VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Microsoft Robotics Developer Studio - programování robotů

Programming Robots - Microsoft Robotics Developer Studio

2010

Bc. Michal Pešat

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Na tomto místě bych rád poděkoval RNDr. Elišce Ochodkové, Ph. D. za odbornou pomoc a profesionální přístup při vedení této diplomové práce.

Abstrakt

Práce se zabývá možnostmi programování robotů pomocí nástroje Microsoft® Robotics Developer Studio. Cílem práce bylo navrhnout a vypracovat aplikaci pro průzkum neznáme místnosti pomocí robota, místnost zaznamenat a pokusit se ji zpracovat ve simulováném prostředí. Implementace zadaných úloh byla zkoumána pro dvě platformy, především pro LEGO NXT a simulované prostředí MRDS. Práce popisuje teoretickou rovinu používání nástroje MRDS k řízení robotů, jeho princip fungování a architekturu. Zabývá se také stavebnicí LEGO NXT, která je využita k realizaci úloh.

Klíčová slova: MRDS, LEGO NXT, VSE, VPL, NXT-G, Explorer, SimplySim

Abstract

This thesis deals with possibilities of programming robots using Microsoft ® Robotics Developer Studio. Goals of this thesis were to design and to implement application for exploring unknown environment, then it should store explored room and to try use it in simulation environment. Implementation was done for tho different platforms, mainly for LEGO NXT and for simulated environment VSE. This thesis describes the theoretical level, using MRDS to robot control, the operating principle and its architecture. It also discusses the LEGO NXT, which is used to implement the tasks.

Keywords: MRDS, LEGO NXT, VSE, VPL, NXT-G, Explorer, SimplySim

Seznam použitých zkratek a symbolů

MRDS	–	Microsoft® Robotics Developer Studio
CCR	–	Concurrency and Coordination Runtime
VPL	–	Visual Programming Language
VSE	–	Visual Simulation Environment
DSS	–	Decentralized Software Services
DSSP	–	Decentralized Software Services Protocol
REST	–	Representational State Transfer
SOAP	–	Simple Object Access Protocol
HW	–	Hardware
PC	–	Personal Computer

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Robotika se v dnešní době dostává čím dál víc do popředí. Už se jí nevěnují pouze kruhy akademické, vojenské nebo medicínské, ale i obyčejní lidé, kteří mají zájem o robotiku. Pořídit si svého vlastního robota se stává jednodušším a navíc roboti začínají mít větší a větší inteligenci a možnosti jak komunikovat se svým okolím. Je zřejmé, že si člověk doma nevytvoří robota, který by uměl zneškodnit výbušninu nebo operovat, protože se jedná více o hračky v podobě robotických vysavačů nebo hraček.

Jako příklad relativně levného řešení jak si pořídit a začít s roboty je stavebnice LEGO Mindstorms NXT, která je použita v této práci. Samozřejmě příchodem prostředí MRDS a jeho nástroje VSE je možné vytvořit či použít předpřipravené robotické modely v simulovaném prostředí, což je pro začátek a otestovaní možností robotiky postačující. Ovšem ne vše funguje v simulovaném prostředí stejně jako v tom reálném.

Cílem práce je vytvořit aplikaci, která bude schopna ovládat robota tak, aby se byl schopen pohybovat po neznámé místnosti. Všechny své pohyby by měl být schopen zaznamenat a na základě takto získaných dat má vytvořit obraz místnosti. Tento obraz by měl mít po té k použití v simulovaném prostředí VSE, ve kterém by se měl simulovaný ekvivalent reálného robota pohybovat bez nárazu. Stejně jako reálný robot by měl i simulovaný umět se pohybovat po neznámém prostředí, které si má zapamatovat a ve výsledku vytvořit mapu místnosti.

Po té co si robot vytvoří obraz místnosti by měl být tuto místnost projet bez kolizí s překážkami, díky výběru senzorů, které robot používá v této práci, projede místnost bez kolizí v podstatě již napoprvé.

Celá práce je rozčleněna do několika tématických celků, které na sebe více či méně navazují. Proto je dobré si přečíst práci od začátku, obzvlášť pokud čtenář nezná prostředí MRDS a NXT, a obecně prostředí robotiky a její problematiky. Tato práce by měla být nápomocna k jejímu lepšímu pochopení a jednodušším začátkům.

