VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODELOVÁNÍ AGENTŮ PRO ROBOTICKÝ FOTBAL

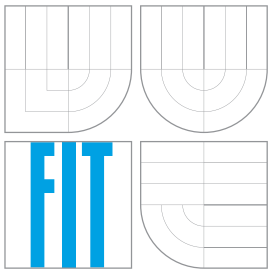
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV SUCHÝ

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODELOVÁNÍ AGENTŮ PRO ROBOTICKÝ FOTBAL

ROBOTIC SOCCER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV SUCHÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VLADIMÍ JANOUŠEK, Ph.D.

BRNO 2009

Vysoké učení technické v Brně – Fakulta informačních technologii

Ústav intelignetních systémů Akademický rok 2008/2009

Zadání diplomové práce

(kopie)

Řešitel: **Suchý Václav, Bc.**
Obor: Inteligentní systémy
Téma: **Modelování agentů pro robotický fotbal**
Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte problematiku robotického fotbalu.
2. Seznamte se s existujícími prostředky pro simulovaný robotický fotbal. Experimentujte s existujcícím softwarem.
3. Navrhněte vlastní variantu řízení robotů. Návrh proveďte s využitím modelů na bázi DEVS.
4. Navržené prostředky realizujte, ověřte funkčnost a vyhodnoťte dosažené výsledky.
5. Diskutujte možný navazující vývoj.

Vedoucí: **Janoušek Vladimír, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 22. září 2008
Datum odevzdání: 26. května 2009

Abstrakt

Tato práce popisuje návrh modelu agenta robotického fotbalu s využitím DEVS formalizmu. Za tímto účelem je představena vlastní modifikace klasického DEVS simulátoru v podobě paralelního realtime simulátoru. Ten umožňuje v reálném čase simulovat chování DEVS komponent založených na paralelním zpracování informací. Funkčnost modelu a simulátoru je předvedena na implementaci klienta robotického fotbalu pro simulační server iniciativy RoboCup. Klient se stal také základem pro připravovanou knihovnu k usnadnění tvorby vlastních agentů robotického fotbalu RoboCup.

Abstract

This work describes a design of an agent model for robotic soccer based on the DEVS formalism. There is also presented a design of own DEVS simulator (based on classic DEVS simulator) for parallel realtime simulations. Functionality of the simulator and the model is shown on an example of a soccer client for RoboCup Soccer Server. Based on this client, there is also presented a design of a library for easier creation of soccer clients for RoboCup.

Klíčová slova

Robotický fotbal, agent, robot, řízení, DEVS fomalismus, simulátor DEVS, Realtime DEVS simulátor, RoboCup Soccer.

Keywords

Robotics soccer, agent, robot, control, DEVS formalism, DEVS simulator, Realtime DEVS simulator, RoboCup Soccer.

Citace

Václav Suchý: Modelování agentů pro robotický fotbal, diplomová práce, Brno, FIT VUT v Brně, 2009

Modelování agentů pro robotický fotbal

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Vladimíra Janouška.

.....
Václav Suchý
26. května 2009

Poděkování

Chtěl bych poděkovat všem, kteří mě podpořili a pomohli mi při tvorbě této práce. Obzvláště pak mému vedoucímu, panu Janouškovi, za jeho ochotu poradit pokaždé, kdykoli nastaly potíže.

Obsah

Kapitola 1

Úvod

Tato práce navazuje na stejnojmennou semestrální práci Modelování agentů pro robotický fotbal [7]. Četba předchozí práce není nutná, protože veškeré důležité poznatky se vyskytují i v této práci.

Cílem práce je představit možnost použití DEVS formalismu pro popis modelu chování fotbalového robota. Dále ukázat, že lze využít dílčích výsledků jednotlivých kroků následné simulace modelu pro řízení robota účastnícího se simulovaného fotbalového utkání.

Robotický fotbal se stal v posledních letech zajímavou vědeckou základnou pro řešení problémů z nejrůznějších oblastí, jako jsou robotika, umělá inteligence, autonomní agenti, zpracování obrazu, řízení a regulace a další.

Robotický fotbal je hra podobná klasickému fotbalu. Zpravidla proti sobě stojí dva týmy autonomních robotů, tedy roboti kompletně řízeni počítačem, jejichž cílem je dát soupeři co největší počet gólů, vyhrát zápas. Hra se řídí podobnými pravidly, jako klasický fotbal. Na utkání dohlíží lidští rozhodčí, řeší fauly a sporné situace.

Ve světě existuje několik spolků pro robotický fotbal, které stanovují pravidla pro různé třídy robotů a pořádají turnaje. Mezi nejznámější patří iniciativa *RoboCup* (Robot World Cup Initiative) [13] a federace mezinárodních spolků robotického fotbalu *FIRA* (Federation of Internationl Robotsoceer Association) [9]. Obě organizace pořádají turnaje různých kategorií, avšak já se více zmíním jen o jedné organizaci, o RoboCup, protože jedné z jejich kategorii se týká tato práce.

Kapitola 2

RoboCupSoccer

RoboCup je pravděpodobně největší sdružení zabívající se robotickým fotbalem, na kterém se podílí přes 300 univerzit a výzkumných ústavů na celém světě. Kromě turnajů také každoročně pořádá konference a výukové semináře. Poslední sympózium RoboCup 2008 se konalo 14. až 20. června 2007 v Suzhou v Číně [10]. Robocup 2009 [11] bude probíhat od 29. června až do 6. července v Rakouském městě Graz. Cílem iniciativy RoboCup je do roku 2050 vyvinout humanoidní roboty, kteří by ve fotbale porazili skutečné mistry světa.

První zmínka o robotech hrajících fotbal se objevila v roce 1992 v článku Profesora Alana Mackworthta (Univeristy of British Columbia, Canada) „On Seeing Robots“. Nezávisle na něm, ve stejném roce, skupina japonských výzkumníků z oblasti umělé inteligence navrhla použití robotického fotbalu k podpoře rozvoje vědy a technologie. Po prozkoumání technologické a finanční stránky věci byla v červnu 1993 spuštěna soutěž *Robot J–League* (podle názvu nově vzniklé japonské profesionální fotbalové ligy), která byla později na žádost účastnících se mezinárodních skupin přejmenována na *Robot World Cup Initiative*, zkráceně RoboCup. První oficiální mezinárodní turnaj RoboCup proběhl v roce 1997 a setkal se s obrovským úspěchem. Od té doby každoročně probíhají turnaje po celém světě [15].

V dnešní době má RoboCup několik větví. Hlavní, se zaměřením na robotický fotbal, je označována jako *RoboCupSoccer*. Další větve jsou projekty zaměřené na záchranné sbory – *RoboCupRescue*, na praktické využití v reálném světě – *RoboCup@Home* a na vzdělávání – *RoboCupJunior*. Každá větev má několik kategorii [13], pro tuto práci jsou nejdůležitější kategorie z RoboCupSoccer: *Small–size, Middle–size, Standart platform* a *Simulation robot league*.

2.1 Small-size robot league

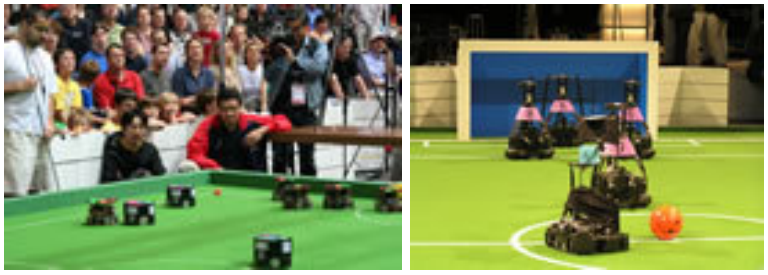
Fotbalové utkání mezi dvěma pětičlennými týmy robotů, jejichž velikost nesmí být větší jak 18 cm v průměru. Hraje se s oranžovým golfovým míčkem na hřišti o velikosti 6,5 x 4,5 metrů, po dobu 2krát 10 minut.

Tato liga se zaměřuje na multi-agentní systémy s centrálním řízením.

2.2 Middle-size robot league

Tito roboti mají velikost až 50 cm, hrají v šestičlenných týmech na hřišti o velikosti 12 x 18 metrů. Poločas trvá 15 minut. Veškeré senzory jsou součástí robota. Robot může komuni-

kovat s okolím pomocí bezdrátové sítě.



Obrázek 2.1: Small-size (vlevo), Middle-size (vpravo) robot league [13]

2.3 Standart platform robot league

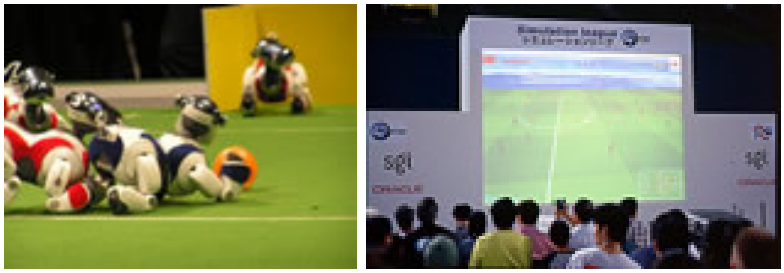
Tato liga nahrazuje velmi úspěšnou tzv. Čtyřnohou ligu (Four–Legged League). Všechny týmy hrají s identickými (standardními) roboty. Proto se mohou výhradně zaměřit na vývoj řídícího softwaru. Roboti jednají zcela autonomně, tedy není zde žádné řízení centrálním počítačem.

Do letošního roku se hrálo s roboty AIBO od společnosti SONY [14].

2.4 Simulation league

Je to jedna z prvních lig RoboCup Soccer. Fotbalové utkání se koná pouze virtuálně v simulátoru. Hráči jsou ovládání softwarovými programy (agenty), a celý zápas je zobrazován ve virtuálním 2D nebo 3D prostředí. Tato liga je velmi oblíbená a je zastoupena největším počtem týmů, protože není tak finančně a hardwarově nákladná jako ostatní ligy.

K usnadnění vývoje a testování řídícího softwaru iniciativa RoboCup volně poskytuje svůj oficiální simulátor. A právě ten jsem použil jako referenční simulátor pro testování vlastního modelu robota.



Obrázek 2.2: Standard platform (vlevo), Simulation (vpravo) league [13]

Kapitola 3

Robot a Agent

Cílem práce je vytvořit agenta pro robotický fotbal. Co je to robot a co je agent, co mají společné a v čem se liší, na tyto otázky by měla odpovědět tato kapitola.

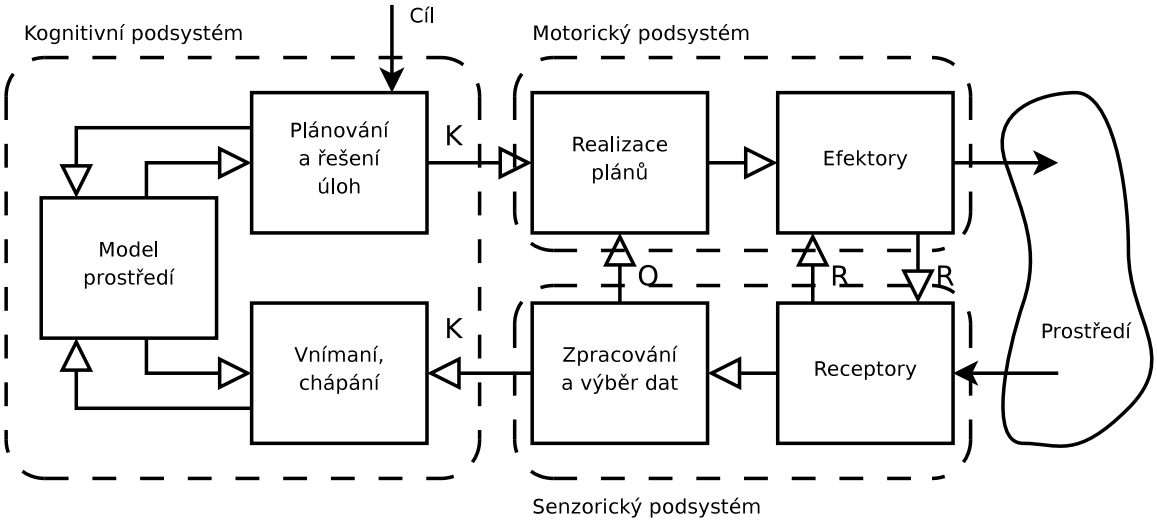
3.1 Robot

Slovem robot dnes obecně označujeme samostatně pracující stroj, který pro nás vykonává nějakou činnost. Toto slovo bylo známo už v 17. století ve významu otrocké práce. Ve spojení se strojem toto slovo poprvé použil český spisovatel Karel Čapek ve své divadelní hře R.U.R. Pro specifické typy robotů dnes používáme označení jako manipulátory (stroje dálkově ovládané), kuchyňské roboty (různé kombinované mixéry), zooidi (roboti se stavbou těla podobnou zvířatům) nebo třeba androidi (roboti se stavbou těla připomínajícího člověka). Jak již bylo zmíněno, obvykle se slovo robot používá ve spojení s nějakým strojem, hardwarem. V oblasti softwaru se s tímto pojmem moc nesetkáváme. Název robotický fotbal vychází hlavně z faktu, že se všech kategorií, krom fotbalových simulací, účastní reálné stroje - roboti. Proč se tedy zmiňuji o robotech, když cílem je vytvořit software pro simulované utkání? Důvodem je nechat se inspirovat složením a funkčností jednotlivých komponent reálného robota a na těchto základech vybudovat vlastní model s podobným rozložením logických a funkčních prvků tak, aby bylo možné tento model třeba použít na řízení reálného robota.

