VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

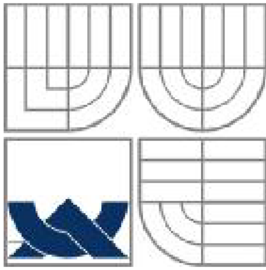
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# MOBILNÍ AGENTI V BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH

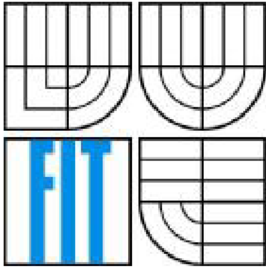
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE PAVEL SPÁČIL
AUTHOR

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


# MOBILNÍ AGENTI V BEZDRÁTOVÝCH SENZOROVÝCH SÍTÍCH

MOBILE AGENTS IN WIRELESS SENSOR NETWORKS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS


AUTOR PRÁCE PAVEL SPÁČIL
AUTHOR

VEDOUCÍ PRÁCE Ing. FRANTIŠEK ZBOŘIL, PhD.
SUPERVISOR


BRNO 2009

## Abstrakt

Cílem této práce je popsat návrh platformy a implementaci interpretu pro mobilní agenty v bezdrátových senzorových sítích. Čtenář se seznámí jak s teoretickou, tak i praktickou stránkou věci - nabude informace o jednoduchém operačním systému TinyOS  použitém pro programování senzorových uzlů, o jazyce nesC a o inteligentních agentech. Dále se seznámí s agentním jazykem ALLL. Jeho sémantika je vysvětlena na názorných příkladech. Bude popsán návrh platformy a konkrétní implementace interpretu jazyka ALLL. Na závěr nesmí chybět příklad kódů agenta s detailním popisem činnosti.



## Abstract

The aim of this work is to describe platform's concept and interpreter's implementation for mobile agents in wireless sensor networks. The reader will meet with theoretic and practical respect – he will obtain information about simple operating system TinyOS used for Motes' programming, about nesC language and about intelligent agents. Then he will meet with agents' language ALLL. Its semantic is illustrated on clear examples. The platform's concept and the concrete ALLL interpreter implementation will be described. Some examples with detailed description mustn't be missing at the end.



## Klíčová slova

Agent, bezdrátové senzorové sítě, TinyOS, nesC, ALLL, interpret, zásobníkový automat, MicaZ, platforma

## Keywords

Agent, wireless sensor networks, TinyOS, nesC, ALLL, interpreter, stack automaton, MicaZ, platform

## Citace

Spáčil Pavel: Mobilní agenti v bezdrátových senzorových sítích, bakalářská práce, Brno, FIT VUT v Brně, 2009

# Mobilní agenti v bezdrátových senzorových sítích

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, PhD.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Spáčil
18. května 2009

## Poděkování

Tímto bych chtěl poděkovat Ing. Františkovi Zbořilovi, Ph.D. za výborné vedení během tvorby bakalářské práce, ochotu poradit a účastnit se konstruktivního dialogu při vzniklých problémech. Dále pak kolegovi Bc. Janu Horáčkovi za dokonalou spolupráci na společném projektu, mé přítelkyni za její neustálou podporu a všem ostatním lidem, kteří mi pomáhali.

© Pavel Spáčil, 2009
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah


1 Úvod ..... 2
2 Bezdrátové senzorové sítě ..... 3
  2.1 Historie ..... 3
  2.2 Využití ..... 3
  2.3 Problémy ..... 3
3 Senzorový uzel MICAz ..... 4
4 TinyOS ..... 5
  4.1 Hardwarová abstrakce ..... 5
  4.2 Rádiová komunikace ..... 6
  4.3 Limity TinyOS ..... 7
5 Jazyk nesC ..... 8
6 Agent ..... 10
7 Návrh platformy pro agenta ..... 11
8 Jazyk ALLL (Agent low level language) ..... 13
  8.1 Změna syntaxe jazyka ..... 13
  8.2 Abstraktní datové struktury pro interpretaci ALLL ..... 13
  8.3 Unifikace ..... 14
  8.4 Substituce registrů ..... 14
  8.5 Maturace registrů ..... 14
  8.6 Sémantika jednotlivých akcí ..... 15
9 Interpretační cyklus ..... 19
10 Průběh implementace ..... 23
  10.1 ArrayC ..... 23
  10.2 ArrayOpC ..... 24
  10.3 StackC ..... 24
  10.4 TableC ..... 24
  10.5 AgentC ..... 25
  10.6 PlatformSvcC ..... 25
11 Příklady kódů agenta ..... 27
  11.1 Blikání LED na Mote ..... 27
  11.2 Vzdálené blikaní mezi dvěma Mote (rádiová komunikace) ..... 29
12 Závěr ..... 31
Literatura ..... 32

# 1 Úvod

Popularita bezdrátových senzorových sítí v posledních letech roste a jejich využití se z průmyslové sféry přesouvá i do domácností. Malé senzory tvořící bezdrátovou sít bývají řízeny pevně daným kódem bez možnosti změny za běhu sítě. Použití inteligentních agentů v těchto senzorech může rozšířit schopnosti celé sítě a zvýšit její flexibilitu.

Účelem této práce je seznámit čtenáře s jedním z mnoha možných řešení použití agentů v bezdrátových senzorových sítích. Popsat vytvoření platformy s interpretem pro mobilní agenty na senzorových uzlech jak po stránce teoretické, tak i praktické, doplněné názornými příklady.

V první kapitole se čtenář seznámí s historií a současným využitím senzorových sítí včetně možných problémů. Další kapitoly popisují konkrétní senzorový uzel a prostředky pro jeho programování – jazyk nesC a jednoduchý operační systém TinyOS. Poté jsou popsáni agenti včetně návrhu platformy pro jejich interpretaci. Následující kapitola se zabývá jazykem pro programování agentů včetně názorných příkladů. Jsou zde zmíněny také služby naimplementované platformy. Dále je rozebrán interpretační cyklus zahrnující popis funkcí jednotlivých akcí pseudokódem a průběh implementace s detailním zaměřením na komponenty interpretu. V poslední části práce jsou vysvětleny dva ukázkové kódy agentů, které byly použity při vývoji platformy a interpretu.

# 2 Bezdrátové senzorové sítě

## 2.1 Historie

Počátky této technologie jsou spojeny s vojenským výzkumem během studené války. Příkladem může být systém SOSUS [8] nebo IUSS [7]. SOSUS byl vojenský program z padesátých let dvacátého století, jehož účelem bylo sledování ponorek. Systém byl tvořen nezávislými stanicemi, které byly umístěné pod mořskou hladinou a zachytávaly hluk plujících ponorek. Následníkem byl systém IUSS, který využíval modernější technologie, například mobilní stanice. Snahou bylo vytvořit sít malých stanic, které měřily požadované veličiny a navzájem mezi sebou komunikovaly. Postupem času se z velkých stanic staly malé senzory ovládané mikroprocesory, které měly schopnost měřit různé veličiny a komunikovat mezi sebou. Tyto senzory se seskupují do ad-hoc sítí a mohou sledovat rozsáhlé oblasti.

Za dalším vývojem a hlavně rozšířením mobilních senzorů stojí kalifornská univerzita Berkeley, na které vznikly senzorové uzly zvané MICA [5] a jednoduchý open-source operační systém TinyOS [11], který je založený na událostech a upravené verzi jazyka C. Také společnost Infineon, která se zabývá vývojem a výrobou polovodičových součástek, vyvinula řadu senzorových uzlů zvaných eye-IFX, které jsou podporované V TinyOS. Stejně tak i senzorové uzly TinyNode společnosti ShockFish. Všechny tyto zařízení mají několik společných charakteristik: malá velikost, rádiový modul, velice nízká spotřeba a možnost rozšířit je o různé senzory.

## 2.2 Využití

Využití senzorových sítí se neustále rozšiřuje. Původně byly tyto sítě určeny pro sledování určitého jevu na velkých plochách nebo pro dlouhodobé sledování životního prostředí. V dnešní době se uzly používají pro sledování objektů, řízení jaderných reaktorů, detekci požáru nebo sledování automobilového provozu. Při změně sledované veličiny mohou uzly buď samý vykonat nějakou akci, nebo jen odešlou zprávu do sítě o nastalé situaci. Síť je vždy tvořena velkým množstvím senzorových uzlů zvaných obvykle „Mote“, které mimo procesor a napájení obsahují specifickou množinu senzorů pro danou aplikaci, a několika speciálními jednotkami, kterým se říká basestation nebo „data acquisition unit“. Ty sbírají data ze sítě a předávají je dál pro zpracování. Celá taková síť je ve většině případů spojena bezdrátově.

## 2.3 Problémy

Jelikož jsou uzly rozmístěny na velké ploše a jsou napájeny bateriemi, je kladen důraz na nízkou spotřebu jednotlivých jednotek. V určitých prostředích lze použít solární články pro dobíjení baterií. Problémem je také rádiová komunikace. Musí existovat způsob jak z každého uzlu sítě odeslat zprávu do bázového uzlu (většinou je připojen k počítači a slouží pro sběr dat ze sítě) a naopak, případně jinému uzlu. Nelze také zanedbat rušení a možné kolize při souběžném vysílání více blízko položených uzlů.

# 3 Senzorový uzel MICAz

Tento typ uzlu je vylepšenou verzí původního uzlu MICA vyvíjeného na univerzitě Berkeley. Při vývoji uzlů je kladen důraz na nízkou spotřebu, malé rozměry, snadnou rozšiřitelnost a robustnost. Uzel je složen ze tří hlavních částí: deska s procesorem a rádiovým modulem, senzorová deska a baterie. Uzly, na kterých probíhalo programování bakalářské práce, jsou složeny z procesorové desky MPR2400CA a senzorové desky MTS400CA a bateriového pouzdra pro dvě AA baterie. Procesorová deska obsahuje nízkonapěťový procesor Atmel ATmega128L [9], který má 128kB flash paměti, 4kB SRAM paměti a rádiový modul pro komunikaci s okolím. Na senzorové desce je několik čidel fyzikálních veličin, například čidlo teploty, vlhkosti nebo okolního osvětlení. Další informace o procesorové a senzorové desce lze nalézt v [2] a [3].