V kapitole 2 je představeno prostředí MRDS, jeho historie, architektura a základní součásti tohoto prostředí jako je knihovna CCR, která nám pomáhá s asynchronním prostředím robotiky, prostředí DSS ve kterém běží služby ovládající samotné roboty. Dále je zde zmíněno prostředí VPL a simulační prostředí VSE, které je taktéž využito k implementaci této práce. V kapitole 3 se podíváme na HW Robota LEGO NXT, který je použit pro vypracování úloh této práce. Je zde popsána struktura robota, jeho senzorů. Dále v této kapitole zmiňuji jak funguje komunikace mezi robotem a počítačem a jaké problémy to přináší. Poslední část je věnována grafickému programovacímu jazyku NXTG, který je součásti balení tohoto robota. V kapitole 4 je představena konfigurace reálného a simulovaného robota. Největší pozornot je věnována tomu simulovanému, jelikož při jeho řešení jsem musel překonat několik problémů. Je zde například i popsáno jak vytvořit vlastní model robota pro simulované prostředí. V kapitole 5 se věnuji popisu návrhu jednotlivých služeb a celkovému pohledu na aplikace jako celek. Snažím se osvětlit jak jsem vymyslel logiku pro ovládání robota tak, aby splnil své úkoly dané zadáním. Pomocí několika diagramů je znázorněno jak jednotlivé služby komunikují mezi sebou jako jeden celek. V kapitole 6 se snažím popsat implementaci jednotlivých služeb, které dají dohro-

mady aplikaci pro řízení robota splňující zadání. Je zde popsáno s jakými problémy jsem se setkal během implementace. Hlavně se zde zaměřuji na popis implementace reálného robota a mapovací služby.

2 Microsoft® Robotics Developer Studio

Vývojové prostředí Microsoft Robotics Developer Studi je nástroj určený k vývoji programů ovládajících roboty. Tato platforma umožňuje řízení reálných robotů jako jsou LEGO NXT, Pioneer, Rumba atd., ale také modelovat roboty a prostředí v simulátoru. Prostředí MRDS je složeno z několika částí, které budou popsány v následujících kapitolách. Jednotlivé části platformy jsou CCR, DSS, VPL, VSE, kde nejdůležitější jsou první dvě zmiňované, jelikož se starají o obsluhu a vykonávání ovládání robotů. Kromě standardního grafického jazyka VPL je možné vytvářet programy také v jazyku Visual Basic .NET, C#, JScript a IronPython. Poslední část tvoří simulované prostředí VSE, ve kterém je možné vymodelovat prostředí včetně simulovaných robotů a senzorů. Výsledné aplikace je možné sledovat a řídit skrz webové čí desktopové rozhraní, ve kterém je také možné zjišťovat aktuální stavy jednotlivých služeb.

2.1 Historie

Nástroj pro programování robotů Microsoft® Robotics Studio Developer a s ním související tým, začal vznikat na konci roku 2003, po té co několik společností a univerzit oslovilo společnost Microsoft. Tandy Trower, který pracoval pro Billa Gatese, sepsal studii, ve které popisuje co by měl Microsoft udělat pro podporu programování robotů, na základě této studie a po analýze obchodních příležitostí, se rozhodli, že založí tým věnující se této problematice. Výsledky tohoto týmu můžeme vidět v podobě platformy Microsoft Robotics Developer Studio.

V prosinci 2006 vychází první verze s označením 1.0, která je následována v červenci 2007 verzí 1.5. Postupem času vycházejí další vývojové verze až do července 2009, kdy vyšla zatím poslední verze s označením 2008 R2 a ta byla použita pro implementaci této diplomové práce.

2.2 Architektura

Platforma MRDS je založena na RESTful architektuře a aplikace jsou složené z jednotlivých služeb, které mezi sebou komunikují pomocí asynchronně posílaných zpráv. Zprávy jsou posílány pomocí HTTP protokolu v podobě SOAP zpráv. Samotné služby jsou vytvářeny pomocí knihovny CCR, která se stará o obsluhu vláken a je velice důležitou součástí tohoto frameworku, jelikož každá služba může být tvořena velkým množstvím vláken. Aktuální verze MRDS je založena na CLR 2.0 a díky tomu umožňuje velice dobrou přenositelnost mezi různými prostředími, jako jsou stolní počítače, PDA nebo mobilní telefony popř. přenositelnost mezi různými operačními systémy. Jelikož je vše založeno na webových službách, je velice jednoduché získávat informace o stavech jednotlivých služeb, měnit jejich konfiguraci atd., a to pouze pomocí webového prohlížeče. Navíc jednotlivé služby mohou běžet na různých počítačích apod.

