplikácie. Okrem toho, nie je jasné, ako často by sa prebúdzalo kontrolné vlákno a tiež aký dlhý čas by sa považoval za priveľkú dobu nečinnosti u `MessagePool`-u, aby nedochádzalo k zbytočnému preťažovaniu komunikácie pri náročných operáciach vykonávaných modulami. Takýto modul by sa totiž po skončení operácie vrátil k spracovávaniu nových správ a čakal by naň zoznam plný budiacich správ.

Problém korektného ukončenia .Vzniká tiež problém korektného ukončenia všetkých inicializovaných modulov. Ak `MessagePool` dostane žiadosť o ukončenie celej aplikácie od nejakého modulu, rozošle všetkým modulom správy signalizujúce, aby sa korektne ukončili, pripadne uvolnili alokované prostriedky a pred skončením vlákna zaslali správu o konci

činnosti späť do `MessagePool`-u. V prípade, že sa nejaký modul zasekne, aplikácia bude na jeho správu o ukončení čakať donekonečna, kým ju nevypne užívateľ. Problém v súčasnej implementácii nie je nijako riešený. Ako riešenie by však prichádzalo v úvahu, aby `MessagePool` čakal určitý čas na ukončenie všetkých modulov a keď to moduly nestihnú, ukončil sám aplikáciu násilne. Ak by modul potreboval na ukončenie viacej času, musel by `MessagePool`-u posielať pravidelné ping-y, správy so žiadosťou o predĺženie ukončovacieho času, po ktorých by `MessagePool` ostávajúci čas navyšoval. Bohužiaľ aj v tomto prípade hrozí neustále posielanie ping-ov a tým zamrznutie aplikácie. Všetky tieto riešenia spôsobujú problém, buď príliš krátkeho čakania, alebo príliš dlhého čakania, alebo zamrznutia aplikácie. Najvhodnejším vyzerá byť kompromis, keď má modul k dispozícii určité množstvo ping-ov . Keď ich vyčerpá ďalšie sa už neberú do úvahy a `MessagePool` aplikáciu po uplynutí času ukončí. Teda pokiaľ sa modul zasekne a neposiela ping-y, `MessagePool` aplikáciu ukončí pomerne rýchlo. Pokiaľ posiela ping-y a je zaseknutý ukončí ju po dlhšom čase. Ak modul potrebuje len trochu viacej času, ping mu ho zabezpečí.

Problém posielania správ cez sockety. V predchádzajúcom texte padlo rozhodnutie na komunikáciu medzi modulmi využívať sockety. Vzniká teda problém ako preniesť perlovú štruktúru cez socket, ktorý dokáže dáta prenášať len v binárnom formáte. V programovaní sa často používa pojem serializácia. Tento proces slúži na ukladanie dát do prenositeľnej formy, napríklad do textového reťazca. Opačným procesom je deserializácia, ktorá naopak z textového reťazca vytvorí použiteľnú dátovú štruktúru. Ako riešenie nastoleného problému sa teda ponúka práve spomínaná serializácia a deserializácia. Ako obvykle prišiel výber z viacerých riešení:

1. Rekurzívne prejsť štruktúru správy a nejakým spôsobom zapísať dáta do reťazca. Mohlo by to spôsobovať isté problémy vzhľadom k tomu, že štruktúra správy môže byť pomerne rozvetvená a niektoré jej časti nemusia byť presne definované, teda môžu mať obsah.

2. Druhou, oveľa jednoduchšou možnosťou, bolo využitie niektorého z existujúcich nástrojov. Perl tu ponúka niekoľko možností. Súčasťou základnej distribúcie je balík `Data::Dumper`, ktorý danú funkčnosť poskytuje[5]. Existujú aj ďalšie balíky, ale oproti `Data::Dumper` majú niekoľko nevýhod. Nie sú súčasťou základnej distribúcie, teda nemusia byť aktuálne a chyby v nich sa nepovažujú za kritické pre Perl. Nedá sa spoľahnúť na rýchlu opravu. Modul `Data::Dumper`, je navrhnutý na ukladanie perlových štruktúr do reťazcov. Z analyzovaných pre a proti mi vyšiel najlepšie modul `Data::Dumper`, keďže mi v tomto prípade viacej záleží na stabilite a funkčnosti na všetkých perlových platformách.

Z uvedených riešení som sa rozhodol pre druhé ponúkané. Riešenie je štandardné, ponúkané priamo Perlovou distribúciou a vyžaduje najmenej účasti na implementácii a teda najmenšiu možnosť chybovosti.

3.3 Modul Automat

Základom práce má byť implementácia jazyka SCXML. Doteraz som rozoberal analýzu návrhu práce ako celku a je na čase prejsť k analýze konkrétnych problémov, ktoré sa objavujú pri implementácii SCXML.

Členenie postupu. Postup interpretácie sa dá rozpísať do nasledujúcich krokov.

1. Nahranie XML súboru so zdrojovým kódom a jeho následné spracovanie.
2. Inicializácia prostriedkov na interpretáciu SCXML.
3. Vytvorenie štruktúr potrebných pre interpretáciu SCXML.
4. Interpretácia SCXML a komunikácia s centrálnou autoritou.
5. Ukončenie interpretácie a uvoľnenie prostriedkov.

Prvý krok. V prvej fáze je potrebné súbor otvoriť a sprístupniť, zatiaľ čo v druhej rozparsovať XML štruktúru a prispôsobiť ju pre jednoduché vytvorenie štruktúr, potrebných k samotnému interpretovaniu jazyka.

Bolo potrebné nájsť vhodný už implementovaný parser, ktorý by splnil úlohu spracovania XML súboru. Keďže všetky parsery pre Perl sú v istom ohľade špecifické, či už svojou stabilitou, alebo výsledkom spracovania, rozhodol som sa rozdeliť úlohu nahrania a spracovanie na dve časti a to s použitím prostredníka. Prostredníkom nazývam triedu `XMLParser`, ktorá v podstate neparsuje XML štruktúru, ale spracuje výsledok po nejakom použitom parseri. Úloha je teda rozdelená na nasledujúce časti:

- Použitý XML parser nahrá súbor z disku, analyzuje jeho štruktúru a vráti výsledok spracovania. V tejto fáze zároveň prebieha prípadná validácia xml dokumentu pomocou xsd schémy, ak je nastavená v SCXML aplikácii.
- Trieda `XMLParser` analyzuje štruktúru a pripraví ju do stavu vhodného na predspracovanie samotným interpreterom.

Mohlo by sa zdať, že trieda `XMLParser` tu vystupuje zbytočne. Pre jej použitie som sa rozhodol z dôvodu zachovania istej úrovne abstrakcie nad technikou parsovania XML súboru. Vďaka nej je možné použiť akýkoľvek XML parser, pričom nie je nutné meniť kód interpreteru.

Iným riešením by bolo priamo implementovanie podpory jediného parseru, kde by však v prípade problémov v jeho implementácii, ktoré by autori mohli nechať nepovšimnuté, nastali vážne komplikácie, ktoré by mohli vyústiť v nutnosť naimplementovať vytváranie štruktúr z nasledujúceho kroku od počiatku.

Druhý krok. Pred spracovaním XML štruktúry do interpretovateľného stavu je potrebné inicializovať komponenty využívané intepreterom. Menovite je to hlavne skriptovací modul, ktorý sa stará o vytvorenie príslušného dátového modelu a vyhodnotenie *location expressions* a *value expressions*. K zisteniu, aká konkrétna implementácia modulu sa aktivuje, je potrebné využíť konfiguráciu, ktorá je prístupná hneď po inicializácii modulu. Nutnosť inicializovania skriptovacieho modulu vyplýva z povinnosti interpreteru vytvoriť dátový model pred spustením samotnej SCXML aplikácie. Celému skriptovaciemu modulu bude venovaná jedna z nasledujúcich kapitol, keďže sa jedná o veľmi dôležitú a flexibilnú triedu zabezpečujúcu podstatnú časť fungovania interpreteru.

Tretí krok. V tomto kroku sa prevedie výsledok po spracovaní triedou `XMLParser` do štruktúr, ktoré využívam vo fáze interpretácie SCXML aplikácie. Do úvahy prichádzali nasledujúce riešenia:

1. Využiť už existujúcu stromovú štruktúru z kroku číslo dva a interpretovať v nej sa nachádzajúce data. Čo by so sebou prinášalo výhodu nevytvárania nových štruktúr, nižšiu spotrebu pamäťových zdrojov systému a rýchlejší štart aplikácii. Zároveň by to však viedlo k menším možnostiam optimalizácie v kóde interpreteru a v dôsledku tohto faktu aj pravdepodobne nižším rýchlostiam interpretácie aplikácii.