Obecně se robot skládá z několika abstraktních podsystémů, jak ukazuje obrázek 3.1. Jsou to *senzorický*, *motorický* a *kognitivní* podsystém. Tyto podsystémy mezi sebou komunikují pomocí tzv. *smyček*. Složení a nastavení podsystémů tak vlastně definuje celkové chování robota.

3.1.1 Senzorický podsystém

Senzorický podsystém má většinou dvě hlavní částí, jedna část je skupina *receptorů* a druhá *systém pro zpracování a výběr dat*. Receptory snímají fyzikální signály z okolí a převádí je na vhodné vnitřní signály [6]. Příkladem receptorů mohou být teploměry, dálkoměry, sonary, kamery apod. Všechny signály z receptorů prochází přes systém pro zpracování dat. Zde se signály třídí a filtrují, vybírají se pouze ty informace, které jsou důležité pro daného robota.



Obrázek 3.1: Podsystémy obecného robota

3.1.2 Motorický podsystém

Motorický podsystém se také skládá ze dvou částí, ze skupiny *efektorů* a z *realizátoru plánů*. Efektory jsou opakem receptorů. Receptory získávají informace z okolí, naopak efektory do okolí informace předávají, ovlivňují ho, umožňují robotovi se v okolí pohybovat. Efektory tak převádí vnitřní informaci na fyzikální jev. Nejčastěji jde o nějaký mechanický pohyb, buď celého robota, nebo jeho části, například ramene. Ale může jít třeba o vyslání informace pro komunikaci, například pomocí radiového signálu. Jednotlivé efektory jsou řízeny realizátorem plánů.

3.1.3 Kognitivní podsystém

Kognitivní podsystém představuje nadřazené inteligentní řízení. Tento podsystém provádí hlubší analýzu informací přicházejících ze senzorického podsystému. Pro kvalitní analýzu je vhodné, aby měl robot k dispozici nějaký model prostředí a stanoven cíl své práce. Na základě analýzy modelu prostředí a cíle práce se zde vytváří plán akcí, které nakonec robot provede [6].

3.1.4 Smyčky

Hlavní komunikační smyčkou je smyčka *kognitivní* (na obrázku 3.1 písmeno *K*). Jedná se o smyčku nejvyšší úrovně. Šíří se přes ni hlavní signály. Začíná v receptorech, prochází všemi podsystémy a končí v efektorech.

Mezi senzorickým a motorickým systémem existují zpětnovazební smyčky nižší úrovně. *Operační* smyčka (na obrázku 3.1 písmeno *O*) zajišťuje vykonávání naplánované úlohy. *Reflexní* smyčky (na obrázku 3.1 písmeno *R*) představují rychlé reakce robota v podobě reflexů, jako je například zastavení pohybu při kolizi s překážkou, vyhýbání se překážek a podobně.

3.2 Agent

Pojem agent vychází z latinského slova *agentum*, které znamená „ten, kdo jedná“. V našem reálném světě je agentem chápána osoba, která jedná v zájmu jiné osoby, svého klienta. Podobné chování očekáváme od umělého agenta. Obecná definice agenta může znít takto:

Umělý agent je člověkem vytvořené dílo, které v prostředí, do kterého je umístěno, jedná samostatně ve prospěch svého klienta [17].

Agent sám o sobě je systém, a je zároveň součástí systému. Podmnožina systému, která neobsahuje daného agenta, je označována jako okolí agenta, přesněji *prostředí*. Říkáme, že agent je v tomto prostředí *situován*, koná v daném prostředí. Agent přijímá podněty z prostředí pomocí senzorů (receptorů), zpracovává je a zpětně prostředí ovlivňuje svými efektory.

Na základě definice by se vlastními slovy dalo říct, že agent a robot mají hodně společného. Samostatně vykonávají nějakou činnost v zájmu jiných, umí reagovat na změny v okolí a toto okolí také mohou svým jednáním ovlivňovat. V simulovaném robotickém fotbalu se pro klienta - hráče více hodí označení agent, protože se jedná o sotwarový program, ne o reálný stroj. Avšak chování a struktura modelu může být velmi podobná robotovi, a proto ať už v další části textu použiji slovní spojení model agenta nebo model robota, budu vždy myslet model softwarového klienta (hráče) simulovaného robotického fotbalu.

Kapitola 4

DEVS formalismus

Tato kapitola přibližuje problematiku DEVS formalismu, na jehož základě je postaven model vlastního robota pro robotický fotbal.

Discrete event system specification (specifikace systémů s diskrétními událostmi) je modulární a hierarchický formalismus pro popis diskrétních systémů. Tento nástroj umožňuje vytvářet matematicky přesné modely systémů a následně simulovat jejich chování. Stal se oblíbeným v oblasti modelování a simulace především pro jeho hierarchické a modulární vlastnosti, díky nimž lze využívat znovupoužitelnosti vytvořených komponent bez nutnosti opakovat jejich verifikaci. Také se vyznačuje prvky objektového orientovaného přístupu, kdy jednotlivé komponenty představují objekty, které s okolím komunikují pomocí vstupních a výstupních událostí, jsou charakterizovány svým vnitřním stavem, který se může v čase vyvíjet. Skládáním (propojováním) těchto komponent lze modelovat nejrůznější systémy od simulace fronty u přepážky [4], přes simulaci skákajícího míčku po podložce až po modelování, simulaci a testování zbraňových systémů [2] a plánování strategií bitev. Další velkou výhodou DEVS formalismu je relativně snadná převoditelnost jiných známých metod modelování (konečné automaty, petriho sítě) na tento DEVS popis.

Klasický DEVS, navržený Dr. Bernardem P. Zeiglerem (profesor University of Arizona), byl původně vytvořen pro systémy s diskrétními událostmi. Zeigler ho se svým týmem představil v roce 1976 ve své knize *Theory of Modeling and Simulation*. Od té doby byl původní DEVS rozšířen o další vlastnosti a vznikly tak modifikace pro popis systému s diskrétním časem (DTSS), pro popis spojitých systémů (DESS), pro systémy tvořené celulárními sítěmi (CELL-DEVS), DEVS pro real-time simulace (RT-DEVS) a další [18].

4.1 Základní DEVS

DEVS formalismus matematicky popisuje chování modelu systému pomocí hierarchicky propojených DEVS jednotek - *Atomických* a *složených* DEVSů. Jedna atomická jednotka představuje dílčí část problému na nejnižším stupni abstrakce. Tyto jednotky se pak mohou skládat (propojovat) do větších celků - složených DEVSů, jejichž chování je pro okolí stejné jako chování atomického DEVSu, ale reprezentují model problému na vyšším stupni abstrakce.

4.1.1 Atomický DEVS

Atomický DEVS je základní jednotka modelu. S vnějším prostředím komunikuje pomocí vstupů a výstupů, označovaných jako *porty*. Uvnitř DEVSu se nachází stavové řízení, jehož typ a složitost není předem definována. Může tak obsahovat formalizmy konečných automatů, Petriho sítě, nebo například algoritmy spadající do třídy Turingových strojů. Tato vlastnost poskytuje velkou flexibilitu při tvorbě modelů, umožňuje např. návrh heterogenních modelů.

Model atomického DEVSu je definován jako sedmice

$$M = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta) \tag{4.1}$$

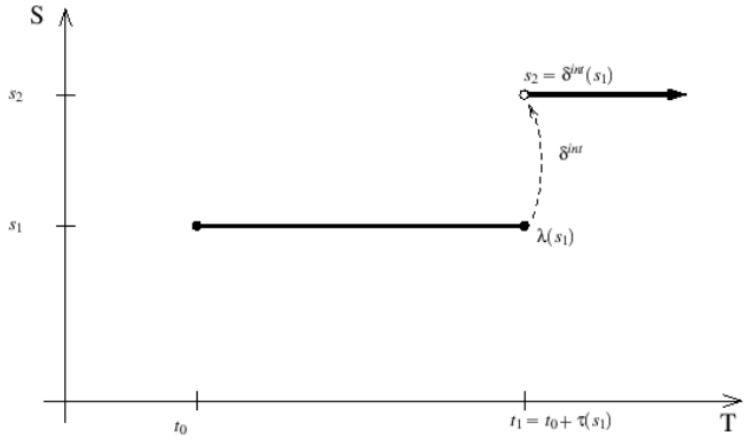
kde

- $X = \{(x_1, x_2, \ldots, x_m) | x_1 \in X_1, x_2 \in X_2, \ldots, x_m \in X_m\}$ je strukturovaná množina vstupů,
- $Y = \{(y_1, y_2, \ldots, y_p) | y_1 \in Y_1, y_2 \in Y_2, \ldots, y_p \in Y_p\}$ je strukturovaná množina výstupů,
- S je množina vnitřních stavů,
- $\delta_{ext} : Q \times X \rightarrow S$ je externí přechodová funkce. Q je množina totálních stavů $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$,
- $\delta_{int} : S \rightarrow S$ je interní přechodová funkce,
- $\lambda : S \rightarrow Y$ je výstupní funkce,
- $ta : S \rightarrow \Re_0^+ \bigcup \{\infty\}$ je funkce posunu času, která udává zbývající čas do výskytu následující vnitřní události.

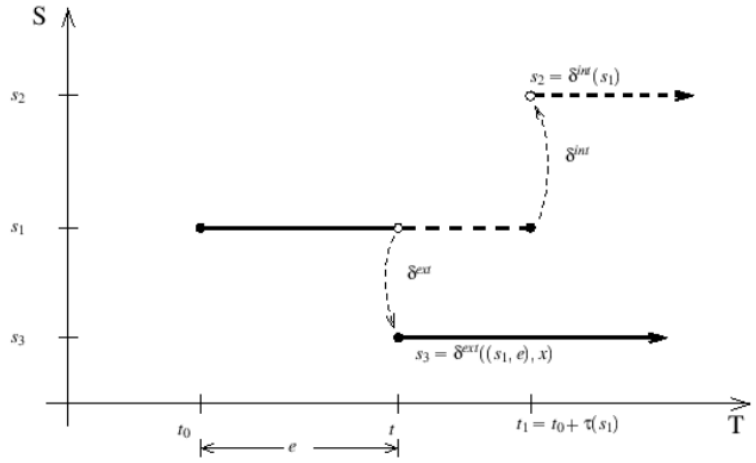
Vnitřní stav $s \in S$ modelu je spjat s časem t. Vztah mezi stavem a časem vyjadřuje funkce času $ta(s)$, která udává dobu trvání aktuálního stavu s. Pokud dojde k vypršení času aktuálního stavu, provede se výstupní funkce $\lambda(s)$, která na základě aktuálního stavu vyvolá výstupní událost a ta se projeví jako vstupní u všech dalších na tento výstup připojených DEVSů. Po výstupní funkci dochází ke změně vnitřního stavu podle interní přechodové funkce $s_n = \delta_{int}(s)$ a následného určení doby trvání nově vzniklého stavu podle funkce $t(s_n)$. Na externí událost komponenta reaguje změnou vnitřního stavu na základě externí přechodové funkce $s_n = \delta_{ext}((s, e), x)$, kde e představuje uplynulý čas od poslední změny vnitřního stavu. Interní přechod (obrázek 4.1) se provede po uplynutí doby $ta(s_1)$. Bezprostředně před tím, než dojde ke změně stavu $s_2 = \delta_{int}(s_1)$, se vygeneruje výstup $\lambda(s_1)$. Po změně stavu se naplánuje další interní přechod na dobu $t + ta(s_2)$. Externí přechod (obrázek 4.2) okamžitě reaguje na vstupní událost x změnou stavu $s_2 = \delta_{ext}((s_1, e), x)$. Negeneruje se žádný výstup, pouze se naplánuje interní přechod na $t + ta(s_2)$.

4.1.2 Složený DEVS

Složený DEVS obsahuje síť vnořených atomických nebo složených DEVSů. Vnořené komponenty jsou propojeny pomocí vstupních a výstupních portů. Tento mechanismus umožňuje vytvářet hierarchické modely (obrázek 4.3). Složené DEVSy se starají o správné rozesílání



Obrázek 4.1: Interní přechod v DEVS [18]



Obrázek 4.2: Externí přechod v DEVS [18]

datových a řídících zpráv mezi svými komponentami. Z hlediska topologie modelu dělíme zprávy na tři typy. *Vstupní* zprávy přichází z vnějšku do složeného DEVSu, ty jsou přeposlány na vstupní porty vnořených komponent. Vnořené komponenty si mezi sebou posílají *vnitřní* zprávy. *Výstupní* zprávy jsou produkovány vnořenými komponentami a jsou vedeny na výstupy složeného DEVSu.