Důležité je si uvědomit, že při vytváření programů pro ovládání robotů je nutné používat asynchronního programování, jelikož v robotice, stejně jako v běžném životě, se může odehrát několik událostí najednou, např. při jízdě vpřed může dojít ke stisknutí

dotykového senzoru, způsobené nárazem do překážky. Proto jednotlivé služby, které tvoří celou aplikaci, fungují nezávisle jedna na druhé a komunikují spolu pomocí zpráv. Tyto zprávy jsou ukládány do fronty, která se v prostředí MRDS nazývá port.

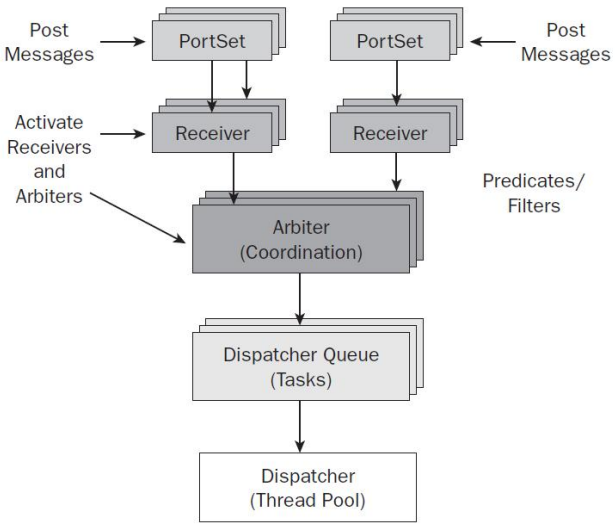
2.3 CCR

Jak již bylo zmíněno jednou ze základních komponent je CCR - Conccurency and Coordination Runtime, protože z podstaty robotiky vyplývá požadavek na multivláknové aplikace. Tato komponenta usnadňuje práci s paralelizací, koordinací a řízením chyb. Pomáhá nám vytvářet bloky kódu, které mohou běžet nezávisle na sobě. Pokud je to nutné, mohou komunikovat pomocí zpráv, které si předávají skrz fronty nazývané porty. Díky komunikaci pomocí zpráv není nutné synchronizovat jednotlivé kusy kódu během spuštění. Pokud je potřeba počkat na dokončení dvou nebo více operací, obsahuje tato knihovna prostředky i k těmto věcem. Samozřejmostí jsou součásti, které se starají o spouštění jednotlivých úkolů, systém pro správu chyb atd.

CCR nám přináší možnost jak jednoduše vytvořit multivláknové aplikace, bez toho aniž bychom museli psát složité konstrukce, které by se nám staraly o spouštění a řízení vláken. Není nutné se starat o synchronizaci proměnných atd. Avšak stále je možné dojít k zamrznutí pokud poskládáme kód pomocí CCR špatně.

2.3.1 Základní funkčnost

Pro fungování modelu založeném na CCR potřebujeme několik základních prvků, které dále doplňují další konstrukce knihovny CCR. Na obrázku 1 můžeme vidět spolupráci několika základních konstrukcí knihovny CCR, které budou popsány v následující podkapitolách.



Obrázek 1: Průběh zpracování zprávy v CCR. Zdroj: [1]

V jednoduchosti se dá říci, že celý proces začíná zasláním zprávy. Tato zpráva je uložena v portu, který přijímá zprávu daného typu. Zpráva zůstává uložena v portu dokud není vyzvednuta příjemcem. Po přijetí je zpracována a ze zprávy se vytvoří úloha. Tato úloha je uložena do fronty pro spuštění a následně poslána ke spuštění. Když je úloha naplánována ke spuštění, je spuštěna příslušná obsluha úlohy. A tato obsluha po dokončení nejčastěji vytvoří novou zprávu, která je zaslána na nějaký port a celý koloběh pokračuje.

2.3.2 Zpráva

Základním prvkem pro komunikaci mezi asynchronně běžícími službami je využit objekt nazvaný zpráva. Tento objekt může být jakéhokoliv typu v rámci prostředí .NET, který je serializovatelný. Během vyvíjení našich služeb si můžeme vytvořit vlastní typy zpráv, které si budou jednotlivé služby posílat mezi sebou. Typický příklad zprávy je zaslání požadavku na provedení operace služby.

Jelikož služby mohou běžet na různých místech, přesněji na různých počítačích, které jsou propojeny pomocí sítě, je nutné aby objekty, které budeme posílat jako zprávy, byly serializovatelné. Z tohoto vyplývá, že zpráva je před odesláním serializovaná do XML, poslána skrz síť a na cílovém místě je opět deserializována.