**Obrázek 3.1 - senzorový uzel MICAz**

# 4 TinyOS

Pro programování uzlů vytvořili vědci z UC Berkeley jednoduchý operační systém zvaný TinyOs [4] (zkráceně TOS). Je programován v dialektu jazyka C zvaném nesC [12] a je založen na událostech. Není to tudíž pravý operační systém, jak jej známe například ze stolních počítačů. První verze byly složené z kódu psaného v jazyce C a skriptů v jazyce Perl. Verze 1.0 vytvořená v roce 2002 již byla programována v nově vytvořeném jazyce nesC.

Od počátku je TOS open-source a může se tak šířit mezi velké množství vývojářů a uživatelů. Aktuální verze 2.1.0, která byla zveřejněna v srpnu roku 2008, podporuje velké množství zařízení (mica, micaz, telosb,eyeIFX, …) a je výrazně odlišná od předchozí verze 1, jejíž vývoj byl ukončen na konci roku 2005. Změny se týkají hlavně rádiové komunikace, spouštění systému a zdrojové kódy nejsou vzájemně kompatibilní. Je však možnost, jak zdrojové kódy převést na novou verzi, díky odlišnostem systémů ale není převod dokonalý a přesný.

## 4.1 Hardwarová abstrakce

TOS poskytuje základní prostředky pro platformě nezávislé řízení jednotlivých částí uzlů. Tím je například měření dat ze senzorů, komunikace přes rádio nebo sériovou linku (je-li modul součástí basestation), řízení diagnostických LED nebo ovládání permanentního úložiště. Konkrétní ovládání hardwaru záleží na použitém zařízení. Hardwarová abstrakce (HAA – Hardware Abstraction Architecture), která je znázorněna v následující tabulce, je v TOS složena ze tří vrstev [4]. Účelem je sjednotit návrh a programování obsluhy různorodého hardware a napojení takto vytvořených modulů do systému a aplikací.

| **HIL** – Hardware independent layer |
|---|
| **HAL** – Hardware abstraction layer |
| **HPL** – Hardware presentation layer |

**Tabulka 4.1 – architektura HAA**

Nejnižší vrstva **HPL** realizuje přímou kontrolu konkrétního hardware platformy. Hlavní úlohou této vrstvy je prezentovat schopnosti hardware pomocí konceptů TOS. Přistupuje k hardware přes paměť nebo přes porty mapované do adresového prostoru paměti, v opačném směru zpracovává přerušení od hardware. Funkce v tomto rozhraní by neměly udržovat žádný vnitřní stav. Jako první vrstva abstrakce zjednodušuje přístup k HW skrze rozhraní, jež obsahuje jednoduché funkce:

- Inicializace, spouštění a zastavování hardwarových modulů pro efektivní řízení spotřeby
- „get“ a „set“ příkazy pro registry, které přímo řídí HW
- Příkazy pro zjišťování a nastavování nejpoužívanějších příznaků v HW
- Povolování a zakazování přerušení
- Obslužné funkce pro zpracování přerušení

Funkce na této vrstvě vykonávají pouze jednoduché, časově kritické operace a přenechávají zbytek zpracování na vyšší vrstvy. Účelem je pouze zjednodušení ovládání konkrétního hardware.

Prostřední vrstva **HAL** představuje jádro architektury. Využívá rozhraní vrstvy HPL pro vytvoření další úrovně abstrakce a skrývá složitost přímého přístupu k hardware. Účelem této vrstvy je

odhalit specifické vlastnosti konkrétní platformy pomocí nejlepší možné abstrakce a zpřístupnit je pro tvorbu aplikací. Naproti funkcím z předchozí vrstvy můžou funkce udržovat vnitřní stav pro řízení a správu zdrojů.

Nejvyšší vrstva **HIL** vytváří rozhraní, která nejsou závislá na konkrétní platformě, přitom využívá specifických rozhraní vrstvy HAL. Tato vrstva poskytuje úplnou abstrakci nad hardware, která usnadňuje vývoj aplikací, pomocí skrývání různých odlišností platforem.

Dále obsahuje např. asynchronní řízení nebo ovládání úloh, v poslední verzi je zahrnuta i podpora pro vlákna, jak jsou známé z plnohodnotných operačních systémů. Pro každou platformu jsou k dispozici i hlavičkové soubory běžných funkcí z jazyka C, jako je například práce s řetězci/pamětí, znakové funkce apod. Abstrakce procesoru je zajištěna překladačem (GCC), a systém tak nemusí například řešit vytváření spouštěcího kódu, inicializaci globálního zásobníku nebo tabulky přerušení. Pro přístup k některým speciálním službám procesorů je proto nutný nízko-úrovňový přístup.

## 4.2 Rádiová komunikace

Rádiová komunikace mezi uzly pracuje na technologii ZigBee [1], která je postavena na standardu IEEE 802.15.4. Stejně jako Bluetooth patří do kategorie bezdrátových osobních sítí (Wireless PAN). Hlavní přednosti této technologie je spolehlivost, jednoduchá a nenáročná implementace, nízká spotřeba komunikačních zařízení a také nízká cena. Síť pracuje v několika bezlicenčních pásmech a přenosová rychlost dosahuje několika set kilobitů za sekundu. Jelikož v síti není žádný hlavní řídící prvek a všechny uzly jsou rovnocenné, probíhá směrování ad-hoc, tj. řídícím prvkem se stane některý z uzlů a v případě, že tento uzel vypadne, stane se po krátké době řídícím prvkem jiný uzel. Tato technologie je určena hlavně do průmyslového nasazení, kde by použití Bluetooth nebylo příliš vhodné a kde nejsou požadovány přenosy velkých objemů dat. Implementace protokolů je díky nutnosti nasazení v málo výkonných procesorech velice jednoduchá. Dosah ZigBee je závislý na okolních podmínkách a může dosáhnout až 50 metrů.

Protokol je složen ze tří základních vrstev: fyzická a linková vrstva podle standardu IEEE 802.15.4, síťová vrstva a aplikační vrstva. Fyzická a MAC (Media Access Control) vrstva definuje několik rádiových pásem pro použití v různých zemích, hlavní skupiny jsou: globální, Amerika a Austrálie, Evropa. Datový signál je modulován metodou O-QPSK (Offset Quadrate Phase-Shift Keying), což je digitální modulace signálu, která moduluje fázi vstupního signálu pro přenos po fyzickém médiu, a je přenášen prostřednictvím DSSS (Direct Sequence Spread Spectrum), který kóduje každý přenášený bit do pseudonáhodných sekvencí bitů (tzv. chipů). Tím je zavedena redundance a zvýšena odolnost proti rušení a náhodnému odposlechu, protože signál má podobu náhodného šumu a je velice obtížné jej bez znalostí mechanismu vytváření pseudonáhodné sekvence rozkódovat. Stejně jako v jiných bezdrátových technologiích je pro přístup k fyzickému médiu (vzduch) použita metoda CSMA/CA (Carrier Sence Multiple Access with Collision Avoidance). Chce-li nějaké zařízení odeslat svoji zprávu, musí nejdříve „poslouchat," zdali nevysílá už někdo jiný, a případně počkat. Až je médium volné, může svoji zprávu odeslat. To slouží k zabránění vzniku kolize.

MAC vrstva (linková) definuje 4 druhy přenášených rámců: datové, potvrzovací, beacon a nastavovací/řídící rámce. Beacon rámce slouží pro uspání určených zařízení na zvolenou dobu, ta může být od 15ms až do zhruba 15 minut. Potvrzovací rámce slouží pro potvrzení komunikace na této vrstvě. Standard také definuje tři síťové topologie: hvězda, strom a síť. Liší se hlavně dosahem řídícího uzlu a vzájemným propojením ostatních uzlů. Adresování v sítí je založeno na 64 bitových adresách (případně ve zkrácené podobě jen 16 bitů) a 16 bitových identifikátorech sítí. Datagramy jsou šifrovány metodou AES o délce 128 bitů a v případě potřeby je možné zapnout šifrování i na linkové vrstvě. To však zvyšuje nároky na výpočetní schopnosti uzlů. Síťová vrstva zajišťuje řízení komunikace,

zabezpečení a směrování. Aplikační vrstva je zodpovědná za párování zařízení, jejich vyhledávání a volbu zabezpečení.

## 4.3 Limity TinyOS

Jako každý systém, i TinyOS má svoje omezení [6]. Praxe ukázala tři hlavní limity tohoto systému: podpora nových platforem, konstrukce aplikací a spolehlivý provoz. Přidání podpory pro novou platformu se většinou provádí zkopírováním již existující platformy a upravením odlišných částí. Přestože TinyOS definuje jistý způsob hardwarové abstrakce, její hranice nejsou pevně dány a může se stát, že komponenty přistupují přímo k hardware. Pokud by taková implementace byla použita v jiné komponentě, musí se dát pozor na to, zda není využitý hardware použit na jiném místě. Návrh aplikací založený na velkém počtu komponent může vést k nepředvídatelným problémům díky závislostem a vzájemným interakcím. Více komponent může využívat stejný hardwarový prostředek a při souběžném použití dojde k selhání jedné z komponent.

Nejvíce problémů je v obsluze rádia, ať už při vysílání nebo při příjmu. Při příjmu paketu ho sice řídící program rádia úspěšně přijme a odešle potvrzení odesílateli, ale nemusí se podařit zařadit úlohu do fronty, v takovém případě je paket zahozen bez informování odesílatele. Horší situace může nastat při odesílání paketu, zásobník úloh může být před vložením úlohy potvrzení příjmu plný a program, který na takové potvrzení čeká, se může zaseknout. V takovém případě je úloha zavolána přímo pomocí přerušení. I toto řešení není dokonalé, protože může způsobit problémy v programu, který využívá jen úlohy. Použití úloh všeobecně může způsobit problémy a zablokovat celý program.

# 5 Jazyk nesC

Tento speciálně vytvořený programovací jazyk je založen na klasickém jazyce C, ale obsahuje některé vlastnosti modulárního přístupu. Je přizpůsoben pro potřeby TinyOS a pro bezdrátové senzorové uzly s omezenými systémovými prostředky, mezi ně patří hlavně malá velikost programové a RAM paměti a omezený zdroj energie. Program v nesC je složen z několika vzájemně propojených komponent, které využívají a poskytují funkce jiným komponentám v TinyOS. Komponenty jsou podobné objektům z OOP, mají svůj lokální jmenný prostor, obsahují lokální funkce a některé z nich mohou poskytovat přes rozhraní jiným komponentám.