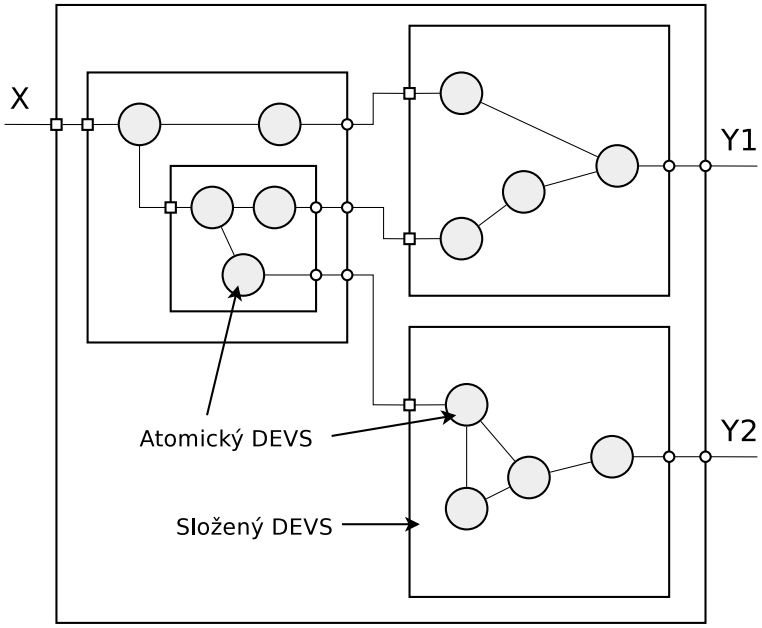
Množina DEVS systémů je uzavřená vzhledem k operaci skládání, tj. každý složený systém je současně DEVS. Tato vlastnost je podstatná pro vytváření hierarchických systémů a byla formálně dokázána [18].

Složený devs je osmice

$$N = (X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select) \tag{4.2}$$

kde

- X je množina vstupních událostí,
- Y je množina výstupních událostí,
- D je množina jmen, odkazů na DEVS komponenty,
- $\{M_i\}$ je množina DEVS komponentů, kde pro všechny $i \in D$ je M_i atomický nebo složený DEVS,
- $C_{xx} \subseteq \{X \times \bigcup_{i \in D} X_i\}$ je množina propojení mezi vnějšími a vnitřními vstupy,
- $C_{yx} \subseteq \{\bigcup_{i \in D} Y_i \times \bigcup_{i \in D} X_i\}$ je množina propojení mezi vnitřními vstupy a vstupy,
- $C_{yy} \subseteq \{\bigcup_{i \in D} Y_i \times Y\}$ je množina propojení mezi vnitřními a vnějšími výstupy.
- $Select : 2^D \rightarrow D$ je funkce (tie–breaking function) pro rozhodování pořadí událostí, které se vyskytly současně.



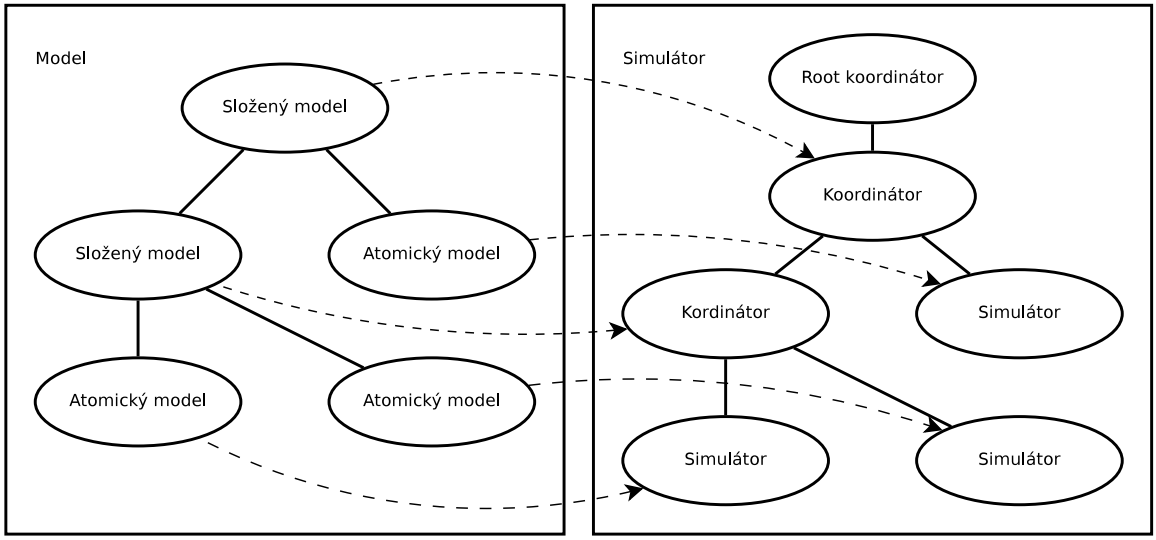
Obrázek 4.3: Složený DEVS model

Chování složeného DEVSu je podobné jako u atomického. Složený DEVS N změní stav některé své komponenty, pokud do N přijde vnější událost $x \in X$, nebo když proběhne naplánovaná vnitřní událost, která opět generuje výstup $y_i \in Y_i$. V obou případech jsou spuštěné události vyslány všem závislým komponentám, kde závislosti jsou definovány pomocí množin propojení C_{xx}, C_{yx} a C_{yy}.

4.1.3 DEVS simulátor

Výše popsané chování DEVS komponent je řízeno *simulátory* a *koordinátory*. Každý atomický DEVS model je součástí simulátoru. Tato jednotka komunikuje s ostatními simulátory pomocí řídících zpráv a stará se o správný běh DEVS komponenty v čase tak, že volá její

příslušné funkce. Každý složený DEVS je součástí koordinátoru. Ten se stará o synchronizaci času a rozesílání zpráv mezi vnitřními komponentami. Nejvyšší koordinátor v hierarchii modelu se nazývá kořenový - *root* koordinátor. Představuje hlavní smyčku celé simulace. Na začátku inicializuje celý model do požadovaného počátečního času a dál ve smyčce provádí jednotlivé kroky simulace tak dlouho, dokud není splněna podmínka ukončení simulace. Hierarchie simulátorů a koordinátorů pro jednoduchý DEVS model je pro ilustraci znázorněna na obrázku 4.4.



Obrázek 4.4: Simulátor DEVS modelu

Simulátor tedy vedle Atomického DEVS modelu ($M = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$) dále obsahuje odkaz na nadřazený koordinátor *parrent*, čas poslední vnitřní události tl, čas další naplánované události tn a poslední výstupní hodnotu DEVS modelu y.

Koordinátor se skládá ze složeného modelu ($N = (X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select)$), odkazu na nadřazený koordinátor *parrent*, času poslední tl a příští tn události, kalendáře událostí a odkazu na aktuální vybraný prvek $d*$. Kalendář událostí je struktura, která uchovává odkazy na podřízené DEVSy a umožňuje rychle uspořádat jejich pořadí podle času následné událost tn.

Simulátory a koordinátory mezi sebou komunikují pomocí 4 základních zpráv. *Inicializační* zpráva (i, t) je rozeslána postupně od root koordinátoru všem koordinátorům a simulátorům. Tak se model uvede do požadovaného počátečního stavu. *Interní* zprávu $(*, t)$ posílá nadřazený koordinátor svým podřízeným a informuje je tak o provedení plánované vnitřní události. Dále nadřazený koordinátor informuje své podřízené o externí události pomocí *vstupní* zprávy (x, t). O tom, že došlo k výstupní události uvnitř DEVS modelu, informuje simulátor/koordinátor svůj nadřazený koordinátor pomocí *výstupní* zprávy (y, t).

Inicializace simulátoru je prostá (kód 4.1). DEVS komponenta si jen uloží inicializační čas jako čas poslední události a spočítá čas události následné.

```
Simulator::message(i,t)
{
    tl=t;          // poslední čas je inicializační čas
```

```
    tn=ta(s);    // příští řas je funkce aktuálního stavu
}
```

Listing 4.1: Reakce simulátoru na inicializační zprávu (*i*,*t*)

Při inicializaci koordinátoru (kód 4.2) se inicializační zpráva přepošle všem podřízeným komponentám. Poté dojde k seřazení kalendáře (event_list) tak, aby na prvním místě v kalendáři byla komponenta s nejbližší následnou událostí. Čas poslední události koordinátora je definován jako maximální hodnota z časů posledních událostí všech podřízených komponent a čas následné události jako minimum z časů následných událostí podřízených komponent.

```
Coordinator::message(i,t)
{
    foreach (d* from D)
    {
        d*->message(i,t);          // pošle init všem podřízeným
    }
    Event_list.sort_by_tn();       // podle tn každého d z~D
    tl=D.max(TL);                  // max hodnota tl ze všech D
    tn=D.min(TN);                  // min hodnota tn ze všech D
}
```

Listing 4.2: Reakce koordinátoru na inicializační zprávu (*i*,*t*)

Když simulátoru přijde interní zprava $(*, t)$, nejprve DEVS model vypočítá výstupní hodnotu $y = \lambda(s)$ a tu odešle nadřazenému koordinátorovi. Dále určí svůj nový stav s, čas poslední tl a následné tn události (kód 4.3).

```
Simulator::message(*,t)
{
    if (t!=tn) ERROR wrong sync.

    y=lambda(s);               // výstupní událost
    parent->message(y,t);      // odešli nadřazenému
    s=delta_int(s);            // novy stav

    tl=t;                      // čas poslední události
    tn=t+ta(s);                // čas následné události
}
```

Listing 4.3: Reakce simulátoru na interní zprávu (*,*t*)

Na vstupní zprávu simulátor reaguje spočtením uplynulého času e od poslední události. Následně z aktuálního stavu s, uplynulého času e a příchozí vstupní události x určí svůj nový stav. Nakonec opět spočítá časy poslední a následné události (kód 4.4).

```
Simulator::message(x,t)
```

```
{
    if(t<tl || t>tn) ERROR wrong sync.  //test synchronizace

    e=t-tl;                    // uplynulý čas
    s=delta_ext(s,e,x);        // výpočet nového stavu
    y=lambda(s);               // výstupní událost

    tl=t;                      // čas poslední události
    tn=t+ta(s);                // čas následné události
}
```

Listing 4.4: Reakce simulátoru na vstupní zprávu (x,t)

Po přijetí interní zprávy $(*,t)$ koordinátorem dojde k přeposlání této zprávy první podřízené komponentě z kalendáře událostí. Následně dojde k opětovnému seřazení kalendáře a k určení časů poslední a následné události (kód 4.5).

```
Coordinator::message(*,t)
{
    if (t!=tn) ERROR wrong sync.

    d*=Event_list.get_firs();    // vyber prvního
    d*->message(*,t);            // a~tomu přepošli zprávu

    Event_list.sort_by_tn();     // seřaď kalendář

    tl=t;                        // čas poslední události
    tn=D.min(TN);                // čas následné události
}
```

Listing 4.5: Reakce koordinátoru na interní zprávu $(*,t)$

Vstupní zprávu (x,t) koordinátor přeposílá všem připojeným komponentám. Toto propojení je dáno množinou C_{xx}. Poté opět dojde k seřazení kalendáře a aktualizaci časů (kód 4.6).

```
Coordinator::message(x,t)
{
    if(t<tl || t>tn) ERROR wrong sync.  //test synchronizace

    foreach (d* in D)
    {
        if(Cxx.contain(x,xd)     // pokud existuje propojeni
            d*->message(xd,t);   // přepošli komponentě zprávu
    }

    Event_list.sort_by_tn();
```

```
    tl=t;                                   // aktualizace času
    tn=D.min(TN);
}
```

Listing 4.6: Reakce koordinátoru na vstupní zprávu (*x,t*)

Posledním typem zprávy, kterou přijímá koordinátor, je zpráva výstupní (kód 4.7). Přijatá zpráva je přeposlána na vnější výstup složeného DEVSu (pokud existuje propojení mezi výstupem dané komponenty a vnějším výstupem složeného DEVSu v C_{yy}) a dále všem připojeným vnitřním komponentám na jejich vnitřní vstup podle C_{yx}.

```
Coordinator::message(yd,t)
{

    if(Cyy.contain(yd,y)      // pokud existuje propojení
            parrent->message(y,t);   // přepošli zprávu nadřazenému
    }

    foreach (d* in D)
    {
        if(Cyx.contain(yx,xd)    // pokud existuje vnitřní propojení
            d*->message(xd,t);   // přepošli komponentě zprávu
    }
}
```

Listing 4.7: Reakce koordinátoru na výstupní zprávu (*yd,t*)

Zbývá root koordinátor (kód 4.8). Jeho úkolem je inicializovat model do času t_{init} a pravidelně volat další naplánovanou událost a testovat, zda nebyla splněna podmínka pro ukončení simulace. Root koordinátor obsahuje aktuální čas simulace t a odkaz na první koordinátor *child*.

```
RootCoordinator::simulate(t_init)
{
    t=t_init;

    child->message(i,t);      // inicializuj model
    t=child->get_tn();        // další naplánovaná událost

    do                        // opakovaně volej vnitřní události
    {
        child->nessage(*,t);
        t=child->get_tn();

    } while (!end_of_simulation) //doku není konec simulace
}
```

Listing 4.8: Root koordinátor

4.2 Real-time DEVS (RT-DEVS)

V předchozí kapitole byl nastíněn obecný princip klasického DEVS simulátoru. Simulace byla založena na událostech, o kterých simulátor věděl, v který okamžik přesně nastanou. Navíc tento čas byl vnitřní, relativní, a jeho krok se měnil podle potřeby. Pro realtime systémy potřebujeme DEVS simulátor, který bude pracovat s reálným časem a události v něm budou vznikat nejen na základě naplánovaných vnitřních přechodů, ale i tzv. na „požádání“, kdy komponenta dává simulátoru znamení, že dokončila nějakou činnost.