O zprávě taky někdy mluvíme jako o požadavku nebo odpovědi a to hlavně, v místech kde se bavíme o službách a jejich operacích.

2.3.3 Port

Pod pojmem port si můžeme představit jednoduchou frontu založenou na principu FIFO, která přijímá zprávy určitého typu. Zprávy jsou uchovávány v portu do doby než jsou vyzvednuty, ale zároveň odesílatel není blokován, jelikož zpráva je uložena ve frontě a odesílatel může pokračovat ve vykonávání činnosti. Samozřejmě mohou existovat případy, kdy odesílatel čeká na odpověď od příjemce. Ve výpisu 1 vidíme jednoduchou deklaraci portu, který přijímá pouze zprávy typu String a odeslání zprávy na port.

```
Port<String> simplePort = new Port<String>();
simplePort.Post("Test");
```

Výpis 1: Deklarace portu a umístění zprávy na port

2.3.4 PortSet

V předchozí části jsme si ukázali, jak funguje jednoduchý port, který umožňuje příjem pouze zpráv jednoho typu. Ale mohou existovat situace kdy potřebujeme zajistit příjem zpráv více typů. Dobrý příklad je hlavní operační port služby, který umí přijímat velké množství typů zpráv. Teprve logika obsluhy pak rozhodne, co se má s přijatým typem zprávy udělat. Pro tuto funkčnost je možné použít třídu `PortSet`, která má nadefinované typy zpráv, které přijímá podobně jako port.

```
PortSet<int, String> simplePortSet = new PortSet<int, String>();
simplePortSet.Post(10);
simplePortSet.Post("Test");
```

Výpis 2: Deklarace třídy `PortSet`

2.3.5 Úloha

Základní jednotkou práce v rámci CCR je úloha. Tato jednotka musí být po vytvoření zařazena do fronty pro zařazení ke zpracování a následně zpracována. Všechny třídy, které mají být využívány jako úlohy, musí implementovat rozhraní ITask. Toto rozhraní dává úlohám jednotnou formu, kterou musí splňovat, aby mohly být naplánovány a spuštěny. Pro vykonání úlohy musí existovat metoda, která se postará o vykonání požadované úlohy.

```
protected override void Start()
{
    base.Start();

    Spawn(BasicTaskHandler);
}

public void BasicTaskHandler()
{
    for (int i = 0; i < 5; i++)
    {
        Console.Writeline("Cislo i: " + i);
    }
}
```

Výpis 3: Příklad jednoduché obsluhy úlohy

V příkladu 3 je možné vidět úplně obyčejný příklad obsluhy nějaké úlohy, ve skutečné službě bude samozřejmě takováto úloha vypadat mnohem složitěji. Nejjednodušším způsobem jak vytvořit úlohu a zařadit ji do fronty je použít metodu `Spawn`, která je zděděna z třídy `DsspServiceBase`, která je předkem pro všechny služby v MRDS.

Pokud bychom chtěli, aby náš úkol předával nějaké vstupní hodnoty obsluze úlohy, můžeme to udělat jednoduše pomocí přidání typu a hodnoty k deklaraci jak je ukázáno ve výpisu 4

```
protected override void Start()
{
    base.Start();

    Spawn<int>(5, BasicTaskHandler);
}
```

```
public void BasicTaskHandler(int value)
{
    for (int i = 0; i < value; i++)
    {
        Console.Writeline("Cislo i: " + i);
    }
}
```

Výpis 4: Příklad úlohy se vstupními hodnotami

2.3.6 Arbitři a příjemci

Abychom mohli přijaté zprávy zpracovat a změnit je v úlohy, potřebujeme vytvořit příjemce. CCR obsahuje třídu nazvanou `Arbiter`, která nám pomáhá s vytvořením jednotlivých příjemců. Příjemci jsou uložení ve frontě, která je součástí jednotlivých instancí třídy `Port` nebo `PortSet`. Tato fronta není součástí fronty zpráv. Vytvoření příjemci vytvářejí jednotlivé úlohy, které jsou následně zpracovávány dále tím, že vyzvednou zprávu z fronty, zpracují ji. Pokud používáme k vytvoření příjemce třídu `Arbiter` je nutné příslušnou metodu této třídy aktivovat metodou `Activate`, která je získána z třídy `DsspServiceBase`. V následujících podkapitolách si ukážeme několik základních metod třídy `Arbiter`, které jsou běžně používané. Velice dobrou vlastností arbitrů je možnost je libovolně kombinovat a tak vytvářet jednoduché logické struktury.