2. Ako ďalšia technicky komplikovanejšia možnosť sa javí previesť XML štruktúru na triedy a štruktúry optimalizované pre spracovanie. Hlavnú výhodu tu vidím v možnosti upraviť si triedy podľa seba, čo zabezpečí jednoduchší návrh a implementáciu samotného procesu interpretovania SCXML aplikácie. Nesie to so sebou spomenuté nevýhody vyšších pamäťových nárokov a nutnosti XML predspracovať, z čoho plynie relatívne pomalší štart aplikácii.

Po zvážení výhod a nevýhod som sa rozhodol pre druhé riešenie, teda jednoduchší a prehľadnejší návrh a implementáciu za cenu vyšších pamäťových nárokov a pomalšieho procesu spúšťania aplikácii.

Štvrtý krok. Samotná interpretácia SCXML aplikácie prebieha na základe algoritmu definovaného špecifikáciou. Ako som už v kapitole o SCXML spomínal algoritmus nie je veľmi dobre popísaný. Vďaka tomuto faktu, som si musel pomerne veľa veci vyhľadať v

špecifikácii pri konkrétnych elementoch, prípadne navrhnúť sám. Algoritmus poskytuje akúsi základnú kostru, podľa ktorej sa aplikácia musí správať, aby spĺňala podmienky prenositeľnosti. V špecifikácii je tento algoritmus popísaný vymysleným jazykom s podobnou syntaxou akú má funkcionálny jazyk LISP.

Hrubé pojatie behu aplikácie rozdelené na kroky:

1. Prebieha tu vytvorenie náhodných hodnôt pre systémové položky, ktoré nedefinoval programátor, alebo sú priamo určené na generovanie systémom. V súčasnej špecifikácii sú len tri systémové premenné a to `_event`, `_sessionid` a `_name`. `_event` sa definuje až pri spracovávaní vzniknutej udalosti a dovtedy musí ostať nenastavená. Premenná `_sessionid` je vždy v tomto kroku generovaná systémom. Posledná premenná `_name` sa nastavuje ako parameter v elemente `_scxml` a v prípade, že jej nebola priradená žiadna hodnota, nastavím ju na prázdny reťazec.

2. Prvotné spustenie algoritmu bez čakania na externú udalosť. Podľa špecifikácie sa musí ako prvý vykonať prechod z elementu `scxml` do stavov, ktoré špecifikuje. Prechod sa môže zapísať spôsobom ako parameter `initialstate`, alebo pseudostav `initial` s definovaným prechodom bez parametru udalosti a podmienky. Následne sa spracujú všetky interné udalosti vyvolané príchodom do definovaných stavov, alebo spustiteľným obsahom prechodu. Algoritmus sa zastaví v prípade, že všetky interné udalosti sú už spracované. Externé udalosti sa ešte nemôžu vyskytovať, pretože systém ich v tomto kroku ešte neprijíma od autority `MessagePool`.

3. V tomto kroku sa čaká na príchod externej udalosti, ktorá sa spracuje a následne sa spracúvajú interné udalosti, ktoré môžu jej príchodom vzniknúť, rovnakým spôsobom ako v predchádzajúcom prípade. Po spracovaní externej udalosti a následne všetkých interných prechádza automat opäť do čakania na externú udalosť.

4. Ukončenie aplikácie. Môže prebiehať dvoma spôsobmi:

   Automat príde do finálneho v scxml elemente a ukončí činnosť, pričom požiada `MessagePool` o ukončenie behu celého systému.

   Príde správa od `MessagePool`-u, ktorá ukončuje činnosť celého systému a automat bude nútený správu spracovať a činnosť ukončiť. Riešením na odchytenie tejto situácie v každej SCXML aplikácii je vytvoriť stav, ktorý ju celú zapuzdruje a v ňom prechod, ktorý vedie do finálneho stavu v elemente `scxml`, a zároveň zachycuje udalosť ukončenia modulu Automat. V opačnom prípade bude modul Automat stále čakať na externé udalosti a ukončený bude násilím pri vypnutí hlavného vlákna systému.

Tesne pred ukončením aplikácie prichádza na rad uvoľňovanie systémových prostriedkov využívaných pri interpretáciu algoritmu. V jazyku Perl to našťastie nebýva veľkým problémom a rovnako je tomu tak aj teraz. Perl neposkytuje prostriedky na uvoľňovanie

štruktúr a stará sa o to prostredníctvom garbage collectoru sám. Teda po skončení interpretácii stačí ukončiť ostatné moduly, ktoré v prípade potreby mohli využívať nejaké externé knižnice, či aplikácie.

3.4 Skriptovací modul a dátový model

Dôležitou súčasťou implementácie SCXML je vyhodnocovanie výrazov typov *location expression* a *value expression*, obsahu elementu script, či uchovávania dát.

3.4.1 Skriptovací modul

Od začiatku navrhovania práce som bol rozhodnutý pre využitie jazyku Perl na vyhodnocovanie častí, ktoré má na starosti *script module* známy zo SCXML. Pri navrhovaní implementácie tejto časti interpreteru mi ale prišlo zbytočné obmedzovať sa na jeden využiteľný skriptovací jazyk a rozhodol som sa vytvoriť vrstvu abstrakcie, ktorá umožní jednoduchým spôsobom implementovať podporu hocijakého skriptovacieho jazyka bez nutnosti meniť kód samotného interpreteru. Táto vrstva je v práci spomínaná ako skriptovací modul.

Ako som spomínal o implementáciu konkrétneho skriptovacieho jazyka sa stará konkrétny skriptovací modul navrhnutý pre tento jazyk. Jedinou podmienkou je splnenie spoločného interface zastúpeného triedou `Modules::Automat::Script::Base`. Interface obsahuje metódy pre vyhodnocovanie podmienok, kódu písaného v danom skriptovacom jazyku. V týchto metódach sprístupňuje položky dátového modelu podľa špecifikácie pomocou špeciálnej premennej `_data`. Vo všeobecnosti kód, ktorým sa metódy vyhodnocujú sa v špecifikácii označuje ako *data and value expressions*.

Skriptovací modul by sa však mal starať ešte o jednu veľmi dôležitú funkčnosť a tou je vyhodnotenie umiestnenia v rámci dátového modelu, teda *location expression*. V tomto smere som skriptovaciemu modulu značne uľahčil a vytvoril som špeciálnu sadu tried na vyhodnocovanie tohto typu výrazov. Spĺňajú vlastný špeciálny interface, ktorý umožňuje zistiť, či premenná existuje, kopírovať časti dátového modelu, či priradzovať do premenných v dátovom modeli. Premenné sa v interface identifikujú pomocou *location expression* kvôli transparentom prenose výrazov zo SCXML. Táto vetva návrhu je však voliteľným doplnkom a od programátorov implementujúcich vlastné skriptovacie moduly nie je nijako vyžadované držať sa praktiky oddeľovania vyhodnocovania *location expressions* od ich skriptovacieho modulu.

3.4.2 Dátový model

So skriptovacím modulom je pevne spojená komponenta zvaná dátový model, v špecifikácii zvaná *data module*. Ich vzájomná väzba vyplýva z nutnosti vyhodnocovania výrazov pomocou skriptovacieho modulu. Túto komponentu, známu zo SCXML špecifikácie, som sa však rozhodol vynechať. Jej funkcia spočíva v práci s premennými. Vzhľadom k možnému problematickému prepojeniu jednej implementácie uchovávania dát na rôzne skriptovacie jazyky som ponechal túto funkciu plne v moci skriptovacieho modulu. Zastúpená je sadou metód na prácu s premennými, ktoré implementuje skriptovací modul. Jeho povinnosťou je uchovávať hodnoty uložené v jeho implementácii dátového modelu medzi jednotlivými volaniami jeho metód, teda dátový model nesmie nikdy zaniknúť.

3.4.3 Scope

Pri práci s premennými v SCXML existuje špeciálna vlastnosť nazývaná *scope*. Zabezpečuje zapuzdrenie premenných do určitého úseku spracovania. V SCXML sa delí na dva druhe: lokálny a globálny *scope*.

Premenné, ktoré vzniknú v oblastiach s lokálnym *scope* musia po ukončení spracovaní tejto oblasti zaniknúť. V mojej implementácii som to vyriešil jednoduchým spôsobom. Keďže skriptovací kód je písaný v jazyku Perl a spracovaný pomocou príkazu `eval` premenné, ktoré v ňom vzniknú po ukončení bloku spracovania automaticky zaniknú.

Oproti tomu, premenné, ktoré vzniknú v oblastiach s globálnym *scope* si musia udržiavať svoju hodnotu aj mimo tejto oblasti. Riešenie tejto situácie ponúka samotné SCXML, ktoré prikazuje ukladať premenné do dátového modelu a sprístupňovať ich pomocou špeciálnej premennej `_data`. Túto premennú som realizoval ako referenciu na uložený dátový model a teda zmeny, ktoré sa na nej realizujú sa prejavia priamo na dátovom modeli, ktorý je skriptovací modul povinný prechovávať medzi jeho jednotlivými volaniami.