Každá komponenta může využívat rozhraní jiných komponent a poskytovat své rozhraní jiným komponentám. NesC má tři typy souborů: rozhraní, modul a konfigurace.

Soubor definující rozhraní obsahuje seznam příkazů a událostí. Modul může tato rozhraní využívat nebo poskytovat. Z využívaných rozhraní může volat příkazy rozhraní a musí implementovat všechny události, s poskytovanými rozhraními je to přesně naopak. Konfigurace definuje vzájemné spojení komponent přes jejich rozhraní, přičemž vztah nemusí být vždy 1:1. Více modulů může využívat rozhraní jedné komponenty. Každý modul je nutné před použitím vytvořit, podobně jako v Javě je nutné vytvořit instanci objektu, těch může být více a lze je parametrizovat.

Pro ovládání komponent hardwaru se používá neblokující ovládání zvané „split-phase.“ Místo aby se například při odesílání zprávy zavolala funkce, která provede celý proces, a blokovala tím provádění dalšího kódu, zavolá se pouze příkaz pro zahájení odesílání. Program může poté normálně pokračovat a až je požadovaná operace dokončena, je vyvolána příslušná událost. Během provádění operace je zakázané manipulovat s pamětí, kterou daná operace používá.

**Příklad:** pro odesílání zprávy přes rádio slouží funkce send, jejím parametrem je ukazatel na blok paměti, který se má odeslat. Po dokončení odesílání je vyvolána událost sendDone, která mimo jiné znamená, že je opět možné pracovat s blokem paměti pro odeslání.

**Ukázka:** tři typy souborů v nesC a jejich struktura. Nejdříve soubor definující rozhraní, ten obsahuje pouze blok `interface` s výčtem příkazů a událostí.

Příklad rozhraní `Send`, které mimo jiné definuje příkaz `send` a událost `sendDone`:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);
  ...
}
```

Modul obsahuje dvě sekce: `module` a `implementation`. První definuje, která rozhraní modul používá a poskytuje, druhá sekce obsahuje vlastní kód modulu, tj. implementaci příkazů, událostí a případně lokální funkce, které nejsou přístupné z jiných modulů.

Příklad modulu, který poskytuje rozhraní `TableI` (musí implementovat všechny jeho příkazy) a využívá dvě rozhraní `ArrayOpI` (přejmenované na `PoleOp`) a `LedkyI`. V sekci implementace je jedna lokální funkce a příkaz z využívaného rozhraní:

```
module TableC {
  provides {
     interface TableI;
```

```
  }
  uses {
    interface ArrayOpI as PoleOp;
    interface LedkyI;
  }
}
implementation {
  uint16_t getStart(int8_t table_id) {
  ...
  }
  command bool TableI.empty(int8_t table_id) {
  ...
  }
}
```

Konfigurace slouží pro vzájemné propojení komponent či modulů, vytvoření instancí modulů a případné napojení jejich rozhraní na rozhraní komponenty. Soubor obsahuje opět dvě sekce: `configuration` a `implementation`. V první sekci jsou definována rozhraní, které komponenta využívá a poskytuje, v druhé je pak vytvoření modulů a jejich vzájemné propojení.

Komponenta `MobileAgentAppC` využívá a poskytuje nějaká rozhraní, která jsou definována v první sekci. Ve druhé sekci jsou moduly vytvořeny, případně přejmenovány a jsou také napojena jejich rozhraní. Šipka doprava znamená, že komponenta vlevo využívá rozhraní, které komponenta vpravo poskytuje, je-li šipka naopak, je význam obrácený. Rovnost znamená, že rozhraní komponenty je přímo napojeno na rozhraní vnitřního modulu:

```
configuration MobileAgentAppC
{
  uses {
    interface ControlI;
    ...
  }
  provides {
    interface TableI;
    ...
  }
}
implementation {
  components ArrayC as Pole, ArrayOpC as OperacePole,
             StackC as Zasobnik, TableC as Tabulka,
             ...

  OperacePole.PoleCntrl -> Pole.StdControl;
  OperacePole.Pole -> Pole.ArrayI;