Příkladem takového nástroje může být RT-DEVS [16] navržený tak, že rozšiřuje model atomického DEVSu o podmíněné aktivity, model složeného DEVSu zůstává stejný a simulátor se skládá především ze dvou částí, jedna řídí klasické plánované události, druhá reaguje na žádosti od komponent.

Atomický RTDEVS model, RTAM, je definován takto [16]:

$$RTAM = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta, A, \psi) \tag{4.3}$$

kde

- X je množina vstupů vnějších událostí,
- S je množina postupných stavů,
- Y je množina výstupů,
- $\delta_{int} : S \to S$ je interní přechodová funkce,
- $\delta_{ext} : Q \times X^b \to S$ je externí přechodová funkce. Q je množina totálních stavů $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ a X^b je množina vaků (bags) nad prvky množiny X,
- $\lambda : S \to Y^b$ je výstupní funkce, Y^b je množina vaků nad prvky množiny Y,
- ta je funkce posunu reálného času (v reálu),
- A je množina aktivit s podmínkami,
- $\psi : S \to A$ je funkce mapující stavy na aktivity.

RTDEVS simulátor pracuje s reálným časem, to znamená, že model musí být schopen měnit stav a produkovat výstupní události ve specifickém čase. RT simulátor je proto navržen tak, že je schopen obsloužit dva typy událostí, *periodickou* událost a *reaktivní*. DEVS formalismus už definuje dva typy událostí, vnitřní a vnější, na které reaguje vnitřní přechodová funkce (δ_{int}), respektive vnejší přechodová funkce (δ_{ext}). RTDEVS formalismus nám dovoluje na tyto vnější a vnitřní události mapovat reaktivní respektive periodické události.

Každá devs komponenta má vlastní simulátor. Ten je rozdělen na dvě základní části. První část - periodická (kód 4.9), se stará o plánování a provádění vnitřních událostí. Má roli časovače. Ve vlastním vlákně v nekonečné smyčce postupně plánuje a volá vnitřní události. Výstupní funkce může vyvolávat aktivity, které představují nějakou výpočetní činnost, po jejímž skončení může být vyvolána vnější událost. Na tyto vnější události reaguje druhá část simulátoru - reaktivní (kód 4.10). Ta má vlastnosti přerušení. Může v jakémkoli časovém okamžiku na základě vnější události změnit vnitřní stav komponenty. Root koordinátor se pouze stará o inicializaci simulace.

U tohoto typu simulace je velmi důležitá synchronizace reálného času a také rozesílání zpráv mezi DEVS komponentami. V tomto návrhu simulátoru se o šíření zpráv staralo rozhraní CORBA [16].

```
while(true)
{
    t=checkCurrentTime();     // aktuální čas
    wait(sigma);              // čekej do další naplánované události
    y=lambda(s);              // výstupní událost
    s=delta_int(s);           // změna vnitřního stavu
    t=child->get_tn();        // další naplánovaná událost
    tl=checkCurrentTime();
    tn=lt+ta(s);
    }
```

Listing 4.9: Periodická část RT simulátoru

```
external_event_triggered(x)
{
    t=checkCurrentTime();         // aktuální čas
    e=t-tl;
    s=delta_ext(s,e,x);           // změna vnitřního stavu
    tl=checkCurrentTime();
}
```

Listing 4.10: Reaktivní část RT simulátoru

Toto rozšíření DEVS formalismu je vhodné pro modelování chování fotbalového robota. Při hraní potřebujeme, aby simulace probíhala v reálném čase, aby byl robot schopen kdykoli reagovat na vstupní události. Zároveň však můžeme s výhodou využít modulární vlastnosti DEVS formalismu a jednotlivé logické části robota rozdělit do samostatných modulů a jejich chování řešit zvlášť. Propojením modulů pak vznikne výsledný model robota.

Kapitola 5

RoboCup Soccer simulátor

Model chování robota bude obecný, avšak jeho funkčnost je třeba otestovat v nějakém prostředí. Toto prostředí bude představovat již zmíněný simulátor robotického fotbalu od iniciativy RoboCup. Proto se v této kapitole seznámíme se základními vlastnostmi a požadavky simulátoru.

RoboCup Soccer simulator oficiální simulátor iniciativy RoboCup, který používá při utkáních v kategorii Simulation league. Podrobnější informace o serveru jsou k dispozici v manuálu dodávaným RoboCupem [5].

5.1 Základní vlastnosti

Soccer Server simulátor je systém, který umožňuje simulovat (hrát) fotbalový zápas mezi dvěma týmy autonomních robotů.

Systém se skládá z jednoho serveru a z několika klientů. Server představuje virtuální hřiště, simuluje veškerý pohyb hráčů a míče. Každý klient ovládá chování jednoho svého hráče. Komunikace mezi serverem a klienty je zajištěna pomocí UDP/IP soketů. Díky tomu mohou být klienti vytvořeni v libovolném programovacím jazyku podporující tuto komunikaci.

Server má dvě částí: *soccerserver* a *soccermonitor*. Soccerserver řídí vlastní simulaci zápasu, soccermonitor umožňuje informace ze soccerserveru zobrazovat v podobě 2D hřiště s hrajícími hráči.

5.2 Pravidla

Pravidla tohoto simulátoru jsou podobná pravidlům skutečného fotbalu.

Fotbalové utkání spolu hrají dva týmy po 11 hráčích, jeden brankář a 10 hráčů v poli. Brankař jako jediný má schopnost chytit míč a přemisťovat se s ním. Zápas se hraje na dva poločasy. Virtuální rozhodčí dohlíží na dodržování základních pravidel, jako jsou auty nebo ofsajdy. Při těchto situacích dochází k přerušení hry, přemístění hráčů a k opětovnému rozehrání.

5.3 Komunikační protokol

Server s klienty komunikuje přes síťové rozhraní pomocí UDP/IP soketů. Každý hráč komunikuje se serverem přes vlastní port, který mu je přidělen po prvním připojení k serveru.

První verze soccerserver byla napsána v jazyce Lisp, proto všechny přenášené zprávy mají podobu *S výrazů (S–expression)*, neboli symbolických výrazů. Server posílá několik typů informací. Po té, co se klient připojí, mu server oznámí základní nastavení serveru, port klienta a základní vnitřní informace hráče, jako je jeho vytrvalost, rychlost a podobně. Po připojení dostatečného počtu hráčů začíná vlastní utkání. Simulace probíhá v časových intervalech, v základním nastavení serveru 150 milisekund dlouhých. Během jednoho intervalu (kola) server posílá tři hlavní senzorické zprávy a přijímá jeden z 9 příkazů (motorické zprávy), kterými klient může ovlivnit své chování.

5.3.1 Senzorické zprávy

Server během hry pravidelně posílá tři typy zpráv:

- *Hear* – Zpráva představuje zvukový vjem klienta. Touto formou mohou přicházet informace od trenéra, rozhodčího nebo od ostatních spoluhráčů, kteří se nacházejí v nejbližším okolí. Hlavní atributy jedné zprávy jsou text (obsah zprávy), typ odesilatele a směr, odkud zpráva přišla.

  Zprávu typu *hear* server přeposílá všem příslušným klientům okamžitě.

- *See* – Tato zpráva obsahuje informace o objektech v klientově zorném poli. Můžeme tak zjistit detailní informace o okolních objektech, například typ objektu, jeho vzdálenost, směr, rychlost apod. Množství poskytnutých detailů o objektu se odvíjí také od vzdálenosti mezi hráčem a objektem.

  Zpráva typu *see* přichází klientům v pravidelných intervalech, typicky po 150 ms.

- *Sense_body* – V této zprávě jsou poskytnuty veškeré informace o fyzickém stavu hráče. Najdeme zde údaje o aktuální rychlosti, vytrvalosti, úhlu pootočení hlavy a další.

  Zprávu typu *sense_body* server hráčům posílá pravidelně po 100 ms.

5.3.2 Motorické zprávy

Klient může ovlivňovat sebe a své okolí pomocí následujících zpráv, které zasílá serveru:

- *Catch* – Povel pro pokus chytit míč. Tuto zprávu zasílá pouze brankář.

- *Change_view* – Změna pohledu. Ovlivňuje rychlost a množství přijímaných vizuálních podnětů.

- *Dash* – Povel pro zrychlení pohybu vpřed nebo vzad.

- *Move* – Na začátku hry nebo například při změně stran lze tímto příkazem umístit hráče přesně na určitou pozici. V ostatních případech lze hráčem pohybovat pouze relativně vzhledem k jeho minulé pozici.

- *Say* – Hráč může předávat svému okolí zprávy.

- *Sense_body* – Informace o stavu hráče přichází v pravidelných intervalech. Okamžité aktuální informace lze získat použitím tohoto příkazu.
- *Turn* – Otočit směr pohybu hráče o zvolený úhel.
- *Turn_neck* – Povel otáčí hlavu hráče o požadovaný úhel. Tento příkaz je vhodný pro rozhlížení.

Detailní popis jednotlivých zpráv, atributů, typů hodnot, můžeme najít v manuálu k RoboCup Soccer Server [5].

Kapitola 6

Návrh a implementace

V této kapitole bude podrobně rozebrán vlastní návrh modelu agenta pro robotický fotbal a jeho realtime simulátoru. Budou zde popsány jednotlivé komponenty modelu, jejich vlastnosti, typy zpráv předávané mezi komponentami. Dále zde uvedu, jaké nástroje (programovací jazyk, knihovny) byly při implementaci využity a za jakým účelem.

6.1 Nástroje

Robocup soccer simulátor si neklade žádné velké nároky na programovací jazyk, ve kterém by klient musel být vytvořen. Jedinou podmínkou je fakt, že veškerá komunikace se serverem probíhá pomocí síťového protokolu UDP/IP, a tak zvolený jazyk a systém musí poskytovat prostředky pro tuto komunikaci. Z tohoto pohledu nemá server na výběr programovacího jazyka žádný vliv.

Dále bylo vhodné použít nějaký nástroj umožňující modelovat a simulovat DEVS systémy. Bohužel, těchto prostředků volně dostupných mnoho není. Existují nástroje jen pro známější jazyky a to vždy převážně v jednom či dvou exemplářích. V úvahu přicházeli nástroje [8] pro jazyky C++ (ADEVS, DEVS/C++), Java (DEVSJAVA, SimBeans) a jazyk SmallTalk (nástroj SmallDevs). Z programovacích jazyků mi osobně nejvíce vyhovuje C++.

Při shromažďování informací k Robocup soccer simulátoru i robotickému fotbalu obecně jsem narazil na projekt Toma Howarda RCCParser [12]. Parser v podobě C++ knihovny vytvořen speciálně pro rychlé zpracovávání zpráv od RoboCup Soccer simulátoru. Možnost využití této knihovny také přispěla k finálnímu rozhodnutí implementovat vlastního klienta v jazyce C++.

Z dostupných modelovacích nástrojů jsem si vybral ADEVS, protože byl dodáván v podobě samostatných tříd jak pro DEVS komponenty, tak pro samotný simulátor. Nástroj DEVS/C++ je dodáván v podobě kompletního vývojového prostředí jako nástavby do vývojového prostředí Eclipse. Podle specifikace nabízí širší možnosti použití, obsahuje i variantu Real Time DEVS, avšak přestože model lze popsat jazykem C++, výsledná simulace pak probíhá v dodávaném vývojovém prostředí, a proto se tento nástroj nehodí pro samostatné aplikace, jako je například klient robotického fotbalu.

Před návrhem a implementací výsledného modelu byly brány v úvahu tyto prostředky. Aplikace bude implementována v jazyce C++ pod operačním systémem Linux. Ke komunikaci se serverem přes síťový protokol UDP/IP se použije knihovna asio z balíku knihoven boost, ze které lze taktéž využít užitečné nástroje pro práci s kontejnery či vlákny. DEVS model agenta bude modelován a dále jeho chování simulováno nástrojem ADEVS. Pro

usnadnění komunikace se serverem bude využita knihovna RCCParser pro zpracovávání S-výrazů.

6.1.1 ADEVS

ADEVS (A Discrete EVent System simulator) je C++ knihovna určená k vytváření modelů s diskrétními událostmi založených na paralelním DEVS a dynamickém DEVS formalismu. Jejím autorem je Dr. Jim Naruto, který získal v roce 2003 doktorát na univerzitě University of Arisona.