2.3.6.1 Arbiter.Receive Jedná se o nejčastěji používaného arbitra. Vytváří příjemce, který čeká na přijetí zprávy. Po přijetí vytvoří příslušnou úlohu. Tento typ arbitra je možno perzistentně uchovat v portu, tzn. bude existovat a tedy čekat na přijetí zprávy dokud bude existovat příslušný port. Výpis 5 ukazuje trvalého arbitra typu Receive pro instanci třídy `Port` a `PortSet`. Pokud bychom chtěli vytvořit arbitra, který by přijal pouze jednu zprávu a po té by se ukončil, stačí první parametr změnit na false. Ve výpisu navíc můžeme vidět dva možné způsoby zpracování, první pomocí klíčového slova delegate a druhý pomocí metody.

```
Activate(
    Arbiter.Receive(true, simplePort,
    delegate(int i) {
        Console.WriteLine("Cislo i: {0}", i);
    }
    ));

Activate(
    Arbiter.Receive<int>(true, simplePortSet, ReceiveHandler));

private void ReceiveHandler(int i) {
    Console.WriteLine("Cislo i: {0}", i);
}
```

Výpis 5: Arbiter.Receive

2.3.6.2 Arbiter.Choice Tento druh arbitra vytváří dva příjemce, jako parametr přebírá instanci třídy `PortSet` a pro jednotlivé typy zpráv definuje obsluhu. Nejčastěji se využívá při volání operací služeb a čekání na odpověď resp. výsledek operace. Arbiter Choice je možné interpretovat jako logickou operaci `OR`. V příkladu 6 můžeme vidět zpracování zpráv v `simplePortSet`, tento příjemce počká na příchod první zprávy, jednoho z typů a naplánuje úlohu podle typu příchozí zprávy. Tento typ arbitra se neukládá persistentně do paměti, ale pouze do příchodu první zprávy. Po spuštění prvního příjemce ukončí nevyužitého příjemce.

```
Activate(
    Arbiter.Choice(simplePortSet,
    delegate(int i)
    {
        Console.WriteLine("Cislo i: " + );
    },
    delegate(String text)
    {
        Console.WriteLine("Text: " + text);
    }
));
```

Výpis 6: Arbiter.Choice

2.3.6.3 Arbiter.JoinedReceive Dalším příkladem arbitra je Arbiter.JoinedReveive, který je podobný logickému `AND`. Funkčnost je obdobná předešlému typu, tzn. opět arbitr vytvoří dva příjemce, kteří čekají na příchod zprávy zvoleného typu. Narozdíl od předchozího typu se tento neukončí po spuštění prvního, ale vyčká se na spuštění i druhého, navíc přijímá jako vstupní parametry dvě instance řídy `Port`. Po přijetí obou typů zpráv se naplánuje úloha, která se má vykonat. Tento druh je vhodný použít v místě, kde potřebujeme synchronizaci kódu, např. když se dvě větve aplikace mají sloučit v jednu.

```
Port<int> intPort = new Port<int>();
Port<double> doublePort = new Port<double>();

Activate(
    Arbiter.JoinedReceive(false, intPort, doublePort,
    delegate(int val1, double val2)
    {
        Console.WriteLine("Both values received.");
    }
```

Výpis 7: Arbiter.JoinedReceive

2.3.7 Iterátory

Jedním z velice důležitých prvků k programování operací a služeb v asynchronním prostředí je iterátor, který je zahrnut do knihovny CCR při použití jazyka C#. Konstrukce

iterátor nám umožňuje spouštění kódu bez blokování ostatních vláken, které mohou být spuštěny během čekání na dokončení nějaké asynchronní operace. V podstatě čeká na přijetí zprávy před tím, než může pokračovat ve vykonávání.

Iterátory jsou v rámci CCR založeny na sekvenci úloh, které mají být vykonány. Z programátorského pohledu vypadají, jako kdyby se vše spouštělo sekvenčně za sebou, ale ve skutečnosti, dochází k rozdělení na několik asynchronních úloh.