  Zasobnik.Pole -> OperacePole.ArrayOpI;
  Sluzby.WaitI = WaitI;
  ...
}
```

# 6 Agent

Pod pojmem agent [10] si lze představit například osobu, která vykonává nějakou činnost bez vnějšího řízení a ve prospěch svého klienta. Obecně lze říci, že agentem se rozumí nějaký autonomní prvek v systému, který se řídí plánem s určitým cílem a je ovlivňován, ne však řízen, vnějšími změnami. Liší se tak od různých pasivních prvků právě tím, že jedná samostatně. V textu se pojmem agent myslí umělý agent – člověkem vytvořené dílo, které má podobné charakteristiky jako agent živý.

Agent a jeho chování bývá řízeno programem, který je vykonáván na nějaké architektuře. Umělý agent je tedy spojení programu a architektury. Jeho chování může být ovlivňováno vnějšími podněty, ale může i ovlivňovat okolní prostředí. Agentní systém je tvořen větším počtem agentů a prostředím, které je obklopuje. Nejjednodušší agenti jsou tzv. reaktivní, ti pouze reagují na změny prostředí a neuchovávají si žádný jeho model. Složitější jsou deliberativní agenti, kteří se mimo reakce na okolní změny snaží svým jednáním prostředí ovlivňovat, získat určité výhody a dosáhnout tak svých cílů. Tato vlastnost se nazývá proaktivita. Tito agenti mají své mentální stavy, což jsou aktuální podoby jejich přesvědčení, přání a záměrů. Agenti mají také jisté sociální schopnosti známé z lidského prostředí, například vzájemná komunikace agentů, řešení konfliktů nebo spolupráce na dosažení kýženého cíle. Agenti v jazyce ALLL využívají těchto konceptů, ale jazyk je přiblížen více hardwaru.

# 7 Návrh platformy pro agenta

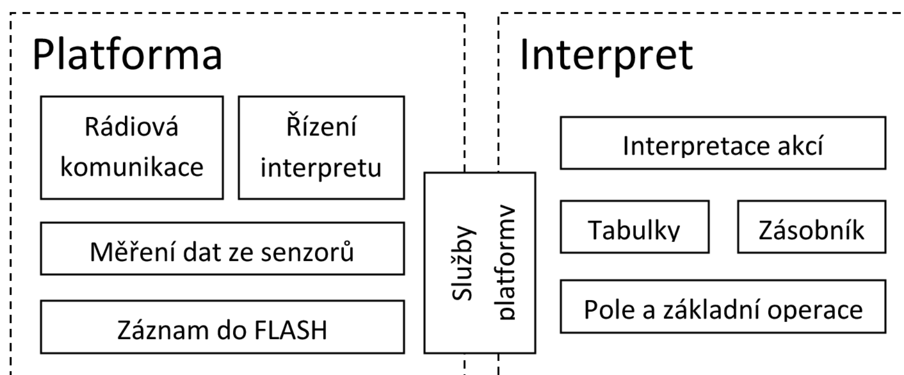
Účelem platformy je vytvořit vhodné prostředí pro činnost agentů. Musí umět řídit činnost agenta a poskytovat mu vhodné prostředky pro danou situaci – například rádiovou komunikaci, mobilitu, měření dat ze senzorů nebo ovládání akčních členů. Platforma tak vytváří prostředníka mezi systémem (hardware a jeho operační systém) a agentem. Jelikož je pro naši platformu použit senzorový uzel s omezenými prostředky (malá paměť DRAM) a operační systém TinyOS, je nutné těmto podmínkám přizpůsobit i její návrh.

Celá platforma je rozdělena na dvě hlavní části: platformu a interpret kódu. Obě části jsou v oddělených komponentách a jsou navzájem propojeny sadou rozhraní.

Platforma řídí činnost interpretu a poskytuje mu abstraktní funkce pro řízení systémových modulů. Řízení lze rozdělit na tři části: inicializace interpretu, provedení jednoho kroku a informování o dokončení vnější činnosti nebo změně okolí. Inicializace zajišťuje nastavení interpretu po spuštění nebo přijmu agenta, aby byl schopen provádět svou činnost. Provedení jednoho kroku spočívá v předání řízení interpretu, který provede první akci v plánu a podle výsledku informuje platformu o dalším průběhu – pokračování interpretace nebo pozastavení a čekání na vnější událost. Informování zahrnuje opětovné spuštění interpretace po vypršení časovače, příjmu zprávy přes rádio nebo dokončení měření dat ze senzorů. Platforma dále poskytuje interpretu určitou sadu služeb, které může agent využívat nejen pro řízení okolí. Služby zahrnují odesílání zpráv a agentů na jinou platformu, pozastavení činnosti na určitý čas, ovládání LED, měření dat ze senzorů nebo množinu algoritmů (aritmetika, práce se seznamy). Implementace platformy je prací kolegy Bc. Jana Horáčka, pro podrobnější informace proto doporučuji nahlédnout do jeho práce.

Interpret vykonává kód agenta v jazyce ALLL a pracuje na principu zásobníkového automatu. Plán agenta je tedy uložen na zásobníku po jednotlivých akcích. Další částí jsou tabulky, které všechny mají stejnou strukturu a operace, ale význam jednotlivých tabulek je různý. Tyto abstraktní datové struktury jsou vytvořeny nad polem s kódem agenta a základních funkcí pro práci s ním. Uvedené části dohromady tvoří třívrstevnou architekturu interpretu. Během provádění akce může interpret volat služby platformy, v každém kroku ale maximálně jednu, jinak by mohlo dojít k problémům se synchronizací nebo k přepsání některé části pole.

Na senzorových uzlech neexistují téměř žádné prostředky pro ladění aplikací. Proto je nutné maximálně eliminovat počet možných chyb, které jsou způsobeny chybným návrhem, při běhu programu, neboť by to mohlo mít vážné následky.




**Obrázek 7.1 - struktura platformy a interpretu**

DRAM

| Proměnné funkcí | Pole s kódem: planBase \| plan \| beliefBase \| inputBase \| reg1 \| reg2 \| reg3 |
| --- | --- |

| Vstupní a výstupní buffer pro rádiovou komunikaci | Indexy částí pole |
| --- | --- |

FLASH

| Kód interpretu a platformy | Počáteční kód agenta | Log pro data ze senzorů |
| --- | --- | --- |

**Obrázek 7.2 - struktura paměti platformy**

# 8 Jazyk ALLL (Agent low level language)

Přestože existuje několik agentních jazyků a kalkulů, byl vyvinut na FIT jazyk nový [13], který přidává navíc některé užitečné vlastnosti. Mezi ně patří hlavně obsluha výjimek, meta-rozhodování a klonování. Agent je tak schopný se vypořádat s nečekanými situacemi a chybami během vykonávání plánu.

Jako každý jazyk je i ALLL složen z vět, které reprezentují jednotlivé plány agenta. Plány jsou opět věty složené z jednotlivých akcí. Jazyk umožňuje hierarchické zanořování plánů, takže nezdaří-li se provedení nějaké akce v podplánu, je tento podplán smazán a provádění pokračuje v plánu na vyšší úrovni. Jazyk obsahuje několik druhů akcí: komunikace přes rádio, operace pro práci s bází znalostí, spouštění plánů a volání služeb platformy.

Komunikace zahrnuje odesílání zpráv na daný uzel a vyzvednutí přijatých zpráv ze vstupní báze. Pro práci s bází znalostí jsou zde akce pro přidávání, odebírání a pro test na nějakou n-tici. Plány lze spouštět buď přímo – akce podplánu jsou přímo parametrem této akce, nebo nepřímo – pojmenované plány jsou uloženy v bázi plánů a hledá se v nich požadovaný plán. Volání služeb platformy je akce, které se předá jméno služby, parametry a platforma tuto akci provede, neboť agent nemá schopnost takové služby přímo provádět.

## 8.1 Změna syntaxe jazyka

Během implementace došlo k výraznému zjednodušení a úpravě syntaxe jazyka ALLL. Byly odstraněny některé závorky okolo plánů, čárky mezi akcemi a upraveny jednotlivé příkazy. Úprava má za následek zjednodušení kódu a hlavně jeho zkrácení, neboť paměť pro kód agenta je relativně malá. Nová BNF upraveného jazyka je uvedena v Příloze 2.

## 8.2 Abstraktní datové struktury pro interpretaci ALLL

V interpretu jazyka ALLL se používají dva hlavní typy datových struktur: tabulka a zásobník.

Tabulka je tvořena seznamy, které nejsou nijak uspořádány. Tabulek je v interpretu několik, každá obsahuje seznamy s odlišným významem. Báze plánů (planBase) obsahuje pojmenované plány, jež je možné provést (vložit na zásobník) během interpretačního cyklu. Báze vstupu (inputBase) obsahuje n-tice obdržené od platformy, například přijaté zprávy z rádia nebo naměřená data ze senzorů. V bázi znalostí (BeliefBase) si agent ukládá konkrétní hodnoty během své činnosti. Poslední částí jsou 3 registry (regx), používají se pro dočasné uložení seznamů (n-tic) a výsledků některých služeb platformy. Do tabulek je možné vkládat seznamy, mazat je a unifikovat je s jiným seznamem.

Zásobník je využit pro plán, jeho prvky představují akce, které se mají provést. Lze vkládat prvky na vrchol zásobníku, zjistit, jaký prvek je na vrcholu, a po provedení akce jej smazat.

## 8.3 Unifikace

Při prohledávání tabulek se využívá operace unifikace. Jedná se v podstatě o hledání stejné n-tice v tabulce jako je ta zadaná. V n-tici podle níž se má unifikovat může být tzv. anonymní proměnná (symbol podtržítka), který reprezentuje libovolný prvek – posloupnost znaků nebo další n-tici. V takovém případě jsou výsledkem všechny n-tice, ve kterých byl za podtržítko nahrazen libovolný prvek.

Jelikož nejsou v jazyce klasické proměnné, je tato operace značně zjednodušena a pracuje prakticky na pouhém hledání dvou stejných seznamů.

## 8.4 Substituce registrů

V každé akci může být místo konkrétní hodnoty zástupný symbol registru, který se před prováděním akce nahradí aktuálním obsahem zadaného registru. Toto chování však není nezbytně nutné, jelikož v hlavní komponentě agenta a v komponentě, která provádí služby platformy, je přístupné pole a jednotlivé indexy částí, tak se nemusí žádné nahrazování provádět a lze pracovat přímo s registrem, na který se akce odkazuje. Registr může být přímo parametr akce/služby nebo součástí složitější n-tice.

Při volání některých služeb se počítá pouze s tím, že registr je přímo parametr služby, toto platí pro ovládání LED a čekání. Při provádění těchto služeb se kontroluje první znak parametru, při nalezení symbolu registru se parametrem stane zadaný registr. V jiných případech se substituce registrů provádí podle požadované akce.

Při vkládání do báze znalostí se nejdříve vloží n-tice tak, jak je v akci, a až poté dojde k nahrazení zástupných symbolů registru. Samozřejmě se i během nahrazování kontroluje velikost kódu, aby nedošlo k překročení velikosti pole, v takovém případě se smaže celá vkládaná n-tice a akce končí chybou.

Při operaci unifikace je postup podobný, jakmile se při porovnávání znaků narazí na znak registru (&), tak se podle následujícího čísla registru změní index „zdroje“ a porovnávání pokračuje do konce daného registru, poté se vrátí index „zdroje“ na původní hodnotu za číslem registru a pokračuje se v porovnávání.

## 8.5 Maturace registrů

Jak bylo předesláno v minulé kapitole, může se v akci plánu vyskytnout symbol registru místo parametru. Za tímto symbolem následuje číslo registru, který se má použít, a také znak určující úroveň symbolu. Před každým krokem interpretace se kontroluje celý plán, zdali neobsahuje nějaký symbol registru s nulovou úrovní, v takovém případě dojde k jeho nahrazení aktuální hodnotou ještě před provedením akce. Tato úroveň se snižuje při zanořování plánů, tj. při (přímém nebo nepřímém) vyvolání plánu dojde ke snížení úrovně všech symbolů registru. Při ukládání plánu do báze plánů se úroveň zvyšuje. Počáteční úroveň symbolů může být různá a umožňuje tak vytvářet programové konstrukce, které by bez použití maturace nebyly možné. Tato funkce nebyla zatím do interpretu zapracována.

## 8.6 Sémantika jednotlivých akcí

Jazyk ALLL je založen na několika jednoduchých akcích, jejich seznam a význam je uveden v následující tabulce. Akce mohou v parametrech obsahovat zástupné znaky registrů, v takovém případě dojde během provádění akce k jejich nahrazení.

| Kód akce | Parametry | Význam |
|---|---|---|
| + | n-tice \| registr | Přidání do BeliefBase, unifikací se kontroluje duplicitní vkládání |
| − | n-tice \|registr | Odebrání položek z BeliefBase pomocí unifikace |
| ! | číslo \| registr<br>n-tice \| registr | Odeslání zprávy zadané n-ticí nebo registrem na mote se zadanou adresou |
| ? | číslo \| registr | Test InputBase na zprávu od mote/senzoru se zadanou adresou |
| @ | seznam akcí | Přímé spuštění, akce se vloží na zásobník se zarážkou |
| ^ | jméno \| registr | Nepřímé spuštění, hledá se plán v PlanBase se stejným jménem |
| & | číslo | Změna aktivního registru |
| * | n-tice \| registr | Test BeliefBase na zadanou n-tici nebo registr, výsledek se uloží do aktivního registru |
| $ | písmeno<br>{n-tice \| registr} | Volání služeb platformy, první parametr (písmeno) je kód operace, druhý parametr jsou parametry služby, nemusí být u všech služeb |
| # | žádné | Zarážka za plánem, sémanticky tato akce nemá žádný význam |

**Tabulka 8.1 - přehled akcí, jejich parametry a význam**

### 8.6.1 Přidání do báze znalostí

Akce přidání n-tice do báze znalostí nejdříve unifikuje zadanou n-tici nebo registr s tabulkou a pokud vkládaná n-tice v tabulce nebyla nalezena, je vložena na začátek tabulky. K úspěšnému vložení ale nemusí dojít vždy, pole pro kód agenta může být zaplněno a vkládaná n-tice by způsobila přetečení pole. V takovém případě akce končí chybou.

Příklady, v prvním registru je n-tice `(456)`:

`+(123,456)` Akce vloží n-tici `(123,456)` na začátek báze znalostí, opětovné volání nebude mít žádný význam, báze znalostí už takovou n-tici obsahuje.

`+(123,&1)` Nejdříve se vloží do báze znalostí n-tice `(123,&1)` a poté dojde k nahrazení symbolu registru za jeho aktuální obsah, v bázi znalostí bude po nahrazení n-tice `(123,(456))`

`+&1` Do báze znalosti se opět nejdříve vloží `&1` a poté dojde k nahrazení za aktuální obsah registru. Vložená n-tice tedy bude `(456)`

### 8.6.2 Odebrání z báze znalostí

Akce odebrání unifikuje všechny n-tice v bázi znalostí se zadanou n-ticí nebo registrem a v případě shody je nalezená n-tice z tabulky smazána. Tato akce vždy končí úspěchem, i když nebyla zadaná n-tice v tabulce nalezena a nebylo tudíž nic smazáno.

Příklady, báze znalostí obsahuje n-tice `(123)(456)(123,456)`:

`-(123)` Akce unifikuje zadanou n-tici s každou n-ticí v tabulce a v případě shody je tato n-tice z tabulky odstraněna. Z tabulky bude tudíž odstraněna první n-tice.

`-(_)` Prvek n-tice je anonymní proměnná, při unifikaci bude shodná první a druhá n-tice a budou odstraněny. V tabulce zůstane tedy jen `(123,456)`.

`-(456,_)` Druhý prvek n-tice je opět anonymní proměnná, ale v tabulce není žádná n-tice, která by této vyhovovala a nebude tudíž nic smazáno.

## 8.6.3 Odeslání zprávy

Odeslání zprávy přes rádio patří mezi služby, které poskytuje platforma. Prvním parametrem je adresa cílové platformy a druhým je n-tice, která se má odeslat. Oba parametry mohou být zadány registry, cílová adresa však musí být pouze jednoprvková n-tice, jinak akce skončí chybou. Odesílaná zpráva je nejdříve zkopírována do výstupního bufferu, poté jsou nahrazeny symboly registrů a spuštěno odesílání zprávy.

Příklady, první registr obsahuje `(1)` a druhý `(123,456)`:

`!(1,(abc))` Akce odešle zprávu `(abc)` na uzel (platformu) s adresou 1.

`!(&1,&2)` Při provádění akce se nejdříve z prvního registru zjistí cílová adresa. Při kopírování zprávy do bufferu dojde k nahrazení symbolu registru za obsah druhého registru. Výsledkem je odeslání zprávy `(123,456)` na uzel s adresou 1.

## 8.6.4 Příjem zprávy

Jelikož jsou přijímané n-tice z rádia vkládány do vstupní báze automaticky, slouží akce příjmu jako test tabulky na n-tici z uzlu s požadovanou adresou, která je jediným parametrem této akce, ten může být opět zadán i registrem. Prvky ve vstupní bázi mají tvar (adresa,n-tice). Při nalezení prvního prvku s požadovanou adresou je n-tice přesunuta do aktivního registru a zbytek prvku smazán.

Příklady, vstupní báze obsahuje `(1,(abc))(2,(123))`, první registr obsahuje `(2)` a druhý (aktivní) je prázdný:

`?(1)` Ve vstupní bázi se hledá n-tice s prvním prvek `1`. N-tice je nalezena, druhý prvek `(abc)` je přesunut do aktivního registru a zbytek n-tice smazán, v bázi zůstane jen `(2,(123))` a ve druhém registru bude n-tice `(abc)`.

`?&1` Místo adresy uzlu je použit symbol registru, bude se tedy hledat podle obsahu registru. Druhý prvek n-tice je opět přesunut do aktivního registru a zbytek smazán. V druhém registru bude n-tice `(123)` a ve vstupní bázi zůstane první n-tice s adresou `1`.

`?(_)` Jako adresa je použita anonymní proměnná, ze vstupní báze bude tudíž vyjmuta první n-tice.

## 8.6.5 Přímé spuštění

Přímé spuštění může být použita pro rizikové plány, u kterých může dojít k selhání nějaké akce, a je nežádoucí, aby selhal celý plán o úroveň výš. Parametrem akce je pouze seznam akcí, jež se mají provést. Pokud není na zásobníku zarážka, vloží se na něj a před ní se postupně vkládají jednotlivé akce. Během vkládání může dojít k přetečení pole a akce končí neúspěchem.

Příklad, na zásobníku je za akcí kód `+(123)`:

`@(+(abc)!(2,(456)))` Na zásobník se vloží zarážka a obsah akce, zásobník bude tedy vypadat následovně:
`+(abc)!(2,(456))#+(123)`

### 8.6.6 Nepřímé spuštění

Nepřímé spuštění je podobné, parametrem akce nejsou další akce, které se mají vložit na zásobník, ale jméno plánu, jež se hledá v bázi plánů. Je-li plán nalezen, vloží se akce se zarážkou (pokud na zásobníku není) na zásobník, v opačném případě akce končí chybou.

Příklady, báze plánů obsahuje jeden plán `(plan,(+(def)!(3,(789)))`, na zásobníku je za akcí kód `+(123)` a první registr obsahuje `(plan)`:

`^(plan)` V bázi plánů se hledá plán se jménem „plan.“ Po nalezení jsou jeho akce vloženy na zásobník a ten bude vypadat následovně:
`+(def)!(3,(789))#+(123)`

`^&1` Parametrem akce je registr a jméno se bude hledat podle obsahu registru, jelikož registr obsahuje stejné jméno plánu jako v minulém příkladě, bude zásobník vypadat stejně.

### 8.6.7 Změna aktivního registru

Pro nastavení aktivního registru jakožto tabulky, kam se vkládají výsledky některých akcí, se používá jednoduchá akce, která má jediný parametr s číslem registru. Podle návrhu má agent tři registry, číslo tedy může být 1 až 3. Při změně registru dojde zároveň ke smazání jeho obsahu.

Příklad:

`&(1)` Akce nastaví první registr jako aktivní a smaže jeho obsah.

### 8.6.8 Testování báze znalostí

Při testování báze znalostí se využívá operace unifikace a n-tice, které jsou unifikovatelné se zadanou n-ticí, se zkopírují do aktivního registru jako seznam. Akce může skončit chybou, když není v poli místo pro vložení výsledného seznamu nebo když se unifikací nenajde žádná n-tice.

Příklady, v bázi znalostí jsou n-tice `(123)(456)(abc,def)`, první a zároveň aktivní registr je prázdný:

`*(123)` Unifikací zadané n-tice a báze se najde jedna n-tice, do registru se vloží jednoprvkový seznam s unifikovatelnou n-ticí, registr bude tedy obsahovat `((123))`.

`*(_)` Anonymní proměnná v akci zastupuje libovolný prvek v n-tici a v bázi znalostí jsou unifikovatelné dvě n-tice, jelikož unifikace probíhá od začátku tabulky a nalezené n-tice se vkládají vždy na začátek registru, bude pořadí n-tic opačné než v bázi znalostí, registr bude mít tedy následující obsah:
`((456),(123))`

`*(abc)` Unifikací se nenajde žádná n-tice, akce skončí chybou a aktivní registr zůstane prázdný.

### 8.6.9 Volání služeb platformy

Pro vykonávání služeb platformy, mezi něž patří například měření dat ze senzorů, ovládání LED nebo pozastavení interpretu, slouží speciální akce, která tyto služby spouští a ovládá. První parametr je kód

služby, jejich přehled je v následující tabulce. Za kódem může být v některých případech parametr služby, ten lze většinou i zadat registrem.

| Kód | Parametry | Popis |
|---|---|---|
| **a** | žádné | Pomocí této služby se aktivuje sledování příchozích zpráv při běžícím interpretu. |
| **f** | seznam \| registr | Služba vloží do aktivního registru první prvek seznamu jako jednoprvkovou n-tici nebo první ze seznamů, je-li jich je více. |
| **k** | žádné | Tato služba zastaví činnost interpretu. |
| **l** | seznam \| registr | Pomocí této služby se ovládají LED na Mote, parametr musí obsahovat kód barvy LED (r, g, y) a za ním může být stav LED (0 – nesvítí, 1 – svítí), není-li zadán, dojde k přepnutí stavu. |
| **m** | seznam \| registr | Parametr této služby obsahuje adresu (číslo) platformy, kam se má kód agenta přesunout, celý kód se zkopíruje do cílové platformy a provádění pokračuje na obou platformách dále, je-li cílová adresa stejná jako ta, kde agent momentálně je, provedením této služby se nic nestane. Za adresou může být parametr „s," který značí, že se má zastavit provádění kódu na zdrojové platformě. |
| **r** | seznam \| registr | Služba vloží do aktivního registru zbytek seznamu bez prvního prvku nebo všechny seznamy bez prvního z nich. Když je seznam pouze jeden a obsahuje jediný prvek, vloží se do akt. registru prázdná n-tice. |
| **s** | žádné | Služba zastaví provádění kódu, dokud nepřijde zpráva z rádia. Bylo-li aktivováno sledování a zpráva byla už přijata, provádění pokračuje dál. |
| **w** | seznam \| registr | Pomocí této služby lze pozastavit provádění kódu na zadaný čas v milisekundách. |
| **d** | žádné \| seznam \| registr | Zavoláním této služby bez parametru dojde ke změření aktuální teploty, v opačném případě je nutné zadat typ hodnoty (a – průměr, m – minimum, M – maximum) a počet hodnot, z kolika se má hodnota vypočítat. Když není naměřen dostatečný počet hodnot pro výpočet, končí služba chybou |

**Tabulka 8.2 - přehled služeb platformy**

V budoucnu je možné přidat do platformy další služby a tím rozšířit její schopnosti, uvedený seznam je pouze vzorek služeb pro prozatímní účely.

### 8.6.10 Zarážka

Poslední akce nemá žádný sémantický význam. Zarážka slouží pouze jako určení konce zanořeného plánu v případě jeho selhání, ze zásobníku jsou smazány všechny akce po tuto zarážku včetně.

# 9 Interpretační cyklus

Po startu platformy jsou spuštěny komponenty interpretu. Ve zdrojovém kódu interpretu může být připraven kód agenta, jehož plán je vložen na zásobník a tabulky jsou naplněny počátečními daty. Poté je spuštěna smyčka interpretace kódu. V opačném případě se platforma uspí a čeká, až je kód agenta přijat přes rádio. Procedura `booted` pro spuštění může být popsána takto:

```
Procedure booted;
  call Array.Init();
  if (agent_code) then // v programu je uložen kód agenta
  begin
    Stack.Push(plan);
    Table.Add(table_data);
    Platform.Run();
  end;
end;
```

Funkce `Init` z komponenty `Array` vymaže pole a vynuluje všechny indexy jednotlivých částí. Protože v paměti může být po spuštění náhodná kombinace bytů, je nutné toto udělat a zamezit tím neočekávanému chování interpretu. Funkce z komponenty `Platform` zde není nutné popisovat, jejich význam plyne z názvu. Funkce pro práci s tabulkou a zásobníkem jsou podrobněji popsány v následující kapitole.

Interpretační smyčka běží, dokud je na zásobníku nějaký prvek nebo nebyla interpretace pozastavena. Jeden krok spočívá ve vyzvednutí prvku z vrcholu zásobníku a provedení akce podle jejího kódu. Je-li výsledek akce kladný, prvek z vrcholu zásobníku se odstraní, v opačném případě dojde ke smazání celého (pod)plánu. Procedura `nextRun` pro jeden krok interpretu vypadá následovně:

```
Procedure nextRun;
  item = Stack.Top();
  if (not item == null) then
  begin
    case item.type:
      bbadd: res = call add_bb(item);    // přidání do báze znalostí
      bbdel: res = call del_bb(item);    // odebrání z báze znalostí
      snd: res = call send_msg(item);     // odeslání zprávy
      rcv: res = call recv_msg(item);     // příjem zprávy
      direx: res = call dir_exec(item);   // přímé spuštění
      index: res = call indir_exec(item); // nepřímé spuštění
      bbtst: res = call test_bb(item);    // test báze znalostí
      srvc: res = call service(item);     // volání služby platformy
      regsw: res = call reg_switch(item); // změna aktiv.registru
      stp: res = true;
    if (res) then
      Stack.Pop();
    else
      pop_plan();
```

```
  end;
  else begin
    Platform.Stop();
  end;
end;
```

Funkce `pop_plan` odstraňuje z vrcholu zásobníku prvky, dokud nějaké jsou nebo dokud prvek na vrcholu není zarážka (#).