ADEVS umožňuje vytvářet hierarchické modely spojováním atomických nebo složených modelů založených na DEVS formalismu a následnou simulaci jejich chování. Implementace simulátoru je založena na novějším přístupu [1] (oproti výše zmíněnému obecnému simulátoru, tak jak ho navrhl Zeigler), který přináší několik výhod:

- dochází k eliminaci nepotřebných simulátorů a koordinátorů u jednotlivých komponent modelu,
- zrychluje se plánování událostí ukládáním odkazů pouze na aktivní modely,
- dochází k odstranění některých nepotřebných synchronizačních zpráv (*-zprávy a d-zprávy).

Celou simulaci opět řídí root koordinátor, jehož funkce je popsána kódem 6.1. Každá devs komponenta obsahuje funkce *compute_and_route_output (CARO)*, *atomic_model_delta_function (AMDF)* a *network_executive_delta_function (NEDF)*, které jsou rekurzivně volány, počínaje koordinátorem, v závislosti na propojení jednotlivých komponent. První funkce, CARO, počítá výstupní funkci všech bezprostředních atomických následníků v čase t_n a jejich výstupy přesměrovává na vstupy připojených komponent. Druhá funkce, AMDF, volá funkce vnitřních přechodů všech komponent, které jsou v čase t_n aktivní (bezpřostřední následníci nebo komponenty, které přijímají nějaký vstup). Poslední funkce, NEDF, počítá časy dalších plánovaných událostí u všech aktivních komponent v čase t_n.

```
variables:
    t_end    // konec simulace
    child    // odkaz na první komponentu

while (child->tn < t_end) // dokud není konec simulace
{
    child->compute_and_route_output();
    child->atomic_model_delta_function();
    child->network_executive_delta_function();
}
```

Listing 6.1: Root koordinátor u ADEVS

Bohužel, knihovna verze 2.2 neobsahuje simulátor pro real-time systémy. I přes tento nedostatek byla použita jako základ pro modelování a simulaci chování agenta robotického fotbalu.

6.1.2 Boost C++ Libraries

Boost C++ Libraries je kolekce multiplatformních knihoven s otevřeným kódem, které rozšiřují funkcionalitu jazyka C++. Poslední verze obsahuje přes 80 samostatných knihoven pro lineární algebru, pseudonáhodné generátory čísel, zpracovávání obrazu, vícevláknové aplikace a další, z nichž některé se plánují zahrnout mezi standardní knihovny nově připraveného standartu jazyka C++ označovaného jako standart C++0x. Kompletní výčet obsahu knihoven Boost lze nalézt na oficiálních stránkách.

Tato kolekce obsahuje i knihovnu *Boost.Asio*, která programátorovi usnadňuje synchronní i asynchronní přístup k I/O (vstupně/výstupní) objektům, jako jsou například síťové sokety. Proto jsem si tuto knihovnu vybral pro realizaci UDP/IP komunikace mezi Soccer serverem a vlastním klientem.

Další často používanou knihovnou ve výsledném klientovi je knihovna pro podporu vícevláknového programování *Boost.Threads*. Ta poskytuje všechny základní prostředky jako jsou vlákna, zámky, semafory, podmíněné proměnné a další. Díky ní výsledná aplikace využívá všech výhod, které vícevláknový přístup nabízí.

6.1.3 Parser

Jak již bylo zmíněno dříve, Soccer server komunikuje s připojenými klienty pomocí s–výrazů. Každý s–výraz nese strukturovanou informaci v podobě jednoho řetězce, jinými slovy v jednom řetězci je uloženo hned několik informací. Aby tyto informace klient mohl využít, musí být řetězec rozložen na jednotlivé části, ze kterých lze pak už danou informaci snadno získat. O rozkládání a získávání informací z řetězců se starají nástroje zvané *parsery*.

Speciálně pro RoboCup Soccer server byl vytvořen parser v podobě C++ knihovny s názvem RoboCup Client Parser [12]. Jeho základem jsou nástroje pro lexikální a syntaktickou analýzu – Flex a Bison, a tak by podle slov autora měl být rychlejší, než vlastní psané parsery. Poslední dostupná verze parseru, verze 1.2.5, umí pracovat s komunikačním protokolem serveru verze 7 až 9.

6.2 Upravený simulátor a DEVS formalismus

6.2.1 Simulátor

Po prostudování manuálů k nástroji ADEVS jsem zjistil, že tato knihovna přímo nepodporuje real-time simulace. Proto jsem si na základě poznatku z [16] vytvořil vlastní pseudo real-time simulátor. Využil jsem myšlenky rozdělit simulátor na část reaktivní a periodickou. Reaktivní bude reagovat na nově vzniklé události, periodická bude provádět naplánované vnitřní události. Každá komponenta modelu tvořena atomickým DEVSem má svojí specifickou úlohu v podobě nějakého výpočtu. Komponenta příjme vstupní zprávu, tu zpracuje a výsledky co nejrychleji pošle dál. Na tyto události reaguje reaktivní část simulátoru. A protože všechny modelované komponenty pracují na stejném principu, zpracovat a poslat dál, není plánovací periodická část simulátoru potřeba. V navrženém modelu se nevyskytují žádné DEVS komponenty, u kterých by bylo potřeba plánovat vnitřní události na dobu pozdější než hned na následný krok.

Důležité je rozesílání zpráv mezi propojenými DEVS komponentami. O tuto činnost se stará vestavěný vnitřními událostmi řízený simulátor ADEVSu, jehož princip byl nastíněn v kapitole 4.2.

Princip výsledného simulátoru vypadá takto (kód 6.2). Simulátor obsahuje 3 klíčové proměnné. První, *a_simulator*, představuje simulátor ADEVS knihovny, který se v tomto případě stará o šíření zpráv v modelu. Zbylé dvě, *condition* a *responce*, jsou podmíněné proměnné, kterými se řídí chod vláken. K těmto podmíněným proměnným mají přístup všechny DEVS komponenty modelu.

Smyčka simulace běží ve vlastním vlákně až do skončení hry. Na začátku se vlákno simulátoru uspí podmíněnou proměnnou *condition* a čeká na výzvu od některé komponenty, že dokončila výpočet a chce výsledek předat dál. Podmínka *isGo* řeší v simulátoru situace, kdy žádost o vyzvednutí výsledku přijde v době, kdy simulátor zrovna provádí krok simulace, a tak by mohlo dojít k „zapomenutí“ tohoto požadavku. Nastavením podmínky *isGo* na *true* říkáme simulátoru, že během jeho činnosti přišel minimálně jeden další požadavek, a proto se nemá uvést do stavu čekání, ale místo toho má pokračovat v dalším kroku simulace. O dokončení kroku simulace může naopak simulátor informovat všechny potenciální čekající vlákna DEVS komponent pomocí podmíněné proměnné *response.notify()*.

Z ukázkového kódu jsou pro přehlednost vypuštěny zámky kritických sekci. Před přístupem do každé sdílené proměnné by mělo dojít k uzamknutí kontextu. Pouze jeden zámek je v příkladu uveden *response.lock()*, a to z toho důvodu, že tento zámek uzamyká nejen přístup ke sdílené proměnné, ale je použit také jako zámek na celý krok simulace. Každá DEVS komponenta, která chce změnit svůj vnitřní stav, musí počkat, až dojde k dokončení kroku simulace, aby nemohlo dojít k nedefinovaným událostem, kdy například při volání výstupní funkce se bude komponenta nacházet v určitém stavu, avšak následně při výpočtu funkce času už ve stavu jiném.

```
variables:
    a_simulator // vestavěný adevs simulátor
    condition   // podmíněná proměnná, spouští krok simulace
    response    // podmíněná proměnná, oznamuje konec kroku

while (!end) // dokud neni konec hry
{
    if(!condition.isGo) // když během kroku nebyl další požadavek
        condition.wait(); // uspi simulátor a čekej na povel

    condition.isGo=false; // vynuluje žádost o další kolo

    response.lock();     // důležitý zámek kritické sekce
    a_simulator->execNextEvent();    // proveď krok simulace
    response.unlock();   // odemkni sekci
    response.notify();   // informuj čekající vlákna o konci kola
}
```

Listing 6.2: Princip vlastního pseudo real-time simulátoru

6.2.2 Atomická DEVS komponenta

Celý model je složený pouze z Atomických DEVSů. Každá komponenta má na starost nějaký výpočetní problém. Například jedna komponenta zpracovává vizuálníc informace, jiná komponenta rozhoduje o příštím kroku výsledného modelu a podobně. Aby chování komponent mohlo být simulováno vytvořeným simulátorem, obsahují DEVS modely vedle klasických proměnných a funkcí navíc ještě dvě již zmíněné podmíněné proměnné, jednu funkci *aktivitu*, která pracuje v samostatném vlákně a provádí hlavní výpočty dané komponenty, *aktivační funkci*, která rozhoduje o spuštění aktivity a o změně vnitřního stavu, a pomocný vnitřní *stav vstupu*, na jehož základě se aktivační funkce rozhoduje.

Model mého atomického DEVSu potom vypadá takto (zápis s využitím vaků (*bags*) byl převzat z textu o ADEVS simulátoru [1]):

$$AM = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.1}$$

kde

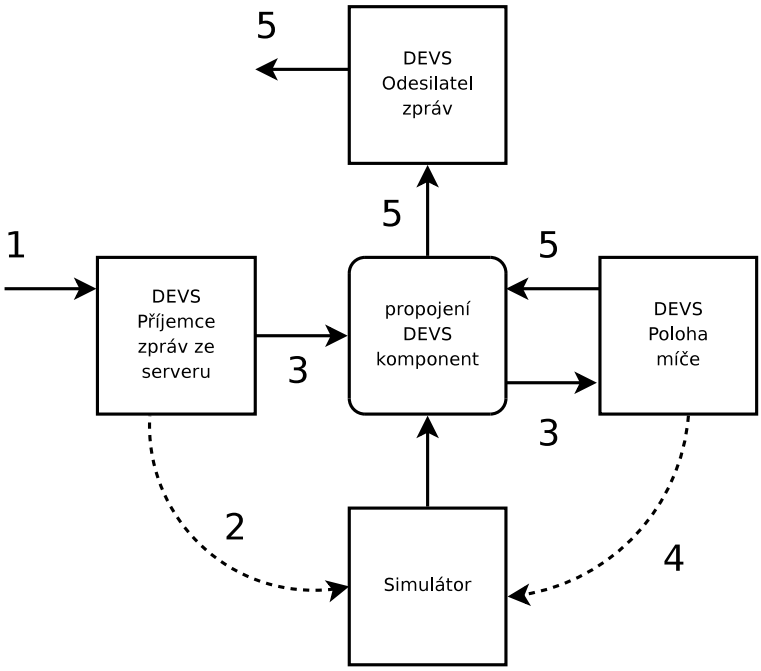
- X je množina vstupů,
- Y je množina výstupů,
- S je množina vnitřních stavů,
- $T = 2^X$ je množina pomocných stavů modelu v závislosti na aktuálnosti vstupů. Každý příchozí vstup se přidá (pokud tam již není) do množiny *aktuální stav vstupů* a na základě tohoto stavu se rozhoduje volání aktivity,
- a je aktivita, funkce, která provádí konkrétní výpočet komponenty,
- δ_{ext} je externí přechodová funkce. Je v modelu kvůli zachování kompatibility. Sama o sobě přímo nemění stav komponenty. Pouze přidává prvky z vaku vstupů X^b ($X^b = 2^X$) do aktuálního stavu vstupů a volá aktivační funkci.
- $\delta_{int} : S \to S$ je interní přechodová funkce,
- $\lambda : S \to Y^b$ je výstupní funkce, kde Y^b je množina vaků nad Y prvky ($Y^b = 2^Y$),
- $ta : S \to \{1, \infty\}$ je funkce posunu času, udává kdy dojde k výskytu následující vnitřní události, v tomto případě tedy hned příští krok simulace nebo nikdy,
- $\alpha : S \times T \to \{a, \emptyset\} \times S \times T$ je aktivační funkce, která na základě vnitřního a pomocného vstupního stavu rozhoduje o spuštění aktivity a přechodu vnitřních stavů.

6.2.3 Příklad průběhu simulace

Na obrázku 6.1 je ukázka možného průběhu simulace. Model se skládá ze tří DEVS komponent, simulátoru a objektu ležícího uprostřed, který představuje propojení jednotlivých DEVS modelů.