3.5 Štruktúry pre interpretáciu

Dôležitou súčasťou analýzy bolo zvolenie vhodnej reprezentácie štruktúr známych zo SCXML špecifikácie tak, aby sa čo najlepšie splnili požiadavky kladené v treťom kroku zo sekcie 3.3 Modul Automat.

3.5.1 Perlové štruktúry

Jazyk Perl je pomerne chudobný na typy podporovaných premenných, zato však s nimi umožňuje veľmi flexibilnú prácu. Podporuje len tri typy:

Scalar môže obsahovať čísla, reťazce a referencie na všetky perlové typy.

Array je reprezentácia poľa známeho z iných programovacích jazykoch. Jeho prvkay môžu byť typu `scalar`.

Hash zastupuje hash mapu, kde sa prvky identifikujú pomocou kľúča a môžu nadobúdať hodnoty typu `scalar`. Často sa využíva ako zastúpenie typu záznam, známeho z iných programovacích jazykov.

3.5.2 Konfigurácia

V automatoch reprezentovaných pomocou SCXML je celá dynamika aplikácie sústredená okolo stavov, udalostí a prechodov. Automat má aktívne stavy a vyčkáva na udalosť, ktorá môže aktivovať nejaký prechod z aktívneho stavu do iného stavu, alebo do toho istého. Tento jednoduchý popis umožňuje zachytiť aktuálny stav automatu do *konfigurácie*. Konfigurácia je zoznam `id` príslušných ku stavom, ktoré sú v automate aktívne, teda tie, ktorých prechody môžu reagovať na prichádzajúce udalosti. Štruktúra charakterizuje momentálnu situáciu automatu a využíva sa na uchovanie stavu medzi spracovaniami udalostí. Požiadavky kladené na túto štruktúru sú nasledujúce. Rýchle zistenie prítomnosti id stavu v štruktúre a zároveň nutnosť vedieť ju prechádzať postupne pričom záleží na poradí prvkov. Pri pohľade na perlové základné typy je jasné, že žiaden túto požiadavku nesplňuje. `Hash` síce umožní konštantné vyhľadávanie, ale poradie sa nedá ovplyvniť a je také, ako ho zvolí Perl. `Array` má poradie prvkov podľa užívateľa, ale vyhľadávanie je v čase n. Ideálne by teda vychádzala kombinácia dobrých vlastností týchto typov. Rozhodol som sa vytvoriť triedu `Hybrid`, ktorá umožní rýchle zistenie existencie a zároveň zachová poradie prvkov. Trieda vznikla kombináciou premennej typu `hash` a premennej typu `array`.

3.5.3 Stavy

Ako prvé bolo treba vyriešiť nejakú formu zoznamu stavov dostupných v aplikácii. Jedinou podmienkou bolo rýchle vyhľadávanie v štruktúre. Podľa špecifikácie sa v SCXML aplikáciach stav identifikuje pomocou svojho parametru `id`. Povolenou hodnotou pre tento parameter je akýkoľvek textový reťazec, ktorý má vlastnosť, že je v rámci všetkých `id` unikátny. Po krátkom zhodnotení problému, som sa rozhodol použiť typ `hash`. Jeho prvky môžu nadobúdať hodnoty hocijakého základného perlového typu a sú identifikované pomocou kľúča, ktorý môže byť aj reťazcom. Kritérium rýchleho vyhľadávania je splnené, vďaka tomu, že hash poskytuje konštantnú časovú zložitosť.

Ako bolo popisované vyššie, typ stav je v SCXML aplikáciach základnou stavebnou jednotkou, od ktorej sa odvíjajú ostatné štruktúry. Na rad prichádza rozhodnutie ako charakterizovať štruktúru tohto typu. Keďže typ stav má pomerne veľa atribútov, nedá sa uvažovať o typoch `array` ani `scalar`. Zo základných typov jedinou možnosťou ostáva typ `hash`, ktorý je určený práve na vytvorenie štruktúry charakterizovanej dvojicami kľúč

a hodnota. Pričom ako kľúč by sa použili názvy atribútov a ako hodnoty, ich hodnoty. V jazyku Perl má však táto štruktúra jednu pomerne veľkú nevýhodu. Pri vypisovaní kľúča je možné sa pomýliť v jeho mene a Perl namiesto zahlásenia chyby vytvorí nový prvok s nedefinovanou hodnotou. Na tento problém sa tu ponúkajú dve dostatočne prijateľné riešenia:

1. Pre každý atribút vytvoriť funkciu, ktorá ho nastavuje a funkciu, čo ho vyberá. Týmto spôsobom by odpadlo vypisovanie mien atribútov a tým pomerne častá chyba v následnej implementácii. Viedlo by to však k veľkému množstvu funkcií a značnému zneprehľadneniu zdrojového kódu.

2. Vytvoriť triedu pre typ stav. Tento prístup prináša všetky výhody predchádzajúceho riešenia a naviac vďaka OOP taktiež zapuzdrenie. Keďže všetky tri typy stavu známe zo SCXML majú pomerne veľa spoločných atribútov a zároveň niektoré odlišné, toto riešenie mi umožní typy stavu pekne charakterizovať pomocou dedičnosti.

Po zvážení všetkých pre a proti pri oboch variantách som sa priklonil k druhému ponúkanému riešeniu.

3.5.4 Prechody

Ďalšou základnou stavebnou jednotkou sú prechody. Keďže priamo súvisia s konkrétnym stavom a typ stav je triedou, vhodným riešením sa ponúka použiť prechod ako atribút stavu. Stav môže mať prechodov viacej a zároveň pri nich záleží na poradí, pretože sa kontrolujú v poradí, v akom sú zapísané v dokumente, v špecifikácii často zmieňované ako *document order*, a aktivuje sa maximálne jeden pre každú udalosť. Prechody podľa špecifikácie nemajú žiaden jednoznačný identifikátor, rozpoznávajú sa iba podľa stavu, v ktorom sú umiestnené, a svojho poradia v rámci stavu. Pri pohľade na dostupné perlové typu sa javí ako priamo ideálne riešenie využiť základný typ `array`, ktorý spĺňa podmienku zachovania poradia a zároveň nie je treba prvky nijako identifikovať. Týmto je vyriešená reprezentácia prechodu v rámci stavu. Ostáva doriešiť, ako bude vyzerať štruktúra samotného prechodu. Podľa špecifikácie prechody majú zopár parametrov a ako potomkov môžu mať spustiteľný obsah. Rovnako ako pri stavoch sa tu objaví viac možností návrhu. Keďže prechod má pomerne málo parametrov a existuje iba jeden druh prechodu, výhody objektov sa veľmi nevyužijú. Zároveň sa prechody využívajú iba na pár miestach, pri testovaní podmienok prechodu a pri prechádzaní medzi stavmi. Rozhodol som sa ho zbytočne nekomplikovať a navrhnúť ako základný perlový typ `hash`.

3.5.5 Udalosti

Udalosti sú v SCXML pevne definované štruktúry, ktorých implementácia je platformne závislá. Podľa zápisu v špecifikácii, by im však najviac vyhovoval typ `struct`,

známy z iných programovacích jazykov. V jazyku Perl ponúka základný typ hash ekvivalentnú funkčnosť. Bez väčšej analýzy som sa rozhodol pre využitie tohto dátového typu. V udalostiach je povinná položka `data`, ktorá slúži na prenášanie k udalosti pripojených dát. Vzhľadom na povahu tohto parametra zo špecifikácie sa naňho rovnako hodí základný perlový dátový typ `hash`. Ostatné parametre sú implementované pomocou základného typu `scalar`, keďže predstavujú jednoduché hodnoty: čísla, alebo reťazce.

3.5.6 Spustiteľný obsah

Jednou z dôležitých súčastí SCXML sú elementy spustiteľného obsahu. Zastávajú základné interaktívne procesy ako generovanie udalostí, posielanie udalostí mimo automat, prácu s premennými a logovanie. Naviac je podľa špecifikácie možné túto sekciu dopĺňať o nové platformne špecifické prvky a tým dodávať ďalšiu funkčnosť a možnosti programátorom aplikácii. Tieto elementy medzi sebou bohužiaľ nemajú žiadne spoločné vlastnosti, iba ich umiestnenie v rámci XML dokumentu. Vďaka rovnakému umiestneniu plnia podobnú funkciu a ich zhrnutie do jednotného celku je na mieste. U elementoch tohto typu záleží na poradí vyhodnocovania a nemajú žiadny vlastný atribút, ktorý by ich identifikoval v rámci dokumentu. Ich identifikátorom je umiestnenie v určitej sekcii a zároveň ich poradie. Vzhľadom na spôsob určenia, ktorý tieto prvky využívajú, rozhodol som sa pre perlový typ `array` ako implementáciu sekcie takýchto prvkov.