Abychom mohli využít iterátor uvnitř metody, musí být nadeklarována s návratovým typem `IEnumerator<ITask>`. V místě, kde dochází dochází k procházení sekvence úloh iterátoru musíme používat konstrukci `yield return`. V takto označeném místě bude docházet k přerušení vykonávání metody, pokud bude nutné čekat na příchod zprávy. Na konci metody ve které se využívá iterátor, je nutné uvést konstrukci `yield break;`, která oznamuje, že procházení iterátorem je u konce.

```
public IEnumerator<ITask> IteratorExample() {

    yield return Arbiter.Receive(false, simplePort,
    delegate(int i)
    {
        Console.WriteLine("Cislo i: {0}", i);
    }

    yield break;

}
```

Výpis 8: Iterátor

Jak je vidět z výpisu 8 největší uplatnění koncept iterátorů nalezne u vytváření příjemců, nejčastěji při čekání na odpověď z operace nějaké služby nebo čekání na přečtení údajů ze senzorů atd. Jednoduše můžeme říct, že iterátory použijeme všude tam, kde by došlo ke zbytečnému zastavení vykonávaní vlákna z důvodu čekaní na dokončení jiné operace.

Zbývá ještě doplnit, jak takový iterátor spustit. Nejjednodušší možnost je spustit iterátor pomocí příkazu `SpawnIterator`, který je obdobou příkazu `Spawn` pro spuštění jednoduché úlohy. Další možností je spustit iterátor jako úlohu pomocí arbitra. Oba tyto případy jsou uvedeny v krátkém výpisu 9

```
// Spusteni iteratoru
SpawnIterator(IteratorExample);
Activate(
    Arbiter.FromIteratorHandler(IteratorExample));
```

Výpis 9: Dvě možnosti spuštění iterátoru

Na závěr bych k iterátorům poznamenal, že je samozřejmě možné, stejně jako u běžné úlohy, poslat i vstupní parametry pomocí generického typu a hodnoty. Iterátorům je velice důležité porozumět, jelikož se s jejich pomocí vytváří velké množství kódu, který by jinými způsoby bylo velice složité napsat.

2.3.8 Dispatcher

V tuto chvíli jsme prošli cyklus od poslání zprávy až k vytvoření a naplánování úlohy. Poslední část, kterou se budeme zabývat je stejně důležitá jako všechny předešlé je `Dispatcher` a `DispatcherQueue`. `Dispatcher` není ve své podstatě nic jiného než zásobník vláken a můžeme ho směle přirovnat k procesoru počítače. Vlákna inicializována uvnitř třídy `Dispatcher` slouží ke spouštění naplánovaných úloh. Každý `Dispatcher` může mít libovolné množství instancí třídy `DispatcherQueue`.

Po vygenerování nové služby, jsou všechny věci kolem inicializace tříd `Dispatcher` a `DispatcherQueue` vyřešeny ve společném předkovi všech služeb `DsspServiceBase` a ve většině případů není nutné vůbec do tohoto standardního nastavení vůbec zasahovat. Základní nastavení odvozuje MRDS z počtu procesorů, respektive z počtu jader, pokud máme více jádrový procesor. Minimální počet vláken je dva, takovýto `Dispatcher` se vytvoří pokud v konstruktoru zadáme počet vláken 0. Pro speciálně případy je možné vytvořit `Dispatcher`, který bude mít pouze jedno vlákno, to znamená, že v jednu chvíli se bude zpracovávat pouze jedna úloha.

Jedna z možností, která není úplně využívána, je možnost vytvořit více instancí třídy `Dispatcher`. Bohužel tato zmíněná možnost nenabízí téměř žádné praktické využití. Jediná příležitost, u které by se toto dalo využít, je mít instanci `Dispatcher` pro vlákna různých priorit.

Třída `DispatcherQueue` slouží k ukládání naplánovaných úloh, její maximální velikost je rovna maximu datového typu `int`. Fronta implementuje několik politik pro ukládání zpráv. Standardní politika ukládá zprávy bez jakéhokoliv omezení. Další politiky jsou založeny na omezeních vycházejících z parametrů dané fronty.

Na závěr kapitoly o třídách `Dispatcher` a `DispatcherQueue` ještě malá ukázka na vytvoření a použití těchto tříd, ale jak již bylo zmíněno na začátku, není příliš časté ani nutné vytvářet vlastní instance zmíněných tříd.

Ve výpisu 10 můžeme vidět vytvoření nové instance třídy `Dispatcher`, která je v tomto případě inicializovaná se čtyřmi vlákny pro zpracování úloh. Po té je vytvořena fronta pro příjem úloh a je jí přiřazen Dispatcher, který bude obsluhovat úlohy v ní uložené. Nakonec je vytvořen pomocí arbitra seznam operací, které budou ukládat své úlohy do nově vytvořené fronty.