```
Function pop_plan();
  do begin
    item = Stack.Top();
    Stack.Pop();
  end while not item == null or not item = '#';
end;
```

Vkládání seznamů do báze znalostí může selhat na nedostatku místa v poli pro kód. Nejdříve se vládaný seznam unifikuje s bází, při kladném výsledku k vložení nedojde. V opačném případě se nejdříve do báze vloží seznam, tak jak je zadaný, a poté dojde k substituci symbolů registrů v seznamu. Při selhání substituce je nutné vložený seznam z báze smazat.

```
Function add_bb(item);
  if (Table.Unify(TBL_BB,item)) then
    return true;
  else begin
    if (not Table.Add(TBL_BB,item)) then
      return false;
    if (not Table.Substitution(TBL_BB)) then
    begin
      Table.Delete(TBL_BB,item);
      return false;
    end;
    return true;
  end;
end;
```

Odebírání seznamů z báze znalostí probíhá v cyklu, dokud se unifikací tabulky se zadaným seznamem nějaký najde.

```
Function del_bb(item);
  while (not (find = Table.Unify(TBL_BB,item)) == null) do
  begin
    Table.Delete(TBL_BB, find);
  end;
end;
```

Odesílání zpráv přes rádio lze považovat za jednu ze služeb platformy a je tak i implementována. První položka seznamu-parametru je adresa cílové platformy, druhá je zpráva, která se zkopí-

ruje do výstupního bufferu a zavolá se funkce platformy pro zahájení odesílání. Během kopírování se zároveň provádí substituce registru. Velikost bufferu je omezena, tudíž se musí kontrolovat počet zkopírovaných znaků. Při překročení velikosti končí funkce chybou a k odeslání nedojde.

```
Function send_msg(item);
  if (not copy(out_buf,item[2])) then
    return false;
  call Platform.send_msg(item[1],out_buf);
  return true;
end;
```

Příjem zprávy je operace podobná nepřímému spuštění s tím rozdílem, že se testuje vstupní báze místo báze plánů. Seznamy v ní jsou dvouprvkové, první je adresa platformy, z které zpráva přišla, a druhý je vlastní zpráva. Hledá se pouze první shoda a druhý prvek nalezené položky se přesune do aktivního registru, zbytek seznamu je z báze smazán.

```
Function recv_msg(item);
  if (not (find = Table.Search(TBL_IB,item)) == null) then
    Table.Move(TBL_ACTREG,find[2]);
    Table.Delete(TBL_IB,find);
    return true;
  end;
  else return false;
end;
```

Přímé spuštění plánu je pouhé vložení položek společně se zarážkou na zásobník, ale až za první prvek, protože ten je právě prováděná akce.

```
Function dir_exec(item);
  if (not Stack.Push_second('#')) then
    return false;
  else
  return Stack.Push_second(item);
end;
```

Funkce nepřímého spuštění prohledává bázi plánů a hledá plán se zadaným jménem. Když je plán nalezen, jsou jeho akce vloženy společně se zarážkou za první prvek na zásobník. Seznamy v bázi plánů jsou opět dvouprvkové, první prvek je jméno plánu a druhý je seznam jeho akcí.

```
Function inder_exec(item);
  if (not (find = Table.Search(TBL_PB,item)) == null) then
  begin
    if (not Stack.Push_second('#')) then
      return false;
    return Stack.Push_second(item[2]);
  end;
end;
```

Testování báze znalostí taktéž využívá unifikaci a z položek, které jsou unifikovatelné, vytvoří seznam, který následně vloží do aktivního registru. Procházení báze znalostí probíhá v cyklu a výsledkem akce může být i prázdný seznam. Jedině při nedostatku místa v poli končí akce chybou a k vložení vytvořeného seznamu nedojde.

```
Function test_bb(item);
  List res;
  while (not (find = Table.Unify(TBL_BB,item)) == null) do
  begin
    res.insert(find);
  end;
  return Table.Add(TBL_ACTREG,res);
end;
```

Volání služby platformy spočívá v podstatě pouze v předání celého seznamu funkci z platformy. Ta zpracuje všechny parametry, provede službu a vrátí její výsledek.

```
Function service(item);
  return Platform.service(item);
end;
```

Poslední typ akce – přepnutí aktivního registru změní hodnotu proměnné, která obsahuje identifikátor registru, na registr zadaný číslem a vymaže jeho obsah.