Ostáva otázka ako reprezentovať samotné prvky spustiteľného obsahu. Na túto tému som previedol analýzu s nasledujúcimi riešeniami:

1. Jednoduchý základný typ `hash`. Umožňuje jednoduché vkladanie parametrov elementu, dvojice názov-hodnota. V prípade komplikovanejších elementov s potomkami, napríklad element `send`, je možné potomkov reprezentovať ako referenciu na typ `hash`, alebo typ `array`. Oproti prechodom, kde sa tento postup bez problémov uplatnil, sa tu vyskytuje jednoznačná nevýhoda v tom, že typ `hash` svoje položky nezapuzdruje a nie je možné k nemu písať metódy. U prechodov to bolo nedôležité, keďže typ prechod je len jeden a práca s ním je iba v niektorých častiach interpreteru. Na rozdiel od neho však prvkov spustiteľného obsahu je veľa druhov a predpokladá sa, že časom budú pribúdať nové. Nevýhoda tu spočíva v nutnosti zmeniť kód načítavania a vyhodnocovania prvkov pri každom novom podporovanom elemente.

2. Typ trieda spolu s manažérom prvkov spustiteľného obsahu. Myšlienka je nasledujúca: pre každý typ prvku spustiteľného obsahu by bola navrhnutá a implementovaná trieda, ktorá by ho vedela z poskytnutej stromovej štruktúry načítať a pri spracovaní by ho mohla vyhodnotiť. Načítanie a spracovanie by prebiehalo v definovanom interface a teda všetky triedy by mali spoločného predka a navonok rovnaké správanie. Manažér by sa staral pri nájdení dosiaľ nepoužitého elementu o nahranie príslušnej triedy zo súboru, jej vytvorenie a zavolanie metód spomenutého interfacu. V

prípade, ak by trieda neexistovala, mohol by manažér podľa nastaveného módu interpretovania vyhlásiť chybu, alebo ju ticho potlačiť. Týmto spôsobom by sa dosiahlo určitej vrstvy abstrakcie, vďaka ktorej by bolo možné nové elementy implementovať len na základe pridania príslušnej triedy do projektu bez zmeny zdrojového kódu interpreteru.

Zo spomínaných dvoch riešení som sa dočasne rozhodol pre prvé, kvôli jednoduchosti implementácie na obmedzenom počte dostupných prvkov sekcie spustiteľného obsahu. V budúcnosti plánujem podrobne navrhnúť a implementovať druhé riešenie, pretože je perspektívnejšie z hľadiska rozšíriteľnosti projektu. Síce je možné rozširovať aj pri prvom spôsobe, ale spôsobí sa tým značné zneprehľadnenie zdrojových kódov projektu.

3.6 Podrobnejší popis algoritmu na interpretáciu

Pred návrhom samotného algoritmu je treba definovať niekoľko štruktúr, ktoré sa v špecifikácii používajú a ja ich budem využívať tiež, či už v špecifikácii zaužívanej, alebo vlastnej podobe.

Internal queue - fronta pre interné udalosti. Uchovávajú sa tu udalosti, ktoré vzniknú vyvolaním elementu `raise`, alebo špecifický prípad elementu `send`, prípadne ich ekvivalentným zápisom v skriptovacom jazyku.

External queue - fronta pre externé udalosti. Táto fronta v podstate nie je implementovaná. Prišlo mi zbytočné čítať správy zo socketu a ukladať ich niekam ako udalosti v osobitnom vlákne, keď je možné správy ponechať v sockete, kým SCXML aplikácia nebude chcieť ďalšiu externú udalosť. V tomto prípade sa správa presunie zo socketu do aplikácie ako udalosť. Keďže čítanie zo socketu je blokujúce, výber udalosti z tejto fronty je rovnako blokujúci.

Configuration - konfigurácia je zoznam id stavov, ktoré sú aktívne a ich prechody môžu byť aktivované aktuálnou udalosťou.

LCA - prvý spoločný predok. Predstavme si, že stavy sú v SCXML organizované do stromovej štruktúry rovnako ako XML elementy. Koreňom je element `scxml`, ktorý reprezentujem pre túto potrebu ako stav. Nech s1, s2 sú stavy v takomto strome. Potom LCA je prvý uzol na ceste od s1 ku koreňu, ktorý má zároveň potomka s2.

Samotný algoritmus postupuje nasledujúcim spôsobom:

1. Čaká, kým zo socketu dorazí správa.
2. Došlú správu rozparsuje pomocou perlového príkazu `eval` do štruktúry udalosti.

3. Udalosť overuje voči všetkým prechodom vo všetkých aktívnych stavoch postupne podľa poradia v dokumente a od listov až ku koreňu. Vždy najprv prejde všetky listy a tak sa posunie o vrstvu vyššie. Overovanie prebieha porovnávaním názvu udalosti zo sekcie 2.2.4 Udalosti. Následnej kontroluje, či je splnená podmienka udalosti. Ak kontrola prebehne úspešne, uloží prechod do zoznamu aktivovaných prechodov.

4. Aktivuje cyklus, ktorý beží pokiaľ sú nejaké dostupné nespracované udalosti.

    1 Opustí všetky stavy, ktoré je treba opustiť, aby sa mohlo pomocou aktivovaných prechodov prejsť do cieľových stavov. Postup je nasledujúci:

        1 Nájde `LCA` pre rodičovský stav a všetky cieľové stavy. Je potrebné opustiť všetky stavy, ktoré sú v pomyselnom stavovom strome vo vetve smerom od `LCA` dole, v ktorej je aj rodič, a zároveň sa nachádzajú v aktuálnej konfigurácii.
        2 Vytvorí zoznam stavov, ktoré sú v pomyselnom stavovom strome medzi rodičom `LCA` a vyberie z neho prvého potomka `LCA`.
        3 Z neho pomocou rekurzie prejde až k ku nejakému koncovému stavu, ktorý je zároveň v aktuálnej konfigurácii. Cestou naspäť pre všetky tieto stavy spúšťa ich spustiteľný obsah nachádzajúci sa v elemente `onexit` a následne ich maže.
        4 Postup sa opakuje pre všetky aktivované prechody.

    2 Pre každý prechod spustí jeho spustiteľný obsah, ak je dostupný.

    3 Vojde do všetkých stavov, ktoré sú ako `target` pre aktivované prechody. Postup je nasledujúci:

        1 Pre všetky prechody a pre všetky ich `id` stavov v parametri `target` vojde do daného stavu.
        2 Príchod do stavu sa líši podľa typu stavu. V každom prípade na začiatku pridá stav do aktuálnej konfigurácie. Pokiaľ stav sa už v konfigurácii nachádzal, ďalej sa v tejto vetve nepokračuje.
            1 V prípade, že je stav typu Atomic, teda stav definovaný elementom `state` a bez potomkov typu stav, vnorí sa rekurzívne do jeho rodiča a po návrate spracuje spustiteľný obsah vo vlastnom elemente `onentry`.
            2 Ak je stav typu `Parallel`, teda stav definovaný elementom `parallel`, vnorí sa rekurzívne do svojho rodiča. Po návrate spracuje vlastný spustiteľný obsah v elemente `onentry` a následne sa vnorí do všetkých svojich potomkov.
            3 V prípade, že sa jedná o stav typu `Compound`, teda stav definovaný elementom `state` a obsahujúci aspoň jedného potomka typu stav. Vnorí sa rekurzívne do svojho rodiča a po návrate spracuje spustiteľný obsah vo svojom elemente `onentry`. Následne prejde svojich potomkov typu stav a zistí, či aspoň jeden je v konfigurácii. Robí to preto, že stav

typu `Compound`, môže mať aktívneho maximálne jedného priameho potomka typu stav. V prípade, že žiaden z jeho priamych potomkov sa v aktuálnej konfigurácii nenachádza, vojde do potomka, ktorý je určený iniciálnym prechodom tohto stavu.

4 Ak sa jedná o stav typu `Scxml`, ktorý zastupuje element `scxml`, správa sa stav obdobne ako stavy typu `Compound`, akurát sa nevchádza do rodiča, pretože žiaden neexistuje.

3 Prejde všetky dostupné stavy v novej konfigurácii a ak sú nejaké prechody bez definovanej udalosti, ktoré majú splnenú podmienku, tak ich označí za aktívne a opakuje predchádzajúci postup.

4 V prípade, že už žiadne prechody spĺňajúce podmienky predchádzajúceho bodu nie sú, pokúsi sa vybrať udalosť z internej fronty. Keďže operácia čítania z internej fronty je neblokujúca, okamžite sa zistí, či tam nejaká udalosť je. V prípade, že áno, opakuje sa celý predchádzajúci postup od bodu 3. s tým, že neaktivuje cyklus, pretože sa v ňom už nachádza. Ak žiadna taká udalosť neexistuje pokračuje sa ďalším krokom.