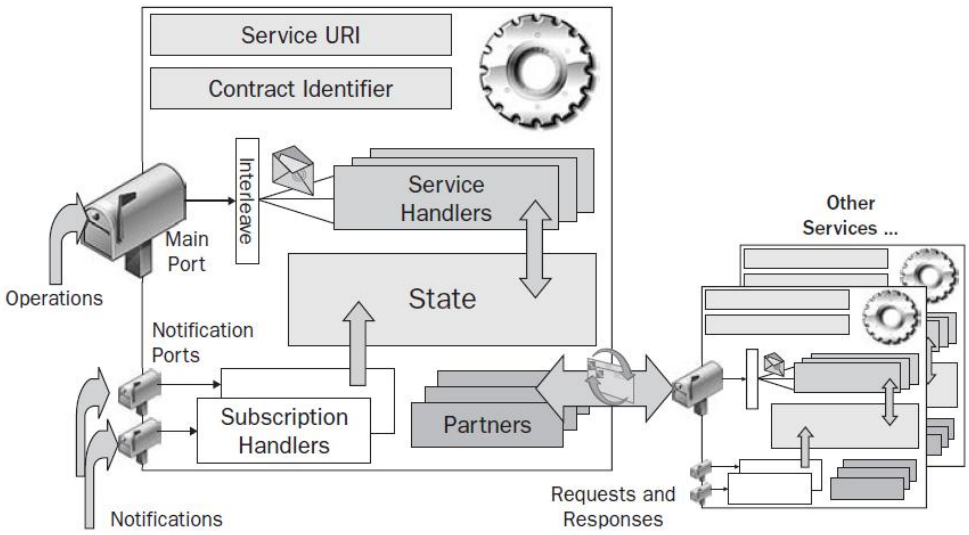
```
public void CreateDispatcher()
{
    Dispatcher d = new Dispatcher(4, "TestDispatcher");
    DispatcherQueue dq = new DispatcherQueue("TestQueue", d);

    Arbiter.Activate(
        dq,
        Arbiter.FromIteratorHandler(IteratorExample),
        Arbiter.FromIteratorHandler(IteratorExample)
    );
}
```

Výpis 10: Použití tříd `Dispatcher` a `DispatcherQueue`

2.4 Služba

Jak už napovídá architektura MRDS, základními prvky pro všechny aplikace v rámci platformy jsou služby, které jsou postaveny na základech webových služeb. Aby služby mohly komunikovat mezi sebou a vytvářet celky, musí mít nadefinovány několik vlastností. Základem každé služby je kontrakt, který nám identifikuje službu, hlavní operační port služby, do kterého přicházejí zprávy z jiných služeb. Dále je to stav dané služby a nakonec také jednotlivé metody nebo operace služby, pomocí kterých služby komunikují se svým okolím. Součástí služby může být také libovolné množství partnerů, čili jiných služeb, se kterými služba komunikuje. Na obrázku 2 můžeme vidět, jak jednotlivé součástí služby spolupracují.

Funkčnost jednotlivých služeb je postavena na knihovně CCR, představené v předchozí kapitole, a knihovně DSS, která se stará o běh a podporu služeb, ta bude představena v další kapitole.



Obrázek 2: Vizualizace služby. Zdroj: [1]

2.4.1 Kontrakt

Kontraktem rozumíme jedinečný identifikátor služby, který nám a hlavně MRDS pomáhá rozlišit jednotlivé služby od sebe. Každá služba musí obsahovat třídu `Contract`, aby ji bylo možno používat v rámci MRDS. Navíc tato třída musí obsahovat pole `Identifier` s hodnotou ve formátu URI, která musí být unikátní. V následujícím příkladu můžeme vidět standardně vygenerovaný kontrakt pro službu `RobotExploration`, tento identifikátor se nám vygeneruje pokud nezadáme název identifikátoru při vytváření služby.

```
public sealed class Contract{

    [DataMember]
    public const String  Identifier  = "http :// schemas.tempuri.org/2010/02/robotexploration.html";

}
```

Výpis 11: Příklad kontraktu

V identifikátoru kontraktu můžeme použít jakékoliv doménové jméno, nebo identifikátor cesty, je to čistě na nás, jelikož se nejedná o reálné URL, které by mělo být funkční. Identifikace pomocí roku a měsíce nám zvyšuje jedinečnost názvu a navíc může sloužit k verzování služeb. Poslední část identifikátoru tvoří název služby.