Na začátku je celkový model inicializován tak, že jedinou naplánovanou událostí je vyzvednutí zprávy z komponenty příjemce zpráv. Ostatní komponenty se nacházejí ve stavu spánku. Taktéž simulátor je pozastaven a čeká na probuzení. Simulace na obrázku probíhá v těchto krocích:

- 1. Do příjemce zpráv přijde externí zpráva ze serveru o tom, co klient vidí. Příjemce ji zpracuje parserem a připraví zprávu pro výstupní událost.
- 2. Pomocí podmíněné proměnné dá simulátoru signál, že má provést jeden krok simulace.
- 3. Naplánovaná událost simulátoru je vyvolat vnitřní funkci příjemce zpráv. Ta se provede a dojde také k volání výstupní funkce příjemce. Díky propojení se tak připravená zpráva dostane do komponenty s polohou míče, kde vyvolá vnější událost. Tato událost změní stav komponenty na *pracující* a v novém vlákně spustí výpočet polohy míče. Nakonec je volána časová funkce komponenty s polohou míče, která díky stavu *pracující* naplánuje další vnitřní událost hned na příští krok simulátoru. Simulátor dokončil jeden krok a uvedl se do stavu spánku.
- 4. Po nějaké době DEVS komponenta s polohou míče dokončí své výpočty a je připravena předat výsledky další komponentě. Změní proto svůj stav na *odesílající* a pomocí podmíněné proměnné probudí simulátor.
- 5. Simulátor opět provede naplánovanou událost, tedy vnitřní událost v DEVS komponentě Poloha míče. Při této příležitosti je také volána výstupní funkce komponenty, která předá zprávu připojené komponentě Odesilatel zpráv. V komponentě Poloha míče dojde ke změně vnitřního stavu opět na *čekající* a časová funkce naplánuje další událost na čas v nekonečnu. Vnější událost v komponentě Odesilatel jen provede poslání příkazu serveru (například o pohybu za míčem) a jeho časová funkce vždy vrátí čas v nekonečnu (tato komponenta jen přijímá, nikdy nevytváří události v DEVS modelu).



Obrázek 6.1: Příklad průběhu simulace

6.3 Výsledný model agenta

Dostáváme se k samotnému návrhu modelu agenta, který bude založen na upraveném DEVS formalizmu. Jeho chování bude simulováno v reálném čase a dílčí výsledky simulace budou použity jako řídící zprávy předávané připojenému fotbalovému serveru. Model i simulátor bude poté implementován v podobě samostatné aplikace schopné účastnit se fotbalového utkání.

6.3.1 Schéma modelu

Výsledné schéma modelu, jednotlivé komponenty a jejich propojení je ukázáno na obrázku 6.2.

Celý model robota představuje jeden složený DEVS model. S okolím komunikuje prostřednictvím dvou portů, *In* a *Out*. Pro upřesnění, ve výsledné implementaci složený DEVS nemá přímo vnější vstup a výstup, komunikace se serverem probíhá na jiné úrovni. Komponenta na obrázku propojena se vstupem *In* se z pohledu DEVS formalismu chová jako producent zpráv (zprávy v ní vznikají) a komponenta přípojená na výstup *Out* se chová jako konzument (zprávy v ní končí). Propojení, jak je uvedeno na obrázku, je jen z důvodu názornosti.

Složený DEVS se skládá pouze z atomických DEVS modelů. Každý model představuje samostatnou jednotku pro řešení konkrétního problému. Rozložení a propojení komponent uvedené na obrázku je můj vlastní návrh založený na poznatcích o složení obecného robota (viz. část 3.1). Popis jednotlivých komponent bude uveden dále v samostatných podkapitolách.

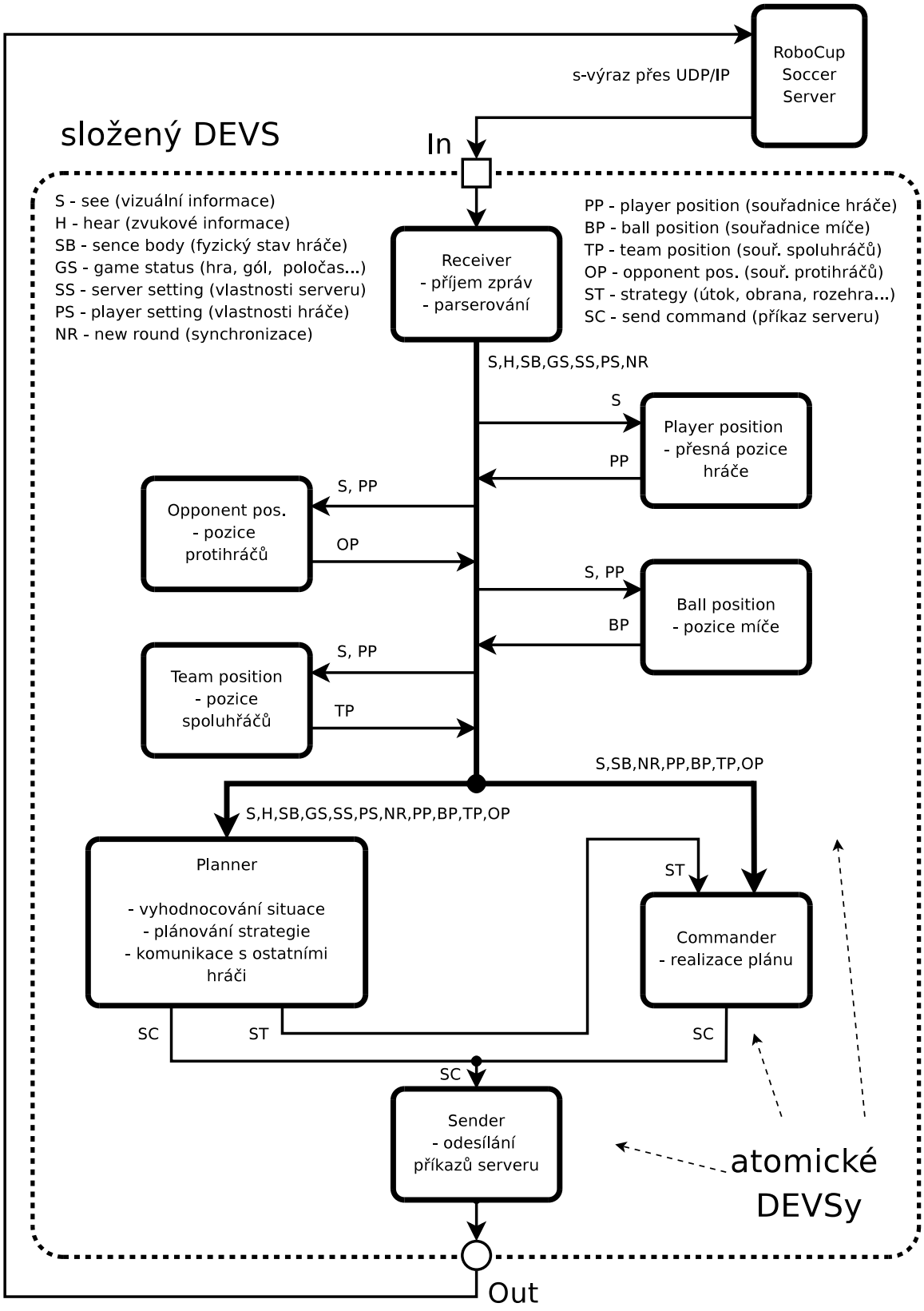
Důležité je propojení jednotlivých komponent. Jak již bylo uvedeno, vlastní navržený simulátor DEVSu má úlohu šiřitele zpráv mezi těmito komponentami. Propojení komponent je definováno na úrovni složeného DEVS modelu. Navržený celkový model agenta pracuje s 13-ti různými typy zpráv:

- *See* – Zpráva s vizuálními informacemi. Tuto zprávu posílá server v pravidelných intervalech, jednou za kolo serverového času. Příchozí informace jsou v podobě množiny viditelných objektů. Viditelnými objekty jsou hráči, míč nebo vlajky okolo hřiště. Každý objekt může obsahovat informace o vzdálenosti, směru a o změně těchto hodnot oproti minulému stavu. Je-li objektem hráč, lze získat navíc informace o příslušném týmu a čísle dresu. Množství informací záleží na vzdálenosti objektu a režimu pohledu (viz. manuál [5]). Pomocí viditelných vlajek, jejichž poloha je známá, lze určit absolutní polohu hráče na hřišti. Informace o přesné poloze objektů server standardně neposkytuje.

- *Hear* – Zpráva se zvukovými informacemi v podobě zpráv od ostatních hráčů. Představuje množinu všech příchozích zpráv, které se skládají z časového razítka, směru příchozí zprávy a obsahu samotného sdělení.

- *Sence Body* – Informace o fyzickém stavu hráče. Obsahuje údaje o aktuální rychlosti hráče a jeho vytrvalosti (množství zbývajících sil) a navíc statistické informace o počtu provedení jednotlivých příkazu jako jsou povel k běhu, kopu, otočení a další.

- *Game Status* – Zpráva s údaji o stavu hry. Obsahuje informace o aktuálním módu hry (výkop, normální hra, trestné střílení…), stavu skóre hry, straně vlastního týmu (levá, pravá), jménu týmu a čísle dresu.



Obrázek 6.2: Schéma složení a propojení výsledného modelu

- *Server Setting* – Po připojení k Soccer serveru klient obdrží kompletní informaci o nastavení serveru. Tato informace obsahuje přes 60 různých parametrů serveru, proto zde opět odkáži na manuál k RoboCup Soccer serveru.
- *Player Setting* – Podobně jako předchozí zpráva je tato zaslána při úvodním připojení a obsahuje kompletní informace o přiřazeném hráči. Nejdůležitější jsou údaje o maximální rychlosti hráče a jeho vytrvalosti.
- *New Round* – Robocup Soccer server přijímá příkazy o pohybu pouze jednou za kolo. Proto je potřeba informovat příslušné komponenty o začátku nového kola. Tato zpráva tak provádí synchronizaci modelu a serveru. Informační složkou zprávy je aktuální čas serveru.
- *Player Position* – Zpráva nese informace o absolutní okamžité poloze hráče v podobě souřadnic [x,y].
- *Ball Position* – Informace o pozici míče získané ze zprávy *See* a dodané informace o absolutní pozici.
- *Team Position* – Zpráva s informacemi o vlastních hráčích získaná ze zprávy *See* a k nim dodané hodnoty absolutních pozic hráčů.
- *Opponent Position* – Stejná struktura jako předchozí zpráva, tentokrát ale informace o protihráčích.
- *Strategy* – Informace o naplánované strategii. Tato zpráva nemá předem stanovený počet parametrů, záleží na implementaci komponenty, která se stará o plánování a rozhodování. V modelovém případě nese tato zpráva informace pouze o aktuálním chování (útoč, jdi za míčem, braň).
- *Send Commad* – Poslední zpráva obsahuje množinu příkazů určených k odeslání zpátky fotbalovému serveru.

Více konkrétních podrobností o obsahu jednotlivých zpráv lze nalézt v programové dokumentaci a v manuálu RoboCup Soccer Server [5].

6.3.2 Komponenta Receiver

První komponentou je komponenta nazvaná *Receiver*. Úkolem této atomické komponenty je přijímat příchozí informace ze serveru, zpracovávat je, třídit do jednotlivých typů zpráv (*See, Hear, Server Status...*) a předávat je dál.

Komponenta obsahuje dva pomocné objekty. Prvním je UDP klient, který umožňuje přijímat příchozí zprávy pomocí UDP/IP protokolu. Příchozí zprávy v podobě řetězců zpracovává druhý objekt, parser. Základem mého parseru je již zmiňovaný RoboCup Client Parser (viz. 6.1.3).

Třetí významnou činností komponenty je synchronizace celého modelu. Na základě informací ze serveru určuje okamžik nového kola serveru. V ukázkovém programu je tento údaj získáván z vizuální informace, kterou server posílá jednou za kolo.

Z pohledu DEVS formalismu se tato komponenta od ostatních liší absencí vstupu. Pro simulátor představuje producenta zpráv, proto tato komponenta nemá definované vnitřní a aktivační funkce, vnitřní stav je stále stejný a aktivita po inicializačním spuštění běží v nekonečné smyčce. Je zde také využita druhá podmíněná proměnná, která informuje

o skončení kola simulace. Vnitřní nekonečná smyčka vždy čeká na tento povel před čtením další zprávy ze serveru.

Formální definice DEVS komponenty *Receiver*:

$$DCReceiver = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.2}$$

kde

- $X = \emptyset$ – komponenta nemá žádné vstupy,
- $Y = \{See, Hear, SenceBody, GameStatus, ServerSetting, PlayerSetting, NewRound\}$ – výstupní zprávy,
- $S = \{Sending\}$ – vnitřní stav, tato komponenta stále posílá výsledky,
- $\delta_{ext} = \emptyset$ – komponenta nemá vstup, nikdy se nevolá,
- $\delta_{int} = \{Sending \rightarrow Sending\}$ – komponenta nemění svůj vnitřní stav,
- $\lambda = \{Sending \rightarrow y^b | y^b \in Y^b\}$ – výstupní funkce, obsah vaku y^b je závislý na aktuálně zpracovaných informací ze serveru,
- $ta = \{Sending \rightarrow 1\}$ – komponenta má vždy naplánovanou vnitřní událost na příští krok,
- $\alpha = \emptyset$ – komponenta nemá vstup, funkce se nevolá.

6.3.3 Komponenta Player Position

RoboCup Soccer Server úmyslně neposkytuje mezi vizuálními informacemi přesné údaje o poloze v podobě absolutních souřadnic. Pouze informace o směru a vzdálenosti objektů. Pro plánování pohybu po hřišti je však vhodné mít přesnou představu o aktuální pozici hráče. O určení polohy hráče se stará komponenta *Player position*. Za tímto čelem jsou serverem poskytovány informace o směru a vzdálenosti viditelných vlajek (*flags*) rozmístěných kolem hřiště, u nichž je známá přesná poloha. Příklad rozmístění orientačních bodů RoboCup serveru je na obrázku 6.3 převzatého z manuálu.