5 Cyklus sa ukončí.

6 Skontrolujú sa podmienky ukončenia aplikácie. Tie sú nasledujúce:

1 V aktuálnej konfigurácii sa nachádza stav typu `final`, ktorý je zároveň priamym potomkom elementu `scxml`.

2 Prišla správa od autority `MessagePool`, ktorá ukončuje činnosť modulu.

Ak je aspoň jedna podmienka splnená, opustia sa všetky stavy aktuálnej konfigurácie a modul Automat ukončí svoju činnosť. V opačnom prípade sa začína postup opäť úplne od začiatku, od bodu 1.

3.7 Modul TTS

Tento modul už celkom nesúvisí s implementáciou SCXML, je však doplnkom, ktorý ostáva z implementácie v ročníkovom projekte. TTS je skratkou za výraz *Text To Speech*, ktorý označuje aplikácie slúžiace na syntézu reči zo zadaného textu. Modul je prepojený s interpretáciou v module Automat, ktorý dokáže spracovať sadu elementov spustiteľného obsahu využívaných na prácu so syntézou textu práve pomocou tohto modulu.

V pôvodnom návrhu sa nevyskytoval modul ako samostatný celok, ale jeho funkčnosť bola rozdelená do viacerých modulov, menovite `Festival` a `gbsoft`, podľa toho akú aplikáciu sa rozhodol programátor na syntézu využívať. V podstate bol pôvodný návrh funkčný, akurát pri testovaní na oboch podporovaných OS, bolo potrebné prepisovať SCXML aplikácie, aby využívali správny modul, pretože rôzne *TTS engine* nemusia spĺňať podmienku multiplatformnosti. Kvôli tejto problémovej situácii som sa rozhodol vytvoriť

nad oboma modulami vrstvu abstrakcie, ktorá by poskytovala rovnakú funkčnosť a zároveň bola dostatočne konfigurovateľná pre špecifické nastavenia konkrétnej aplikácie, či umožňovala rozšírenia o nové *TTS engine* bez významného zásahu do zdrojových kódov.

3.7.1 Interface

Modul teda musí poskytovať interface, ktorý bude dostatočne prispôsobený na jednoduché príkazy obecného charakteru, ako je vysloviť niečo a zároveň umožní predávať parametre špecifické pre jednotlivé aplikácie, ktoré zastrešuje. Súčasťou interface musí byť aj sada odpovedí na jednotlivé požiadavky. Tento interface bol vytvorený čo najpružnejšie, teda neobsahuje žiadne špecifické názvy, iba syntax, akou sa musia zapisovať.

Výsledkom tohto interface je element `tts`, ktorý je možné využívať ako súčasť spustiteľného obsahu v SCXML aplikáciach.

3.7.2 Engine

Pri navrhovaní spôsobu implementácie engine do modulu boli po ruke viaceré možnosti:

1. Jednoduché vetvenie programu na základe názvu `engine`. Ako výhoda je priamočiarosť implementácie tohto postupu. Nevýhoda spočíva v obtiažnej rozšíriteľnosti a značnej neprehľadnosti riešenia.

2. Pre každý `engine` je implementovaná trieda, ktorá spracuje informácie a následne môže zaslať odpoveď. Výhoda je jednoduchá rozšíriteľnosť a dobrá štruktúrovanosť programu. Ako mínus tu je väčšia komplikovanosť implementácie.

Z ponúkaných riešení som sa rozhodol pre riešenie prvé. Keďže podporované engine sú momentálne tri a spracovávaná metóda len jedna, navrhovať a implementovať komplikovanejšie rozhranie je pomerne zbytočné. V budúcnosti je samozrejme možné, že sa prejde ku druhému riešeniu v prípade, ak bude nutné tento modul ďalej rozširovať.

Konkrétne engine. Súčasná implementácia podporuje engine `Festival` a `gbsoft`, pričom `Festival` je implementovaný v dvoch módoch. Jeden je mód *konzolový*, ku ktorému je potrebný len spustiteľný súbor aplikácie. Druhý mód je *serverový*, kde je potrebné mať `Festival` nainštalovaný a spustený ako server. `Festival` je engine, ktorý je opensource a mal by byť spustiteľný na oboch platformách[4]. Mne sa to zatiaľ pod Windows nepodarilo, ale keďže sú dostupné zdrojové kódy malo by to byť možné. Engine `gbsoft` je komerčnou aplikáciou, ktorá rozpráva po česky a jeho spustenie je možné len pod platformou Windows. Oba engine je nutné pripraviť pred tým, ako ich bude systém využívať.

3.8 Modul Sphinx

Nad rámec práce som sa rozhodol pokúsiť vytvoriť podporu rozpoznávania reči. Témy rozpoznávanie reči a syntézy texu sa mi páčia, preto mi táto funkcia prišla ako zaujímavý doplnok k práci. Pre túto súčasť som zvolil vyššie spomínaný software *CMU Sphinx 4*. Modul Sphinx zabezpečuje jeho integráciu do projektu pomocou určitého rozhrania. *CMU Sphinx 4*nie je kompletnou aplikáciou určenou na rozpoznávanie reči, ale frameworkom, v ktorom sa tieto aplikácie programujú[3]. Z tohto dôvodu som sa rozhodol vytvoriť rozhranie, čo bude funkciu rozpoznávania reči spĺňať a aplikácie naprogramované *v CMU Sphinx 4*, ho budú musieť implementovať. Rozhranie obsahuje nasledujúce funkčnosti:

- Začiatok nahrávania zvuku – od Sphinx aplikácie sa predpokladá, že začne nahrávať reč z mikrofónu.
- Koniec nahrávania zvuku – aplikácia má ukončiť nahrávanie reči z mikrofónu.
- Rozpoznanie reči v nahratej časti zvuku – prebehne rozpoznanie reči z nahratej časti zvuku na základe aktuálnej konfigurácie a výsledok sa zašle v podobe textového reťazca.
- Spustenie novej konfigurácie zo súboru – aplikácia má nahrať novú konfiguráciu zo súboru, ktorý je poskytnutý formou cesty k nemu v zaslanom reťazci.
- Ukončenie aplikácie – predkladá sa ukončenie Sphinx aplikácie.

Metódy ďalej nebudem rozpisovať, pretože funkčnosť tohto modulu je zatiaľ v pomerne rannom štádiu vývoja. Zatiaľ je schopný komunikovať prostredníctvom tohto interface, ale niektoré časti návrhu ešte nie sú úplné doriešené. V súčasnej implementácii ho považujem za nevhodný na využívanie, ale predpokladám, že túto zaujímavú funkciu dokončím niekedy v blízkej budúcnosti.

Kapitola 4

Podrobný návrh

V tejto kapitole budem písať iba o detailoch častí, ktoré sú niečim významné. Buď som nad ich tvorbou strávil veľa času, alebo sú nutné k opisu celkovej funkčnosti systému.

4.1 Načítanie konfigurácie, trieda Loader

Interpret sa konfiguruje pomocou príkazového riadku, alebo konfiguračného súboru. Za načítanie konfigurácie je zodpovedná trieda `Loader`. Parametre príkazového riadku majú vyššiu prioritu ako zápis v konfiguračnom súbor. Z toho vyplýva možnosť využiť obe formy zápisu a pri rovnakom konfiguračnom súbore spúštať interpret s rôznymi parametrami prepisovaním príkazovej riadky.

- `konfiguračný súbor` Trieda `Loader` prevezme cestu v súborovom systéme k danému konfiguračnému súboru, ktorý je vo formáte *ini*. Následne vytvorí `hash`, ktorý má ako kľúče názvy jednotlivých sekcií a hodnoty sú referencie na `hash`, ktoré obsahujú páry kľúč-hodnota, rovnako ako sú zapísané v *ini* súbore v danej sekcii. Pri spracovávaní súboru sa berú do úvahy prázdne riadky a komentáre, teda riadky začínajúce znakom `;`. Pomocou metódy `get` je možné vybrať z požadovanej sekcie hodnotu konkrétneho kľúča. Metóda `getSection` zase vracia hash, ktorý obsahuje všetky páry kľúč-hodnota pre danú sekciu. Obdobne metóda `getAll` vráti `hash`, ktorý obsahuje celú vyššie popisovanú internú štruktúru.

- `parametre príkazovej riadky` Pomocou metódy `loadArray` sa do triedy `Loader` vloží pole obsahujúce všetky parametre príkazovej riadky vo formáte:
  `sekcia-kľúč=hodnota`.
  Tieto hodnoty sú následne parsované ako v predchádzajúcom prípade, teda sekcia, kľúč aj hodnota majú rovnaký význam ako keby boli zapísané v konfiguračnom súbore.