```
Function reg_switch(item);
  case item.value
    '1': TBL_ACTREG = TBL_REG1;
    '2': TBL_ACTREG = TBL_REG2;
    '3': TBL_ACTREG = TBL_REG3;
  end;
  Table.Delete(TBL_ACTREG);
end;
```

# 10 Průběh implementace

Začátek implementace byl hlavně spojen s návrhem algoritmů pro práci s polem (kopírování, vkládání, přesuny, mazání) a unifikačního algoritmu. Všechny funkce byly v jedné komponentě a další rozšiřování činilo značné problémy. Proto byla vytvořena struktura interpretu složená z vrstev. Nejnižší vrstva představuje vlastní pole řetězce v paměti, následuje vrstva operací s polem: přesuny, mazání a vkládání častí řetězců. Nad těmito operacemi jsou vytvořeny datové struktury tabulka a zásobník, které reprezentují další vrstvu. Nejvyšší vrstva je interpretační cyklus agenta.

Každá vrstva či její část je reprezentována jednou komponentou, která poskytuje rozhraní s požadovanými operacemi a využívá rozhraní ostatních potřebných komponent. Díky tomuto návrhu je zaručeno, že komponenta na vyšší úrovni vždy využívá služby nižších úrovní a nikdy naopak. Interpret například pracuje se zásobníkem, vkládá na něj prvky, ale už ho „nezajímá“ jak se posunují části pole a kopírují jeho části.

Implementace komponent probíhala od spodní vrstvy s polem. Funkce každé vrstvy byly zvlášť testovány. Po vytvoření základní struktury interpretu se postupně přidávaly další funkce do jednotlivých vrstev, aby se splnily požadavky na jeho funkčnost. Při interpretaci je vypuštěna komplexnější kontrola syntaxe a sémantiky z důvodu zjednodušení celého programu.

Vývoj platformy probíhal zpočátku samostatně jako diplomová práce kolegy Horáčka, v určité fázi bylo však nutné interpret a platformu spojit a pracovat s jedním celkem. Po úspěšném spojení našich prací následovala implementace základních služeb platformy a první „program“ pro blikání LED diod na desce Mote. Během této fáze se objevila spousta problémů, které se nepodařilo odhalit během samostatného vývoje. Do interpretu byla zapracována substituce registrů ve všech akcích a službách platformy pro možnost vytvoření složitějších „programů.“ Dalším krokem bylo zprovoznění komunikace přes rádio, tj. posílání zpráv mezi agenty (platformami) a přenos celého kódu agenta mezi platformami. Přenos zpráv byl demonstrován na modifikaci prvního „programu,“ který posílal informace na druhý Mote, jenž tyto zprávy zpracovával a podle obsahu zprávy blikal. Do platformy bylo poté přidáno měření dat ze senzorů, statistické zpracování naměřených hodnot a jejich ukládání do trvalé (flash) paměti Mote. Jako poslední část vývoje v rámci našich prací bylo vytvoření jednoduchého programu v jazyce Java, který umožňuje posílání a příjem zpráv od agentů v sítí a také posílání kódu agenta na zadaný Mote. Díky tomu není nutné mít ve spouštěcí funkci interpretu naplnění tabulek a zásobníku plánem a daty.

V následujících podkapitolách jsou popsány jednotlivé komponenty interpretu, funkce z poskytovaných rozhraní a jejich význam.

## 10.1 ArrayC

Tato komponenta obsahuje pole s celým kódem a indexy jednotlivých částí. Dále přes rozhraní `ArrayI` poskytuje funkce pro zjišťování/nastavování jednotlivých indexů, vrácení ukazatele na začátek pole a hlavně ladící funkce, které při simulaci vypisují obsah pole a jednotlivé indexy. Komponenta poskytuje rozhraní `StdControl`, které se používá pro standardizované ovládání komponent – spouštění a zastavování činnosti. Zde se využívá jen spuštění, vymaže se obsah pole a vynulují jednotlivé indexy.

## 10.2 ArrayOpC

Pro další vrstvy je nutná sada základních operací s částí pole. Ty implementuje tato komponenta a poskytuje rozhraní `ArrayOpI`. Mezi operace patří mazání, vkládání, přesun a kopírování částí řetězce. Při všech těchto operacích probíhá kontrola parametrů, aby nedošlo k nesprávnému chování programu nebo poruše integrity pole. Dále deleguje funkce pro nastavování a zjišťování indexů částí pole, implementuje funkce pro zjištění znaku na zadaném indexu a pro zjištění délky pole, také implementuje ladící funkci pro výpis jednotlivých částí pole. Komponenta také poskytuje rozhraní `StdControl` pro její ovládání, jeho funkce volají stejnojmenné ekvivalenty z komponenty `ArrayC`.

## 10.3 StackC

Operace nad zásobníkem jsou implementovány v této komponentě, která poskytuje rozhraní `StackI` pro komponenty z vyšších vrstev. Mezi funkce patří `empty`, `push`, `top` a `pop`, které pracují s indexy v poli z výše uvedené komponenty `ArrayC`. Dále `push_str`, která vkládá obecný řetězec na zásobník, a dvojce funkcí `push_second` a `push_second_str`, které porušují principy zásobníku a vkládají prvky až za druhý prvek. Tohoto se využívá při volání plánu (ať už přímém nebo nepřímém), kde na vrcholu zásobníku je stále aktuálně prováděná operace a je nutné vložit obsah plánu až za ni.

Během vkládání může dojít k zaplnění pole, v takovém případě operace vkládání končí chybou a interpret využije funkci `pop_plan` z této komponenty pro smazání aktuálního pod/plánu. Tato funkce odebírá prvky z vrcholu zásobníku, dokud není prázdný nebo nenarazí na zarážku podplánu.

Jelikož nelze v jazyce udělat obecný nekonečný cyklus, je nutné volat na konci každého plánu nějaký další plán (ať už ten, který se právě vykonává, nebo jiný). To by ale způsobovalo neustálé vkládání zarážky za podplán a po několika stech krocích by došlo k zaplnění kódu agenta jen těmito zarážkami. Proto se při funkcích `push_second` a `push_second_str` kontroluje, zdali není vkládaný prvek/řetězec a druhý prvek od vrcholu zásobníku zarážka, v takovém případě k vložení nedojde, neboť je to zbytečné.

## 10.4 TableC

Dalšími důležitými částmi agenta jsou tabulky. Práci s nimi obstarává tato komponenta a poskytuje funkce skrze rozhraní `TableI`. Při volání některých funkcí je nutné určit, s jakou tabulkou se bude pracovat, proto každá taková funkce má jako první parametr identifikátor tabulky. Toto platí pro funkce `empty`, `add`, `add_str`, `delete` a `unify`.

První jmenovaná kontroluje prázdnost zadané tabulky. Další dvě vkládají prvky do tabulky buď pomocí indexů do pole, nebo z libovolného řetězce. Před vložením se provádí unifikace řetězce a tabulky, zda už vkládaný prvek obsahuje. Při kladném výsledku k vložení nedojde, neboť tabulka nemůže obsahovat dva stejné prvky. Tyto funkce mohou skončit chybou, pokud se nepodaří prvky vložit do tabulky z důvodu nedostatku místa v poli. Funkce `delete` taktéž provádí unifikaci, aby zjistila, které prvky má ze zadané tabulky smazat, nemusí však smazat žádný prvek, v tom případě končí operace chybou. Poslední výše uvedená funkce unifikuje zadaný prvek pomocí indexů s tabulkou a výsledek vkládá do aktivního registru v podobě seznamu nalezených n-tic.

Pro potřeby platformy implementuje komponenta funkce `addInputBase` a `addInputBaseSen`, které vloží zadaný řetězec nebo data ze senzoru do tabulky InputBase.

Pro práci s registry zde jsou funkce `changeActReg`, `delActReg`, `getActReg` a `getActRegIndex`, první změní aktivní registr na registr zadaný indexem čísla v poli, druhá smaže aktivní registr, třetí vrací identifikátor aktivního registru a čtvrtá index začátku aktivního registru v poli. Tato komponenta obsahuje proměnnou s identifikátorem aktivního registru.

Poslední dvě poskytované funkce jsou `findPlan` a `searchInputBase`. První prohledává tabulku planBase a hledá plán podle zadaného jména, druhá pracuje na podobném principu, hledá v tabulce InputBase data z požadovaného senzoru nebo zprávy z určité platformy.

## 10.5 AgentC

Tato komponenta stojí na pomyslném vrcholu vrstevného modelu interpretu. Probíhá zde vlastní interpretace operací a z rozhraní `ControlI` implementuje dvě události, které řídí činnost interpretu. Toto rozhraní obsahuje dvě funkce: `booted` a `nextRun`.

`Booted` slouží k inicializaci celého interpretu, spustí se komponenta `ArrayOp` a v případě potřeby se naplní tabulka a zásobník počátečními daty.

Funkce `nextRun` provede jeden krok interpretu. Nejdříve se zjistí indexy vrcholu zásobníku, poté se podle znaku operace provede příslušná akce a podle jejího výsledku se smaže vrchol zásobníku nebo celý pod/plán. Návratová hodnota funkce je určena výsledkem akce. Během provádění se využívá služeb nižších vrstev a také platformy (práce s rádiem nebo volání služeb platformy).

## 10.6 PlatformSvcC

Pro spojení interpretu agenta a platformy slouží toto rozhraní. Agent ve své podstatě pouze řídí běh programu a nemá žádnou výpočetní schopnost, proto existuje sada služeb platformy, která je implementována v této komponentě a poskytována agentovi skrze akci volání služby s příslušnými parametry.

Služby zahrnují operace se seznamy (funkce car/cdr známé z Lispu), komunikaci po rádiu, získávání dat ze senzorů a další funkce (ovládání LED, čekání). Komponenta poskytuje v rozhraní `PlatformSvcI` funkce `svcCall`, `sendMsg` a využívá rozhraní pro práci s polem `ArrayOpI`, tabulkami `TableI` a další potřebné rozhraní z platformy (`WaitI`,`SendI` a `LedI`).

Funkce `svcCall`, které se předají indexy začátku a konce n-tice s parametry služby, podle prvního prvku seznamu určí, o jakou službu jde, a zavolá příslušnou vnitřní funkci komponenty. Pro odeslání zprávy (obecně n-tice) přes rádio slouží funkce `sendMsg`, která má opět dva parametry určující indexy začátku a konce parametru reprezentovaný n-ticí. První prvek je číselná adresa cílové platformy, na kterou se má odeslat n-tice z druhého parametru funkce. Během kopírování n-tice do výstupního bufferu dochází ke kontrole na zástupné znaky registrů a v případě nalezení jsou nahrazeny aktuálním obsahem daného registru. Jelikož je velikost bufferu omezena, musí se před začátkem každého dílčího kopírování kontrolovat, zda nehrozí přepis paměti za koncem pole. Pokud taková situace nastane, končí akce chybou a k odeslání zprávy nedojde, v opačném případě se zavolá funkce pro započetí odesílání `SendI.sendMessgage`, které se předá cílová adresa a délka odesílané n-tice.

Implementace funkcí car/cdr jsou z velké části stejné. Nejdříve se zjišťuje, zda zadané místo v paměti obsahuje více jednotlivých n-tic. Pokud ano, vloží se do aktivního registru první, případně

zbylé, n-tice. V opačném případě se hledá první čárka, ta identifikuje, že v n-tici je více prvků. Během hledání čárky se také zjišťuje, zda je první prvek n-tice nebo ne, to ovlivňuje začátek a konec vložené n-tice.

Funkce pro další služby platformy vždy zpracují parametry, vyžaduje-li služba nějaké, a zavolají příslušný příkaz z komponent platformy. Popis všech služeb je uveden v předcházející kapitole.

# 11 Příklady kódů agenta

Jak již bylo zmíněno v předešlé kapitole, pro testování funkčnosti interpretu a platformy byly použity dva jednoduché „programy,“ které měly v dané fázi vývoje demonstrovat aktuální funkčnost, a proto budou v této kapitole podrobně popsány jakožto ukázka kódů v upraveném jazyce ALLL. Při implementaci jednotlivých akcí a služeb byly použity jednodušší a kratší kousky kódu (do pěti akcí v plánu), které však neměly komplexní význam a nebudou zde uvedeny.

## 11.1 Blikání LED na Mote

První důležitý milník ve vývoji byl kód agenta, který měl při správné implementaci blikat LED diodami na Mote v zadaném pořadí a čase jednotlivých bliknutí. Kód má v bázi plánů dva pojmenované plány, jednoduchý počáteční plán na zásobníku a vypadá takto:

| planBase: |
|---|
| (blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2&(1)$(r,&2)&(3)$(f,&1)$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)^(blik)))<br>(napln,(+(led,r,600)+(led,g,700)+(led,y,800)^(blik)#^(napln))) |
| plan: |
| $(l,(r,1))$(l,(g,1))$(l,(y,1))$(w,(300))$(l,(r,0))$(l,(g,0))$(l,(y,0))^(napln) |

**Tabulka 11.1 - struktura kódu pro blikání LED**

První plán se jménem „blik“ nejdříve vybere z báze znalostí všechny n-tice, které jsou unifikovatelné se zadanou n-ticí, první z nich z báze odstraní a podle prvků v n-tici rozsvítí LED diodu na zadaný čas v milisekundách. Když se unifikací nenajde žádná n-tice, plán končí chybou. Druhý plán „napln“ pouze vloží do báze znalostí předpis, jak se má blikat, vyvolá první plán a při jeho chybě opět vyvolá sám sebe. Počáteční plán rozsvítí na krátkou dobu všechny LED a vyvolá naplnění báze znalostí.

Popis akcí prvního plánu:

| Akce | Význam |
|---|---|
| &(1) | Přepnutí aktivního registru na první |
| *(led,_,_) | Unifikace báze znalostí se zadanou n-ticí, výsledek se vloží do aktivního (prvního) registru, nenajde-li se žádná n-tice, plán končí chybou |
| &(2) | Přepnutí aktivního registru na druhý |
| $(f,&1) | Do aktivního (druhého) registru se vloží první prvek n-tice z prvního registru (první n-tice (led,<barva>,<cas>) |
| -&2 | Tento prvek se odstraní z báze znalostí |
| &(1) | Přepnutí aktivního registru na první |
| $(r,&2) | Do aktivního (prvního) registru se vloží zbylé prvky n-tice z druhého registru, odstraní se „led“ z n-tice |
| &(3) | Přepnutí aktivního registru na třetí |

| $(f,&1) | Do aktivního (třetího) registru se vloží první prvek n-tice z prvního registru, který reprezentuje kód barvy LED |
|---|---|
| $(l,&3) | Zavolání služby pro ovládání LED, ve třetím registru je pouze kód barvy |
| &(2) | Přepnutí aktivního registru na druhý |
| $(r,&1) | Do aktivního (druhého) registru se vloží zbylé prvky n-tice z prvního registru, odstraní se kód barvy z n-tice |
| &(1) | Přepnutí aktivního registru na první |
| $(f,&2) | Do aktivního (prvního) registru se vloží první prvek n-tice z druhého registru, který reprezentuje dobu, po jakou má LED svítit |
| $(w,&1) | Zavolání služby pro pozastavení interpretu na zadaný čas, který je uložen v prvním registru |
| $(l,&3) | Zavolání služby pro ovládání LED, ve třetím registru zůstal kód barvy |
| ^(blik) | Opětovné vyvolání tohoto plánu |

**Tabulka 11.2 - popis jednotlivých akcí v prvním plánu**

Popis akcí druhého plánu:

| Akce | Význam |
|---|---|
| +(led,r,600)<br>+(led,g,700)<br>+(led,y,800) | Do báze znalostí se vloží tyto n-tice, které určují, které LED a jak dlouho mají svítit, není možné do báze znalostí vložit dvakrát stejnou kombinaci barvy a času |
| ^(blik) | Vyvolání plánu „blik" |
| # | Zarážka za plánem, při chybě plánu se smažou všechny akce až po tuto zarážku |
| ^(napln) | Opětovné vyvolání tohoto plánu, naplní se znovu báze znalostí a je tímto vytvořena nekonečná smyčka |

**Tabulka 11.3 - popis jednotlivých akcí v druhém plánu**

Popis počátečního plánu na zásobníku:

| Akce | Význam |
|---|---|
| $(l,(r,1))<br>$(l,(g,1))<br>$(l,(y,1)) | Zavoláním těchto služeb se rozsvítí všechny LED na Mote |
| $(w,(300)) | Krátké pozastavení interpretu |
| $(l,(r,0))<br>$(l,(g,0))<br>$(l,(y,0)) | Zhasnutí všech LED |
| ^(napln) | Vyvolání plánu „blik," začátek nekonečné smyčky |

**Tabulka 11.4 - popis jednotlivých akcí počátečního plánu**

# 11.2 Vzdálené blikaní mezi dvěma Mote (rádiová komunikace)

Druhý kód je rozšířená verze kódu prvního a je složen ze dvou částí, neboť každý Mote musí mít jiný. Do kódu pro první Mote přibyl jeden plán „odesli,“ který odešle n-tici přes rádio a čeká na potvrzení od druhého Mote, ten pouze čeká na příchozí zprávu, odešle ji jako potvrzení nazpět a blikne podle jejího obsahu. Kód pro první Mote vypadá takto:

| planBase: |
|---|
| `(blik,(&(1)*(led,_,_)&(2)$(f,&1)-&2^(odesli)&(1)$(r,&2)&(3)$(f,&1)`<br>`$(l,&3)&(2)$(r,&1)&(1)$(f,&2)$(w,&1)$(l,&3)^(blik)))`<br>`(napln,(+(led,r,600)+(led,g,700)+(led,y,800)^(blik)#^(napln)))`<br>`(odesli,($(a)!(2,&2)&(3)$(s)?(2)))` |
| plan: |
| `$(l,(r,1))$(l,(g,1))$(l,(y,1))$(w,(300))$(l,(r,0))$(l,(g,0))`<br>`$(l,(y,0))^(napln)` |

**Tabulka 11.5 - kód pro první Mote**

Kód je téměř shodný s prvním programem. Do plánu ‚blik‘ přibyla jedna akce – vyvolání plánu „odesli.“ Popis akcí tohoto plánu:

| Akce | Význam |
|---|---|
| `$(a)` | Tato služba zapne čekání na příchozí zprávu |
| `!(2,&2)` | Odeslání obsahu druhého registru na platformu s adresou 2 |
| `&(3)` | Přepnutí aktivního registru na třetí |
| `$(s)` | Uspání interpretu do příjmu zprávy přes rádio |
| `?(2)` | Vyzvednutí n-tice ze vstupní báze |

**Tabulka 11.6 - popis akcí plánu ‚odešli'**

Báze plánu druhého kódu obsahuje dva plány, první čeká na příjem zprávy a vyvolá druhý plán, který je zjednodušenou verzí stejnojmenného plánu v kódu pro první Mote a nebude podrobně popsán. Z plánu je odstraněn výběr n-tic z báze znalostí a na konci nevolá sám sebe, ale první plán. Počáteční plán na zásobníku pouze vyvolá první plán.
Kód pro druhý Mote:

| planBase: |
|---|
| `(prijem,($(s)&(2)?(1)$(a)!(1,&2)^(blik)))`<br>`(blik,(&(1)$(r,&2)&(3)$(f,&1)$(l,&3)&(2)$(r,&1)&(1) $(f,&2)$(w,&1)`<br>`$(l,&3)^(prijem)))` |
| plan: |
| `^(prijem)` |

**Tabulka 11.7 - kód pro druhý Mote**

Popis akcí prvního plánu:

| Akce | Význam |
|---|---|
| `$(s)` | Uspání interpretu do příjmu zprávy přes rádio |
| `&(2)` | Přepnutí aktivního registru na druhý |
| `?(1)` | Vyzvednutí n-tice ze vstupní báze |
| `$(a)` | Zapnutí čekání na příchozí zprávu |
| `!(1,&2)` | Odeslání obsahu druhého registru na platformu s adresou 1 |
| `^(blik)` | Vyvolání plánu „blik“ |

**Tabulka 11.8 - popis akcí plánu ‚prijem'**

# 12 Závěr

Podařilo se nám vytvořit funkční platformu a interpret jazyka ALLL, které jsou určeny pro uzly senzorové sítě a které doposud existovaly pouze jako simulátor na běžném PC. Tato práce je v podstatě začátek dlouhodobého vědeckého snažení, na kterém se podílí několik lidí a jehož cílem by mělo být vytvoření komplexní platformy pro mobilní agenty včetně řízení a sledování pomocí běžného počítače. Platforma umožňuje řadu základních funkcí (řízení interpretu, kompletní obsluha rádia, měření dat ze senzorů a ukládání hodnot do trvalé paměti) a poskytuje několik služeb interpretu. Ten pracuje podle návrhu a dokáže interpretovat všechny typy akcí. Některé jeho části jsou ale značně zjednodušeny, například syntaxe je kontrolována pouze částečně a sémantika je kontrolována jen při volání služeb platformy. To je možné díky vysoké míře abstrakce dat.

Další práce může zahrnovat úpravu stávajícího simulátoru tak, aby podporoval upravenou verzi jazyka, choval se jako skutečný uzel a umožňoval obousměrnou komunikaci s reálnými uzly v síti. Přidávání služeb do platformy a rozšiřování schopností interpretu je také další nezbytnou součástí vývoje. Jedním z praktických využití agentních systému na této platformě může být například sledování šíření požáru v budovách nebo stavu betonových pilířů mostů (vnitřní struktura, pohyby, praskliny).

# Literatura


[1] Bradáč, Z.: Bezdrátový komunikační standard ZigBee, *Automazitace*, 2005, vol. 48, no. 4, s. 261, ISSN: 005-125X.

[2] *datasheet MICAz*, Crossbow Technology, URL <http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf>

[3] *datasheet MTS400CA, MTS420CA*, Crossbow Technology, URL <http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0053-01_A_MTS400-420.pdf>

[4] Handziski, V.; et. al.: *TEP 2 – Hardware abstraction architecture* [online], [cit. 25-04-2009], URL <http://www.tinyos.net/tinyos-2.1.0/doc/html/tep2.html>

[5] Hill, J.L.; Culler, D.E.: Mica: A wireless platform for deeply embedded network, *IEEE Micro*, Listopad/Prosinec 2002, vol. 22, no. 6, s12-24, ISSN: 0272-1732.

[6] Levis, P.; et al.: *T2: A Second Generation OS For Embedded Sensor Networks*, Berlin, Listopad 2005

[7] Pike, J.: *Integrated Undersea Surveillance System (IUSS)* [online], [cit. 16-04-2009], URL <http://www.fas.org/irp/program/collect/iuss.htm>

[8] Pike, J.: *The sound surveillance system (SOSUS)* [online], [cit. 16-04-2009], URL <http://www.fas.org/irp/program/collect/sosus.htm>

[9] *produktové stránky fy Atmel procesoru ATmega128* [online], [cit. 28-04-2009] Atmel, URL <http://www.atmel.com/dyn/products/product_card.asp?part_id=2018>

[10] Russel, S.; Norvig, P.: *Artificial Intelligence: A modern Approach*, 2. vydání, New Jersey, USA: Prentice Hall, 2003, ISBN: 0-1310-3805-2, Kapitola 2, s. 31-52.

[11] webové stránky TinyOS, URL <www.tinyos.net>

[12] webové stránky nesC na SourceForge, <http://nesc.sourceforge.net/>

[13] Zbořil, F.: Simulation for Wireless Sensor Networks with Intelligent Nodes, In *10th International Conference on Computer Modelling and Simulation*, Cambridge, GB: IEEE Computer Society, 2008, ISBN: 0-7695-3114-8.

# Seznam příloh

Příloha 1. CD se zdrojovými texty
Příloha 2. Upravená BNF jazyka ALLL

# Příloha 2