Z rozmístění orientačních bodů pak pomocí analytické geometrie nebo například pomocí zajímavější metody tzv. *Markov Localization* [3] můžeme určit přesnou polohu $[x, y]$ hráče na hřišti. Lze si vybrat libovolný algoritmus a ten implementovat do funkce aktivity této DEVS komponenty.

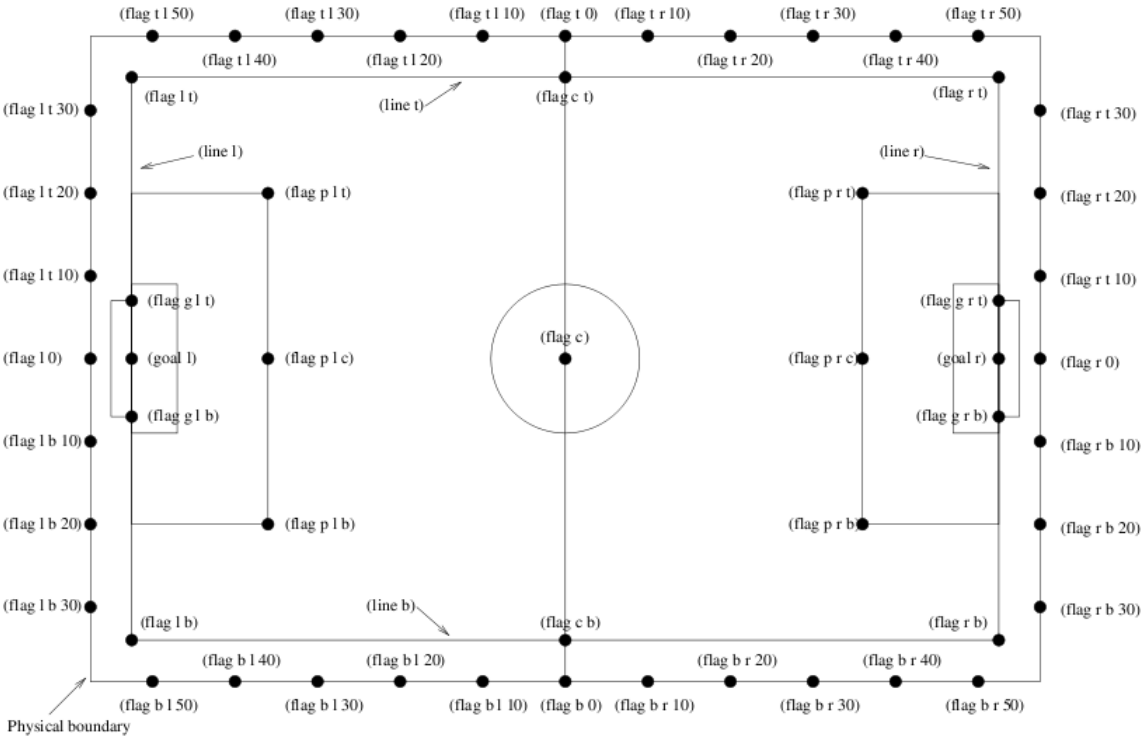
Formální definice DEVS komponenty *Player Position*:

$$DCPlayerPosition = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.3}$$

kde

- $X = \{See\}$ – jediný vstup vizuální informace,
- $Y = \{PlayerPosition\}$ – jediný výstup informace o pozici,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,

- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow \{PlayerPosition\}\}$ – výstupní funkce, odevzdá zprávu o poloze hráče,
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,
- $\alpha = \{(Waiting, \{See\}) \rightarrow (a, Working, \emptyset)\}$ – aktivita se spouští tehdy, když je vstup aktuální a komponenta čeká.



Obrázek 6.3: Rozmístění orientačních bodů okolo hřiště [5]

6.3.4 Komponenty Ball Position, Team Position, Opponent Position

Jak už název napovídá, jedná se o komponenty pro výpočet přesné pozice míče, spoluhráčů a protihráčů. Struktura i implementace komponent je podobná. Na základě zpráv *See* a *Player Position* dopočítávají absolutní souřadnice příslušných objektů. Ačkoli je úkol podobný, zvolil jsem koncept tří nezávislých komponent, abych ukázal schopnosti modelu zpracovávat informace paralelně.

Při příchodu některé vstupní zprávy se tento vstup přidá do množiny aktuálního stavu vstupů. Pokud je splněna aktivační podmínka, tedy komponenta nepracuje a v množině aktuálního vstupu jsou obě požadované zprávy, spouští se aktivita těchto komponent. Ta

ze zprávy *See* zkopíruje příslušné údaje (podle typu komponenty) a k nim ze znalosti pozice hráče analytickou geometrií dopočítá aktuální souřadnice $[x, y]$. Výsledky zabalí do výstupní zprávy *Ball Position*, *Team Position* respektive *Opponent Position* a tu v příštím kole simulátor odešle dalším připojeným komponentám.

Formální definice DEVS komponenty *Ball Position*:

$$DCBallPosition = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.4}$$

kde

- $X = \{See, PlayerPosition\}$ – vstup vizuální informace a údaje o přesné pozici hráče,
- $Y = \{BallPosition\}$ – jediný výstup informace o pozici míče,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,
- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow \{BallPosition\}\}$ – výstupní funkce, odevzdá zprávu o poloze míče,
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,
- $\alpha = \{(Waiting, \{See, PlayerPosition\}) \rightarrow (a, Working, \emptyset)\}$ – aktivita se spouští tehdy, když je vstup aktuální a komponenta čeká.

Formální definice DEVS komponenty *Team Position*:

$$DCTeamPosition = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.5}$$

kde

- $X = \{See, PlayerPosition\}$ – vstup vizuální informace a údaje o přesné pozici hráče,
- $Y = \{TeamPosition\}$ – jediný výstup informace o pozici hráčů z týmu,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,
- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow \{TeamPosition\}\}$ – výstupní funkce, odevzdá zprávu o poloze hráčů z týmu,
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,

- $\alpha = \{(Waiting, \{See, PlayerPosition\}) \rightarrow (a, Working, \emptyset)\}$ – aktivita se spouští tehdy, když je vstup aktuální a komponenta čeká.

Formální definice DEVS komponenty *Opponent Position*:

$$DCOpponentPosition = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.6}$$

kde

- $X = \{See, PlayerPosition\}$ – vstup vizuální informace a údaje o přesné pozici hráče,
- $Y = \{OpponentPosition\}$ – jediný výstup informace o pozici hráčů soupeře,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,
- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow \{TeamPosition\}\}$ – výstupní funkce, odevzdá zprávu o poloze hráčů soupeře,
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,
- $\alpha = \{(Waiting, \{See, PlayerPosition\}) \rightarrow (a, Working, \emptyset)\}$ – aktivita se spouští tehdy, když je vstup aktuální a komponenta čeká.

6.3.5 Komponenta Planner

Komponenta *Planner* reprezentuje kognitivní jednotku robota. Tato část na základě všech dostupných informací rozhoduje a plánuje výsledné chování agenta.

Komponenta je implementována podobně jako ostatní atomické DEVS komponenty. Význačná je jen větším počtem svých vstupů. Rozhodovací algoritmus je implementován do funkce aktivity. Podoba algoritmu opět není ničím limitována. Programátor má celý objekt pod kontrolou a záleží jen na něm, jak příchozí informace využije. Algoritmus rozhodování tak může být založen například na principu rozhodovacích stromů, konečných automatů, petriho sítí, nebo se komponenta může rozhodovat opět na základě nějakého DEVS modelu. Ve výsledné implementaci modelového příkladu je implementován jednoduchý rozhodovací strom.

Tato komponenta může své rozhodnutí zakládat také na spolupráci s ostatními hráči v týmu. RoboCup Soccer simulátor umožňuje komunikovat mezi hráči pomocí „hlasových“ zpráv. Proto je v příkladu uveden i výstup v podobě zprávy *Send Command*, kterým může komponenta přímo posílat serveru příkazy *Say*.

Formální definice DEVS komponenty *Planner*:

$$DCPlanner = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.7}$$

kde

- $X = \{See, Hear, SenceBody, GameStatus, ServerSetting, PlayerSetting, NewRound, PlayerPosition, BallPosition, TeamPosition, OpponentPosition\}$ – vstup veškeré dostupné informace,
- $Y = \{Strategy, SendCommand\}$ – výstupní zprávy pro realizátor plánů nebo zpráva s příkazem pro server,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,
- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow y^b\}\}$ – výstupní funkce, obsah vaku y^b závisí na implementaci aktivity,
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,
- $\alpha = \{(Waiting, x^b) \rightarrow (a, Working, \emptyset) | x^b \in (2^X \setminus \{\emptyset\})\}$ – aktivita se spouští tehdy, když je vstup aktuální. Tato podmínka závisí na implementaci, v ukázkovém příkladě se aktivita spouští po každé vstupní události (stav vstupů je neprázdná množina).

6.3.6 Komponenta Commander

Tato komponenta v modelu robota představuje realizátora plánů. Na základě vizuálních informací a údajů o fyzickém stavu hráče plní poskytnutý plán. Ten může být vybrán na základě zvolené strategie z databáze připravených vzorů chování, nebo může být přímo sestavený plánovací komponentou. Opět je vše v režii programátora a funkce aktivity může vykonávat libovolnou činnost.

V ukázkové implementaci tato komponenta vykonává vestavěné plány podle strategie, kterou zvolil *Planner*. V ukázce jsou realizovány strategie *útoč, braň, běž za míčem, vystřel na bránu.*

RoboCup Soccer simulátor vyžaduje, aby příkazy pohybu (*dash, turn, kick*) přicházely vždy jednou za kolo serveru. Proto je nutné synchronizovat tyto události se serverem pomocí zprávy *New Roud*. Toto je jediná podmínka se strany serveru, jinak lze funkci aktivity implementovat libovolně.

Formální definice DEVS komponenty *Commander*:

$$DCCommander = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.8}$$

kde

- $X = \{See, SenceBody, NewRound, PlayerPosition, BallPosition, TeamPosition, OpponentPosition\}$ – vizuální informace a důležitá zpráva o novém kole *NewRound*,
- $Y = \{SendCommand\}$ – výstupní zpráva s příkazem pro server,
- $S = \{Waiting, Working, Sending\}$ – komponenta se může nacházet ve 3 hlavních stavech: čeká, pracuje, odevzdává výsledky,

- δ_{ext} – přidá vstup do množiny *aktuální stav vstupů* (pokud již neobsahuje), tato množina je prvkem 2^X,
- $\delta_{int} = \{Sending \rightarrow Waiting\}$ – po odevzdání výsledků se vrátí do stavu čekání,
- $\lambda = \{Sending \rightarrow \{SendCommand\}\}$ – výstupní funkce
- $ta = \{(Working \rightarrow 1), (Sending \rightarrow 1), (Waiting \rightarrow \infty)\}$ – při čekání plánovaná událost v nekonečnu, jinak příští krok,
- $\alpha = \{(Waiting, x^b) \rightarrow (a, Working, \emptyset) | x^b \in 2^X \wedge NewRound \in x^b\}$ – aktivita se spouští tehdy, když je mezi aktuálními vstupy vstup *NewRound*.

6.3.7 Komponenta Sender

Poslední komponentou v ukázkovém modelu je komponenta *Sender*. Jejím úkolem je předávat příkazy fotbalovému serveru přes síťové rozhraní UDP/IP. Tak jako komponenta *Receiver* neměla žádný faktický vstup do DEVS modelu, tato nemá žádný DEVS výstup a pro simulátor vystupuje jako konzument zpráv. Pouze zprávy přijímá, žádné nevytváří.

Návrh této komponenty je proto jednoduchý. Z pohledu atomického DEVS modelu komponenta nemění svůj stav, neplánuje vnitřní přechody, nepotřebuje funkci aktivity ani aktivační funkci. Reaguje pouze na externí událost přeposláním obsahu příchozí zprávy *Send Command* fotbalovému serveru.

Formální definice DEVS komponenty *Sender*:

$$DCSender = (X, Y, S, T, a, \delta_{ext}, \delta_{int}, \lambda, ta, \alpha) \tag{6.9}$$

kde

- $X = \{SendMessage\}$ – jediný vstup, zpráva s příkazem serveru,
- $Y = \emptyset$ – žádný výstup,
- $S = \{Waiting\}$ – komponenta stále jen čeká na vstupní událost,
- δ_{ext} – provádí posílání zpráv přes síťové rozhraní serveru,
- $\delta_{int} = \{Waiting \rightarrow Waiting\}$ – tato funkce je jen kosmetická, nikdy k volání nedojde,
- $\lambda = \emptyset$ – výstupní událost nikdy nenastane,
- $ta = \{Waiting \rightarrow \infty\}$ – funkce času vždy vrací nekonečno,
- $\alpha = \emptyset$ – aktivační funkce také není potřeba.