4.2 Komunikácia, trieda MessagePool

Ako som v kapitole o analýze popisoval, trieda `MessagePool` predstavuje v komunikácii centrálnu autoritu, ktorá sa stará o vytvorenie jednotlivých modulov a zabezpečí komunikáciu medzi nimi, pričom nie je nutné aby jednotlivé moduly vedeli o pripojení ostatných modulov.

Na začiatku sa vytvorí zoznam aktivovaných modulov z konfiguračného súboru, konkrétne z položky `load` v sekcii `Project`. Všetky položky sa postupne predajú triede `MessagePool` pomocou metódy `addModule`, ktorá triedu uvedomí o tom, že daný modul je potrebné aktivovať.

Následne sa zavolá metóda `load`, ktorá spustí všetky moduly a spracuje ich udalosti. Ak sa ukončí, znamená to koniec spracovania a tiež činnosti všetkých modulov. Táto metóda najprv vytvorí *serverový socket*. Potom postupne načíta všetky moduly tak, že pre každý modul vytvorí vlákno, v ktorom spustí metódu `mpModuleMain` a nasledovne čaká na pripojenie *klientského socketu* z daného modulu na svoj *serverový socket* a *klientský socket* uloží do `hash` pod kľúč s názvom pripojeného modulu.

Metóda `mpModuleMain`, vloží požadovaný modul do aplikácie pomocou príkazu `eval`, ktorý dokáže interpretovať perlový kód zadaný reťazcom- Vďaka čomu meno vkladaného modulu nie je potrebné vedieť v čase interpretácie/kompilácie programu. Následne vytvorí inštanciu triedy daného modulu a zavolá jej metódu `moduleInit`, ktorá vytvorí klientský socket modulu a pripojí sa ním na serverový socket `MessagePool`-u. Zvyšok času trávi v cykle, ktorý stále volá metódu inštancie modulu `execute`, ktorá vracia informáciu o tom, či sa má pokračovať, alebo či má cyklus skončiť. V prípade koncu cyklu sa ukončí celé vlákno a teda skončí činnosť modulu.

Po takomto spracovaní všetkých aktívnych modulov MessagePool zašle každému modulu štartovaciu správu pomenovanú `mpStart` s dátovou položkou `content`, ktora nadobúda hodnotou `now`.

Následne MessagePool vyčkáva na správy prichádzajúce od modulov, ktoré rozposiela pomocou metódy `resendMessage` na správne destinácie. V prípade ak je správa určená na destináciu `MessagePool`, je spracovaná priamou triedou `MessagePool`. V súčasnosti je implementovaná len jedna takáto správa. Nesie názov `shutdown` a oznamuje triede `MessagePool`, aby ukončila činnosť všetkých modulov a následne ukončila aplikáciu. Po obdržaní tejto správy MessagePool zašle na všetky moduly správu s názvom `_shutdown`, ktorá oznamuje modulu, že sa má ukončiť. Potom `MessagePool` čaká čas stanovený v konfiguračnom súbore a následne ukončí beh celej aplikácie.

4.3 Komunikácia, trieda Modul

S viacerými modulami bolo nepohodlné implementovať rozhranie na komunikáciu s centrálnou autoritou `MessagePool` pre každý osobitne. Teda podľa zásad *OOP* vznikol spoločný predok všetkých modulov, ktorý komunikačné rozhranie implementuje a tým poskytuje výhodu rovnakého spôsobu prijímania a posielania správ pre všetky moduly.

4.3.1 Inicializácia

Medzi zaujímavé časti patrí aj inicializácia modulu. Na ňu slúži metóda moduleInit, ktorá je volaná z hlavnej metódy vlákna, v ktorom je modul spustený. Obvykle to je `mpModuleMain` z triedy `MessagePool`. Inicializačná metóda ma za úlohu vytvoriť *klientský socket*, ktorý sa pripojí na predom stanovený port, ktorého číslo dodá `MessagePool`. Teda komunikáciu zahajuje modul pripojením sa k *serverovému socketu* `MessagePool`-u bežiacom na *localhoste*.

Metóda `mpModuleMain` tiež zabezpečuje volanie metódy `loadConfig`, ktorú každý modul musí implementovať. Tá dostane ako parameter inštanciu triedy `Loader`, ktorej dostupnosť musí zabezpečiť centrálna autorita `MessagePool`.

4.3.2 Posielanie

Rozhranie implementuje niekoľko metód na zasielanie správ cez sockety. Základnou je metóda `sendMessage`, ktorá pošle správu najnižšej úrovne rozhrania.

Správa obsahuje položky:

to - názov modulu, ktorému je adresovaná.

from - názov modulu, ktorý správu odoslal.

event - obsahuje dáta, väčšinou štruktúru SCXML `event`. Možné je posielať čokoľvek, záleží ako túto položku spracujú konkrétne moduly.

Posielanie cez socket sa realizuje spojením serializácie štruktúry pomocou balíku `Data::Dumper` a odoslaním serializovanej štruktúry cez vytvorený socket.

Metódou vyššej úrovne je `sendEvent`, ktorá posiela SCXML `event`. Táto metóda predpripraví štruktúru a odošle ju pomocou metódy `sendMessage` ako parameter `event`.

4.3.3 Prijímanie

Pre čítanie správ zo socketu sa využívajú blokovacie funkcie, ktoré zablokujú beh programu, pokiaľ nedorazia dáta, alebo nespadne spojenie v sockete. Mnou navrhnuté rozhranie tu poskytuje ekvitalenty k metódam, ktoré sa používajú na posielanie správ.

Metóda najnižšej úrovne je tu `receiveMessage`, ktorá prijme správu zo socketu v tvare, v akom sa odosiela a štruktúru rozdelí na časť `from` a časť `event`. Obe časti vráti vo svojom výsledku a časť `to` zahodí, pretože v tejto fáze spracovania správy už nie je potrebné uchovávať informáciu o cieľovom module. Zo socketu sa správa dostane pomocou volania blokujúcej funkcie načítania a následne rozparsovaním načítanej štruktúry perlovým príkazom `eval`. Keďže čítanie zo socketu je blokujúcou operáciou, teda aj volanie metódy `receiveMessage` je blokujúce, čo treba mať na zreteli pri navrhovaní nových modulov.

Na vyššej úrovni je metóda `receiveEvent`, ktorá prijíma SCXML `event` štruktúru. Na jej získanie používa metódu `receiveMessage`, teda jej volanie je tiež blokujúce.

4.4 Interpretácia SCXML, trieda Automat

Ako som už v kapitole o návrhu spomínal, modul Automat predstavuje interpreter SCXML, ktorý aplikáciu načíta zo súboru, vytvorí si potrebné komponenty a nasledne ju interpretuje. Tento modul v implementácii reprezentuje trieda `Automat`. Ako ostatné moduly je odvodená od triedy `Modul` a v návrhu je tiež postavená na úroveň ostatných modulov. Popis procesu interpretácia už bol opísaný v Kapitole 3. Analýza návrhu. V tejto sekcii by som chcel priblížiť štruktúry, ktoré sú v interpretácii využívané.

4.4.1 Reprezentácia výslednej štruktúry pre XMLParser

Pre potreby interpretácie SCXML som využil zjednodušenú štruktúru XML súboru uloženú v premennej perlového typu `hash`. Keďže XML má charakter stromu, dá sa pohodolne rozdeliť na uzly/nody, ktoré majú svojich potomkov, tiež nody.

Nody je možno rozdeliť na textové a ostatné. Textové sú tie, ktoré neobsahujú XML elementy. Výsledná štruktúra pre XMLParser je zjednodušená reprezentácia XML nody, v podobe perlového typu `hash` s nasledujúcimi pevne definovanými položkami:

param je typu `hash reference`. Zoznam atribútov elementu, štýlom názov-hodnota

name je typu `scalar`. Názov elementu.

child je typu `array reference`. Zoznam referencií na nody potomkov.

text je typu `scalar`. obsah priamych textových potomkov nody.