```
<agent>         ::= <seznam-planu> <seznam-akci> <seznam-ntic>
                    //PB, plan, BB

<seznam-planu> ::= <plan> | <plan> <seznam-planu> | e
<seznam-akci>  ::= <akce> | <akce> <seznam-akci>
<seznam-ntic>  ::= <ntice> | <ntice> <seznam-ntic> | e

<plan>          ::= ( <retezec> , ( <seznam-akci> ) )

<akce>          ::= + <ob-ntice> | - <ob-ntice>
                    ! ( <cislo> , <ob-ntice> ) | ? ( <cislo> ) |
                    @ ( <seznam-akci> ) | * <ob-ntice> |
                    $ ( <sluzba> ) | ^ ( <retezec> ) |
                    & ( <reg-id> )

<sluzba>        ::= a | f , <registr> | f , <ntice-2> | k |
                    l , ( <led> ) | m , <registr> | m , ( <retezec> )
                    r , <registr> | r , <ntice-2> | s |
                    w , <registr> | w , ( <cislice> ) | d |
                    d , <registr> | d , ( <data> )

<ob-ntice>      ::= ( <ob-polozky> ) | <registr>
<ntice>         ::= ( <polozky> )
<ntice-2>       ::= ( <polozky-2> )

<polozky>       ::= <polozka> | <polozka> , <polozky>
<polozka>       ::= <retezec> | <ntice> | _

<polozky-2>     ::= <polozka-2> | <polozka-2> , <polozky-2>
<polozka-2>     ::= <retezec> | <ntice>

<ob-polozky>    ::= <ob-polozka> | <ob-polozka> , <ob-polozky>
<ob-polozka>    ::= <retezec> | <ob-ntice> | <registr> | _

<retezec>       ::= <znak> | <znak> <retezec>
<registr>       ::= & <reg-id>

<led>           ::= <barva> | <barva> , <stav>
<data>          ::= <typ> , <cislice>
<znak>          ::= a..z | A..Z | <cislice>
<cislo>         ::= <cislo> | <cislice> <cislo>
<cislice>       ::= 0..9
<reg-id>        ::= 1..3
<barva>         ::= r | g | y
<typ>           ::= a | m | M
<stav>          ::= 0 | 1
```