Uvedený formální popis je založen na finální implementaci komponenty v ukázkovém modelu. V tomto případě jsou nevyužita rozšíření v podobě aktivity a podmíněných proměnných. Výhodou je jednoduchost tohoto řešení, nevýhodou jsou potenciální problémy s rychlostí předávání zpráv. V uvedeném řešení musí simulátor čekat na dokončení odesílání zprávy serveru. Pokud bude více zpráv pohromadě a server je nebude schopen dostatečně rychle obsloužit, mohlo by docházet k nepříjemným prodlevám při šíření zpráv uvnitř modelu. Převedením algoritmu odesílání zpráv do funkce aktivity, která pracuje ve vlastním vlákně, by se dalo těmto potenciálním problému zabránit.

6.3.8 Celkový složený DEVS model

Aby simulátor věděl, komu jaké zprávy předávat, musíme definovat množinu propojení jednotlivých komponent. Propojení můžeme formálně popsat modelem složeného DEVSu.

Formální definice složeného DEVS moedlu výsledné aplikace

$$SoccerClient = (X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}) \tag{6.10}$$

kde

- $X = \emptyset$ je prázdná množina vstupů. Jak jsme již zmínili, z pohledu DEVS formalismu nejsou ve složeném modelu žádné vnější vstupy a výstupy, komunikace modelu s okolím probíhá na jiné úrovni (síťová komunikace),
- $Y = \emptyset$ je taktéž prázdná množina, žádný vnější výstup,
- $D = \{Receiver, PlayerPosition, BallPosition, TeamPosition$ $OpponentPosition, Planner, Commander, Sender\}$ je množina jmen odkazů jednotlivých atomických komponent modelu,
- $\{M_i\}$ je množina DEVS komponentů, kde pro všechny $i \in D$ je M_i atomický DEVS,
- $C_{xx} = \emptyset$ je prázdná množina, nejsou žádná propojení s vnějšími vstupy
- $C_{yy} = \emptyset$ je opět prázdná množina, nejdou definována žádná spojeni s vnějšími výstupy
- $C_{yx} = \{$
  $(Receiver.outSee, PlayerPosition.inSee)$,
  $(Receiver.outSee, BallPosition.inSee)$,
  $(Receiver.outSee, TeamPosition.inSee)$,
  $(Receiver.outSee, OpponentPosition.inSee)$,
  $(Receiver.outSee, Planner.inSee)$,
  $(Receiver.outSee, Commander.inSee)$,
  $(Receiver.outHear, Planner.inHear)$,
  $(Receiver.outHear, Planner.inHear)$,
  $(Receiver.outSenceBody, Planner.inSenceBody)$,
  $(Receiver.outSenceBody, Commander.inSenceBody)$,
  $(Receiver.outGameStatus, Planner.inGameStatus)$,
  $(Receiver.outServerSetting, Planner.inServerSetting)$,
  $(Receiver.outPlayerSetting, Planner.inPlayerSetting)$,
  $(Receiver.outNewRound, Planner.inNewRound)$,
  $(Receiver.outNewRound, Commander.inNewRound)$,
  $(PlayerPosition.outPlayerPosition, BallPosition.inPlayerPosition)$,
  $(PlayerPosition.outPlayerPosition, TeamPosition.inPlayerPosition)$,
  $(PlayerPosition.outPlayerPosition, OpponentPosition.inPlayerPosition)$,
  $(PlayerPosition.outPlayerPosition, Planner.inPlayerPosition)$,
  $(PlayerPosition.outPlayerPosition, Commander.inPlayerPosition)$,
  $(BallPosition.outBallPosition, Planner.inBallPosition)$,
  $(BallPosition.outBallPosition, Commander.inBallPosition)$,
  $(TeamPosition.outTeamPosition, Planner.inTeamPosition)$,
  $(TeamPosition.outTeamPosition, Commander.inTeamPosition)$,

$(OpponentPosition.outOpponentPosition, Planner.inOpponentPosition)$,
$(OpponentPosition.outOpponentPosition, Commander.inOpponentPosition)$,
$(Planner.outStrategy, Commander.inStrategy)$,
$(Planner.outSendMessage, Sender.outSendMessage)$,
$(Commander.outSendMessage, Sender.outSendMessage)$
} je množina vnitřních propojení, jejíž výčet je celkem dlouhý, možná až zbytečně dlouhý, ale uvedl jsem ho proto, aby byl formální popis mého modelu kompletní. Propojení je uváděno jako dvojice portů příslušných komponent.

Kapitola 7

Výsledky

Při návrhu programové části projektu byl kladen velký důraz na možnost představený DEVS simulátor a model celého řízení robota implementovat do podoby knihovny, která by usnadňovala tvorbu a testování klientů pro robotický fotbal.

Simulátor i model byl implementován přesně v podobě, jak byl uveden v předchozí kapitole. Do zamýšleného stavu knihovny (zapouzdřit jednotlivé komponenty do podoby bázových tříd) se nepodařilo z časových důvodů programovou část dovést. Přesto výsledná aplikace je schopna připojit se k serveru a komunikovat s nim. Parser umí zpracovat většinu významných informací ze serveru a vhodně je převést do podoby vnitřních zpráv modelu. Jsou implementovány všechny zmíněné komponenty a jejich základní funkce. Každá funkce aktivity příslušné komponenty obsahuje základní algoritmus pro danou činnost a je schopna tyto výpočty provádět samostatně ve vlastním vlákně, čímž přináší možnosti potenciálního zvýšení výkonu klienta na víceprocesorových architekturách.

Co se týče testování, zaměřil jsem se především na ověření funkčnosti a použitelnosti návrhu simulátoru. Nemohu srovnávat například hardwarovou náročnost svého návrhu s jinými řešeními, protože nebyla k dispozici. Implementace i testování probíhalo na dvoujádrovém procesoru Intel 2x800 MHz. Za celou dobu testování nebyly zaznamenány žádné výkonnostní komplikace. Ukázková verze klienta zvládala (bez znatelného zvýšení zátěže procesoru) zpracovávat, vyhodnocovat a odpovídat na informace ze serveru i se základním nastavením doby trvání jednoho kola o délce 100 milisekund. Původní obavy z nedostatečně rychlého šíření zpráv mezi komponentami tak byly při testování vyvráceny. Na základě těchto poznatků jsem přesvědčen, že uvedený návrh modelu a jeho simulátoru je použitelný jako základ pro vytváření vlastních agentů robotického fotbalu. Tento model může být použit jako kostra klienta, kdy se uživatel bude moct plně soustředit na návrh a implementaci konkrétního chování jednotlivých komponent a nebude se muset zabývat síťovou komunikací se serverem, zpracováním příchozích zpráv, paralelním během jednotlivých komponent, šířením zpráv v modelu a dalšími základními problémy, které již tato kostra řeší.

Pro ilustraci uvádím obrázek s výstupem implementovaného klienta:


```
76 COMM: strategie jdi za micem
vidim mic
dir: 31
dist: 22.2
SEND: posilam zpravu: (turn 15.5)
77 PP: Pocitam polohu hrace...
77 BP: Pocitam presnou polohu mice...
77 TP: Pocitam pozice spoluhracu...
77 OP: Pocitam pozice protihracu
77 COMM: strategie jdi za micem
vidim mic
dir: 26
dist: 20.1
SEND: posilam zpravu: (dash 80)
78 PP: Pocitam polohu hrace...
78 BP: Pocitam presnou polohu mice...
78 TP: Pocitam pozice spoluhracu...
78 OP: Pocitam pozice protihracu
78 COMM: strategie jdi za micem
vidim mic
dir: 27
dist: 20.1
SEND: posilam zpravu: (dash 80)
```

Obrázek 7.1: Výstup pomocných informací klienta při hře. 2D monitor je součástí RoboCup

Kapitola 8

Závěr

Cílem této práce bylo s využitím DEVS formalismu navrhnout a popsat model chování agenta hrajícího robotický fotbal.

Obecně se robot (agent) skládá z několika částí. Má své senzorické, motorické a kognitivní jednotky, jejichž funkce charakterizují výsledné chování robota. DEVS formalismus nám umožňuje popisovat funkci těchto komponent každé zvlášť a výsledný model chování získat pozdějším propojením jednotlivých komponent. To vše lze definovat s matematickou přesností DEVS formalismu.

Chování vytvořeného modelu můžeme simulovat pomocí DEVS simulátorů. Jednotlivé mezivýsledky simulace lze poté využít jako řídící zprávy, například pro ovládání fotbalového robota.

Za tímto účelem byl klasický DEVS simulátor upraven na vlastní verzi realtime paralelního simulátoru tak, aby byl schopen provádět simulaci modelu v reálném čase a mezivýsledky simulace použít na řízení externích zařízení. V návrhové části textu byl představen princip takového simulátoru v podobě „šiřitelé“ vnitřních zpráv mezi komponentami modelu, na jejichž základě mohou tyto komponenty paralelně vykonávat svoji činnost. Funkčnost simulátoru byla poté ověřena na vytvořeném modelu chování agenta v podobě softwarového klienta připojeného k simulátoru robotického fotbalu RoboCup Soccer.

Robotický fotbal je disciplína velmi rozsáhlá. Spojuje v sobě prvky z různých vědeckých oblastí. I ve zjednodušené kategorii simulovaného fotbalu jsou zapotřebí techniky z navigace, plánování, umělé inteligence a mnoho dalších. Každá z těchto oblastí by svým objemem vydala na několik takovýchto prací. Proto jsem se při návrhu svého modelu agenta zaměřil převážně na dekompozici celého problému na jednotlivé komponenty podle jejich účelu v modelu, než na konkrétní funkci každé z nich. Výsledkem je tak obecný model složený z několika komponent, kde každá přestavuje řešení některého z uvedených problémů (poloha, navigace, plánování…).

Programová část projektu se tedy skládá z implementace vlastního DEVS simulátoru, modelu chování robota v podobě několika samostatných komponent řešících dílčí problémy a z rozhraní, přes které je možné komunikovat s oficiálním simulátorem robotického fotbalu iniciativy RoboCup Soccer.

Výsledkem práce je ukázka použití vlastního návrhu DEVS simulátoru pro řízení a předávání zpráv mezi komponentami v reálném čase. Jednotlivé komponenty popsané DEVS formalizmem jsou schopny svou činnost provádět paralelně, nezávisle na stavu simulátoru. Dále byl implementován prototyp knihovny pro snadné vytváření vlastních klientů robotického fotbalu, speciálně pro simulovaný fotbal na serverech iniciativy RoboCup Soccer.

Základem knihovny je představený simulátor a model. Knihovna dále řeší síťovou komunikaci se serverem, parserování příchozích zpráv a celkovou synchronizace mezi klientem a serverem.

Knihovna je stále ve fázi vývoje. Aktuální verze obsahuje na implementovaný simulátor, jednotlivé DEVS komponenty a jejich propojení, síťové rozhraní pro UDP/IP komunikace a parser zpracovávající informace ze serveru. A právě parser může být předmětem dalšího vývoje knihovny, neboť RoboCup server je schopný posílat přes 100 různých informací, které by mohli být využity modelem. Parser v aktuální verzi umí prozatím zpracovávat pouze ty nejzákladnější.

Literatura

[1] A. Muzy, J. J. Naruto: Algorithms for efficinet implementations of the DEVS & DSDEVS abstract simulators. Blaise Pascal University, France, 2005.

[2] B. P. Zeigler, S. B. Hall and H. S. Sarjoughian: Exploiting HLA and DEVS To Promote Interoperability and Reuse in Lockheed's Corporate Environment. University of Arizona, USA, november 1999.

[3] C. Penedo, J. Pavao, P. Lima and M. I. Ribeiro: Markov Localization In The RoboCup Simulation League. Instituto Superior Técnico, Portugal.

[4] J. Lee, M. Kimi and S. Ch: Hierarchical Command & Control System Modeling in Distributed Virtual Warfare Simulation Environment. University of Arizona, USA.

[5] Kolektiv autorů: RoboCup Soccer Server. Users manual, 2003-02-11.

[6] Orság, F.: *Robotika*. FIT VUT v Brně, 2006-11-08.

[7] Suchý, V.: Modelování agentů pro robotický fotbal. Semestrální práce, 2009-01-13.

[8] www stránky: Devs Tolls [online]. `http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm`.

[9] www stránky: Fira — Federation of International Robot-soccer Association [online]. `http://www.fira.net/`.

[10] www stránky: RoboCup 2008 SUZHOU CHINA [online]. `http://robocup-cn.org/`.

[11] www stránky: RoboCup 2009 GRAZ AUSTRIA [online]. `http://www.robocup2009.org/`.

[12] www stránky: The RoboCup Client Parser (RCCParser) [online]. `http://rccparser.sourceforge.net/doc/html/index.html`.

[13] www stránky: RoboCup Official Site [online]. `http://www.robocup.org/02.html`.

[14] www stránky: Sony AIBO Europe - Official Website [online]. `http://support.sony-europe.com/aibo/`.

[15] www stránky: RoboCup: Brief of History [online]. `http://www.robocup.org/overview/23.html`, 1998 [cit. 2008-12-28].

[16] Y. K. Cho, B. P. Zeigler and H. S. Sarjoughian: Design and Implementation of Distributed Real-Time DEVS/CORBA. University of Arizona, USA.

[17] Zbořil, F.: *Agentní a multiagentní systémy.* FIT VUT v Brně, 2006.

[18] Zeigler, B. P.: *Theory of modeling and simulation.* San Diego: Academic Press, 2000, iSBN 0-12-778455-1.