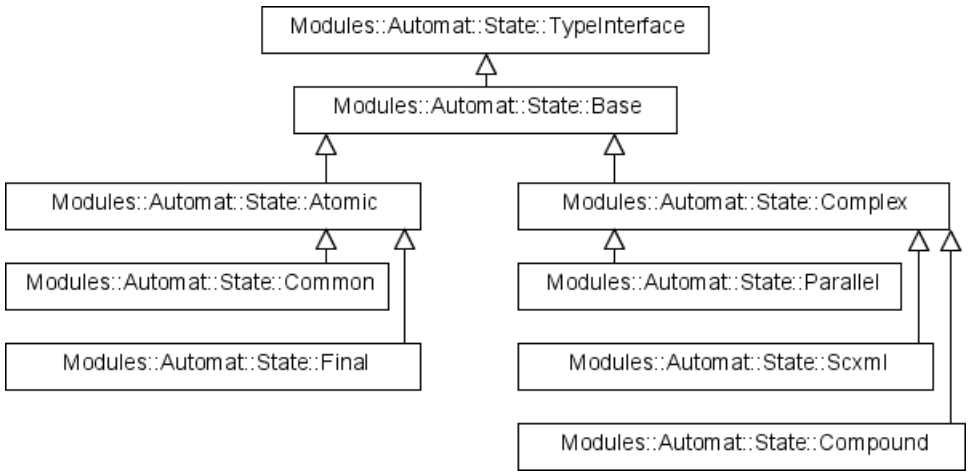
4.4.2 Reprezentácia fronty na interné udalosti

Fronta na interné udalosti je reprezentovaná pomerne priamočiaro, pomocou perlového typu `array`, ktorý obsahuje referencie na jednotlivé udalosti.

4.4.3 Reprezentácia fronty na externé udalosti

Fronta na externé udalosti v systéme nie je priamo reprezentovaná. Využíva sa buffer socketovej štruktúry konkrétneho operačného systému. Táto reprezentácia má isté obmedzenia, napríklad nemožnosť z nej odmazávať. V budúcich verziách bude táto fronta prepísaná a stane sa súčasťou novej vrstvy v rámci komunikácie, podrobnejšie rozpísanej v pasáži o elementoch `invoke` a `cancel` v neimplementovaných častiach.

4.4.4 Reprezentácia stavov



Obr. 4.1: Hierarchická reprezentácia objektového modelu stavov.

Na obrázku 4.1 je hierarchia tried stavov. Pričom posledná vrstva sú triedy stavov, ktoré reprezentujú konkrétne SCXML elementy rovnakého názvu.

Final reprezentuje `Modules::Automat::State::Final`.

Compound reprezentuje `Modules::Automat::State::Compound`.

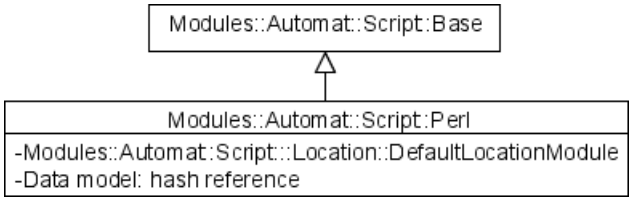
Parallel reprezentuje `Modules::Automat::State::Parallel`.

Scxml reprezentuje `Modules::Automat::State::Scxml`.

Common reprezentuje `Modules::Automat::State::Common`. Je to typ stavu, ktorý je atomický a zároveň nefinálny.

Trieda `Modules::Automat::State::TypeInterface` implementuje metódy, ktoré rozhodujú o type stavu, ktorý daná trieda reprezentuje. Každá trieda predefinuje metódu, príslušnú typu stavu, ktorý zastupuje, z triedy `TypeInterface` tak aby vracala výsledok, ktorý sa vyhodnotí v podmienkach ako pravdivý a ostatné metódy tohto interface ponechá nezmenené.

4.5 Skriptovací modul a dátový model



Obr. 4.2: Hierarchická reprezentácia skriptovacieho modulu a dátového modelu.

Skriptovací modul je sadou tried implementujúcich konkrétne skriptovacie a reprezentácie dátového modelu. Na obrázku 4.2 je vidieť vzťahy jednotlivých tried.

Trieda `Modules::Automat::Script::Base` implementuje základné rozhranie, ktoré musí skriptovací modul spĺňať kvôli spolupráci s triedou Automat. Každý konkrétny skriptovací modul si potom môže zvoliť ako atribút vhodnú triedu na vyhodnocovanie *location expressions*, ktorá pracuje s jeho dátovým modelom. Rovnako vidieť, že reprezentácia dát časti *data model*, je atribútom konkrétneho skriptovacieho modulu a jej reprezentácia je na ňom závislá.

Kapitola 5

Rozdiely oproti špecifikácii SCXML

Štandart SCXML je veľmi obsiahly a keďže je vo verzii *Working Draft* podlieha pravidelným zmenám, vďaka čomu niektoré jeho časti nie sú dostatočne vysvetlené. Bolo pre mňa veľmi ťažké implementovať všetky potrebné súčasti na funkčnú verziu a preto moja snaha smerovala ku tvorbe softvéru, ktorý sa bude dať pomerne jednoducho prispôsobovať prebiehajúcim zmenám. Niektoré súčasti štandardu som preto musel úplne vypustiť, iné sa podarilo nahradiť ekvivalentnou funkčnosťou. V budúcnosti predpokladám pokračovanie v tomto projekte a postupné doplňanie chýbajúcich častí, aby projekt splňoval plnú špecifikáciu SCXML. V tejto kapitole som časť materiálov prevzal z literatúry [1], hlavne v časti o *Neimplementovaných častiach*.

5.1 Neimplementované časti

5.1.1 Invoke, Cancel a Finalize

Element invoke slúži v špecifikácii na vyvolávanie externých entít mimo SCXML, ktorým je potom pomocou elementu `send` možné zasielať správy. Tieto časti môžu následne odpovedať správami, ktoré sa premieňajú na SCXML udalosti. V práci tento element nie je zahrnutý z dôvodu komplikovanej implementácie, ktorá by presahovala rozsah bakalárskej práce. Jeho funkčnosť je však vďaka návrhu zastúpená možnosťou zasielať správy ostatným modulom. Jedinou nevýhodou oproti postupu podľa špecifikácie je nemožnosť vyvolať novú SCXML session bez spustenia ďalšej inštancie celého systému.

Element cancel je priamo spätý s elementom `invoke`. Poskytuje možnosť zrušiť správu, ktorá ešte nebola spracovaná *externou entitou* vytvorenou pomocou elementu `invoke`. V súčasnej práci nemá žiadne zastúpenie, pretože správy sú okamžite doručované pomocou tcp/ip protokolu do fronty cieľového modulu, odkiaľ možnosť vymazávať neexistuje.

Element finalize slúži v SCXML k špecifikovaniu *executable contents*, ktoré manipulujú z dátami každej správy prichádzajúcej z *externej entity*vytvorenej pomocou elementu `invoke`. Vzhľadom na to, že element `invoke` nie je implementovaný, nevidím možnosť ako využiť služby elementu `finalize`, tak aby sa zachoval pôvodný zápis SCXML a preto ho nezahŕňam do súčasnej verzie.

Predpoklady do budúcnosti. Tieto problémy by sa dali vyriešiť pomocou vytvorenia vrstvy spojenia nad vrstvou obyčajného zasielania udalostí. Táto vrstva by zabezpečila prepojenie SCXML s konkrétnym volaným modulom a sprostredkovala by možnosť zasielať správy a možnosť na ne odpovedať bez nutnosti ručne pripisovať parameter `id`. Tieto správy by boli pred spracovaním ukladané do fronty, s ktorou by mohla vrstva pracovať. V prípade príchodu správy vyvolanej pomocou elementu `cancel` by zrušila žiadanú správu ešte pred spracovaním. Rovnako by táto vrstva zabezpečovala funkčnosť elementu `finalize` pri príchode správ do modulu Automat.

5.1.2 History

Element `history` v SCXML zabezpečuje funkciu ukladania časti konfigurácie. Princípom je zapamätanie si aktívnych potomkov rodičovského stavu elementu `history` pri odchode z tohto rodičovského stavu. V prípade príchodu do daného elementu history sa aktivuje okamžitý prechod do týchto potomkov. Jedná sa o pseudostav, ktorý obsahuje práve jeden prechod bez podmienky aktivovaný ihneď po príchode do tohto stavu. Cieľ prechodu je závislý na parametri `type` a type rodičovského stavu. V prípade, že sa jedná o typ `shallow`, vchádza sa len do najbližších potomkov. Ak sa jedná o typ `deep`, vchádza sa do najvzdialenejších potomkov. Element `history` môže byť potomkom stavu typu `compound`, kedy prechod obsahuje práve jeden cieľový stav, alebo stavu typu `parallel`, keď prechod obsahuje množinu stavov, práve jeden pre každú vetvu stavu `parallel`. Napriek tomu, že element `history` je pomerne zaujímavou funkčnosťou, jeho využitie v súčasnej implementácii nie je možné, pretože som ho implementovať nestihol. V budúcnosti ho však samozrejme plánujem pridať.

5.1.3 Validate

Element `validate` sa využíva na kontrolu *dátového modelu*, či niektorých špecifikovaných častí. Kontrola prebieha podľa schémy určenej parametrom `schema`. V súčasnej verzii schémy nie sú implementované, pretože by to zabralo pomerne veľa času a k funkčnej SCXML aplikácii nie sú nutné. Preto som ich odložil do neskorších verzií, keď bude zabezpečená celková funkcionalita podľa špecifikácie.

5.1.4 Anchor

Pre prípad, že potrebujeme uložiť časť výpočtu automatu, poskytuje nám špecifikácia SCXML element `anchor`. Jeho využitie spočíva v uložení celého *dátového modelu*, alebo jeho časti a stavu, ktorého je tento element potomkom. Princíp využitia je nasledovný. V prípade, že sa vojde do stavu, ktorý má za potomka element `anchor`, uloží sa pod hodnotou jeho parametru `type` špecifikovaná časť *dátového modelu* a `id` rodičovského stavu. Ak má nejaký práve aktivovaný prechod namiesto parametru `target`, parameter `anchor`, vyhľadá sa posledný navštívený `anchor` s rovnakou hodnotou v parametri `type`, vojde sa do jeho uloženého stavu a obnoví sa uložená časť *dátového modelu*. Ak takýto `anchor` ešte navštívený nebol, vojde sa znovu do jeho rodičovského stavu. Element v súčasnosti nie je implementovaný, pretože by to presahovalo rozsah bakalárskej práce. Ako jeho návrh by som si predstavoval globálny zoznam typov navštívených elementov `anchor` s uloženými časťami *dátového modelu* a `id` rodičovských stavov. Prechod by bol pravdepodobne implementovaný rovnakým spôsobom ako fungujú štandardné prechody v SCXML.

5.1.5 Exmode

Spracovanie parametru `exmode` v elemente `scxml` sa pri hodnote `strict` nespráva korektne podľa špecifikácie SCXML. Požadovaná je validácia počas behu aplikácie, zatiaľ čo súčasná implementácia validuje dokument pred začatím aplikácie. Pre tento problém som zatiaľ nenašiel riešenie, pretože XML dokumenty sa podľa XSD schém validujú ako celok a nie po častiach.

5.1.6 Profile

Súčasná implementácia neberie v úvahu parameter `profile` v elemente `scxml`. Správa sa ako by bol nastavený vždy na hodnotu `perl`.

V ďalších verziách plánujem vytvoriť špeciálne triedy profilov pre jednotlivé podporované moduly, ktoré budú inicializovať jednotlivé komponenty SCXML. Na začiatku sa podľa hodnoty položky `profile` vyberie žiadaná trieda a nové sa budú môcť pridávať bez významného zásahu do zdrojových kódov interpreteru.

5.2 Naviac implementované časti

5.2.1 Tts

Element tts slúži v práci, k pohodlnému využívaniu služieb modulu TTS priamo z kódu SCXML aplikácie.

SCXML element môže mať jeden z troch dostupných tvarov:

- `<tts method='say' engine='festival' />` V tomto tvare sa metóda a engine, ktorý ju implementuje špecifikujú pomocou atribútou. Položka `engine` je nepovinná, ak sa neuvedie, využije sa nastavenie z konfiguračného súboru.
- `<tts:say engine='festival' />` Ako vidieť metóda sa špecifikuje uvedením priamo v názve. Atribút `engine` funguje presne ako v predchádzajúcom prípade.
- `<tts:festival:say />` Je možné uviesť `engine` aj metódu priamo v názve elementu.

Ďalšie parametre sa uvádzajú priamo ako atribúty tohto elementu. Výnimku tvorí atribút `text`, ktorý je možno uviesť ako textového potomka. Napríklad takto:

```
<tts:say>inline text</tts:say>
```

Zároveň pre atribúty platí, že sa môžu uviesť v tvare `expr_nazovatributu='text'`. Teraz sa pomocou skriptovacieho modulu vyhodnotí položka `text` ako hodnotový výraz a hodnota sa odošle pod názvom `nazovatributu`. Modul Automat pri tomto elemente umožňuje veľmi flexibilný zápis, ktorý následne parsuje a odosiela ako správu určenú pre modul TTS. Správa v položke `data` obsahuje všetky atribúty elementu `tts`, okrem tých, ktoré mali prefix `expr_`. Tie sú uvedené bez prefixu ako som už spomínal vyššie.

Kapitola 6

Záver

V súčasnosti smer v IT sa vyznačuje pomerne veľkým dopytom po nových technológiach, ktoré by bolo možné využívať v komerčnej sfére malých a stredných spoločností. Tie väčšinou musia spĺňať nižšiu cenu, jednoduchú upravovateľnosť a rozširovateľnosť o nové funkcie, pretože tento druh spoločností pracuje s obmedzenými prostriedkami a zároveň svoje riešenia prispôsobuje na mieru rôznym projektom. V práci išlo o implementovanie interpreteru SCXML aplikácii, ktorý by zvládol pracovať s rozumne veľkou podmnožinou funkčností SCXML, tak aby bolo možné programovať SCXML aplikácie a zároveň spĺňal uvedené požiadavky.

Cieľ sa úspešne podaril. Pomocou výsledného interpreteru je možné spúšťať aplikácie SCXML, ktoré síce nepodporujú všetky jeho možnosti, ale za väčšinu z nich je možné nájsť postup poskytujúci vhodnú náhradu. Jeho jednoduchosť a rozšíriteľnosť spočíva v modulárnom návrhu, ktorý umožňuje pridávať nové funkcie prostredníctvom modulov bez zásahu do zdrojových kódov interpreteru a tak upraviť výsledok na mieru konkrétnemu projektu. Projekt je dostupný na internete spolu so zdrojovými kódmi, čo ešte viac prispieva k jeho vývoju a zmenám pri implementovaní ďalších funkcií a tiež prípadných nových verzií SCXML špecifikácie. Jeho prezentácia na internete zároveň poskytuje možnosť hlásenia chýb a požadovania nových vlastností. Následne sa môže sledovať a komentovať priebeh ich začleňovania do systému.

Sekciu o porovnaní implementácií som nezahrnul do textu z jednoduchého dôvodu. Všetky ostatné implementácie sú veľkými projektami rôznych spoločností. Buď sa predávajú ako hotový software, napríklad produkt od firmy Intervoice. Alebo sú to veľké projekty, na ktorých pracujú platení programátori, napríklad Commons SCXML, alebo Qt SCXML Engine. Ich cieľom je dodržiavať najnovšiu špecifikáciu, pričom sa zároveň prispôsobujú vlastným projektom spoločností, ktoré ich vyvíjajú. Ako vidieť, vzniká tu zásadný rozdiel medzi poslaním mojej práce a ostatných produktov, pretože cieľom práce bolo vytvoriť jednoducho modifikovateľný interpreter, tak aby si ho mohli jednotlivci, či menšie spoločnosti prispôsobovať na vlastné projekty. Vzhľadom na tento dôležitý rozdiel v myšlienke práci si myslím, že seriózne porovnanie nie je vhodné.

Ako som už vyššie v práci písal v budúcnosti predpokladám doplnenie všetkých elementov, ktoré SCXML špecifikácia ponúka. Avšak za najvýznamnejšie rozhodnutie, ktorým projekt v súčasnosti prechádza, považujem jeho zmenu z perlového programu na perlovú knižnicu aby ju bolo možné využívať aj z iných perlových skriptov. Po takejto zmene by bolo možné projekt uložiť na *CPAN*, čo je centrum všetkých perlových knižníc a skriptov, ktoré využívajú programátori v perle na celom svete a týmto spôsobom zväčšiť komunitu okolo projektu. Následne by komunita mohla prispievať k urýchleniu vývoja a držania kroku so zmenami v SCXML špecifikácii. V terajšej podobe je projekt síce zabalený do vlastného menného priestoru, takže vyzerá ako knižnica, ale zatiaľ neposkytuje žiadny interface na komunikáciu s okolným programom.

V mojom súčasnom zamestnaní sa plánuje využívať pri testovaní vyvíjaných aplikácii, kde by sa pomocou automatu jednoducho simulovali rôzne špecifické situácie. Vďaka prepojeniu s *UML 2.0* by takto mohli testovacie aplikácie písať priamo návrhári, čo by uľahčilo prácu programátorom a tí by sa takto mohli vo väčšine situácii sústrediť len na vývoj. Ako asi z mojich predchádzajúcich myšlienok vyplýva, prácu považujem za podarenú a pevne verím, že nájde uplatnenie v spomínaných komerčných, či nekomerčných sférach.

Literatúra

[1] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager: State Machine Notation for Control Abstraction, W3C Working Draft 7 May 2009: http://www.w3.org/TR/2009/WD-scxml-20090507.

[2] The Perl Programming Language: http://www.perl.org.

[3] CMU Sphinx: http://cmusphinx.sourceforge.net/wordpress.

[4] The Festival Speech Synthesis System: http://www.cstr.ed.ac.uk/projects/festival.

[5] Comprehensive Perl Archive Network: http://www.cpan.org.

[6] Wikipedia: http://www.wikipedia.org.