Každá služba může implementovat alternativní kontrakt, tzn. kontrakt který už využívá jiná služba, ale chceme překrýt její chování naší implementací nebo chceme, aby i naše služba vystupovala pro jiné služby jako služba definovaná alternativním kontraktem. Nejčastěji se toto používá při vytváření simulovaných robotů, kdy se definuje třída pro pohon robota, která využívá kontraktu služby `DifferentialDrive`. Nebo další příklad je ultrazvukový senzor pro LEGO NXT, který využívá jako alternativní kontrakt analogový senzor.

2.4.2 Hlavní operační port

Každá služba má vždy nadefinován svůj hlavní operační port, který definuje jaké typy zpráv služba přijímá a na které může odpovídat. Tento hlavní operační port doplňuje kontrakt programově a definuje rozhraní služby. V tomto portu se shromažďují požadavky ostatních služeb.

Následující příklad zobrazuje základní definici hlavního operačního portu, tuto definici získáme při vygenerování služby. Obsahuje pouze tři typy zpráv resp. operací, které nabízí. První dvě operace, které jsou získány dědičností z předka, slouží k získání služby a k ukončení služby. Třetí typ zpráv, na které tato služba reaguje je typ operace GET.

```
[ServicePort]
public class RobotExplorationOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get>
{
}
```

Výpis 12: Příklad hlavního operačního bodu

2.4.3 Stav služby

V každé službě nalezneme třídu, která reprezentuje vnitřní stav dané služby. Vlastnosti stavu můžeme rozdělit na dvě skupiny, do první patří vlastnosti, které se během doby běhu služby nemění. Jsou to většinou konstanty, které jsou nutné k běhu služby např. COM port, Polling frekvence (hodnota určující, jak často se budou načítat data ze senzorů)

atd. Druhou skupinou jsou vlastnosti, které se mění v průběhu času. Pokud je služba navrhnuta jako konečný stavový automat, může být v proměnné uložen aktuální stav tohoto automatu.

```
public enum STAGE { MOVEFORWARD, MOVEDONE, ROTATIODONE };
public enum ROTATESTAGES { NORMAL, RIGHT90, RIGHT45, LEFT90, LEFT45 };

[DataContract]
public class RobotExplorationState
{
        [DataMember]
        public STAGE Stage;

        [DataMember]
        public ROTATESTAGES RotateStage;
}
```

Výpis 13: Příklad stavů služby.

2.4.4 Operace služby

Aby dávala služba smysl, musí umět komunikovat se svým okolím. Tato komunikace se děje prostřednictvím operací služby. Jak již bylo zmíněno dříve, každá služba obsahuje už po svém vygenerování tři základní operace. `DssDefaultLookup` operace zajišťuje možnost vyhledat službu a poskytnout její kontext. `DssDefaultDrop` operace zajišťuje zpracování požadavku na ukončení služby. A nakonec operace typu `GET`, která poskytuje předání kopie stavu služby a je pouze ke čtení. Služba může obsahovat mnoho operací, typy jednotlivých operací najdeme v tabulce 1.

2.4.4.1 Struktura operace . Definice operace je složena z několika částí a je uložena v souboru s příponou Type. Nejprve je nutno určit název operace a její typ, poté můžeme nadefinovat obsah požadavku operace, ve kterém jsou určené vstupní hodnoty operace. Je to stejné jako atributy metody v běžném programu, akorát u služby je toto nadefinováno samostatnou třídou. A nakonec můžeme nadefinovat návratové hodnoty operace. Nejčastěji se používá třída `PortSet`, ve které můžeme nadefinovat více typů návratových hodnot. Pokud potřebujeme předat výsledek operace, prostřednictvím odpovědi operace můžeme vytvořit opět samostatnou třídu, stejně jako v případě požadavku. Některé typy operací, jako `Submit` nebo `Update`, umožňují použít předdefinovanou odpověď jako výsledek operace, která nepředává žádné informace jako výsledek operace. Můžeme si to představit jako metodu s klíčovým slovem `void`. V následujícím příkladu je operace `MoveOrder`, která je typu `Submit` a přijímá jako vstup třídu `MoveOrderRequest` a vrací `DefaultSubmitResponseType` nebo `Fault` pokud nastane během provádění operace chyba.

Operace	Popis
Create	Vytvoření nové služby
Delete	Vymazání části nebo celého stavu služby
Drop	Operace k zastavení služby
Get	Operace k získání stavu služby
Insert	Operace k přidání informace o stavu služby
Lookup	Operace k získání informace stavu služby a jejího kontextu. Používá se k získání přístupu ke službě.
Query	Strukturovaná obdoba operace Get
Replace	Operace k nahrazení celého stavu služby
Subscribe	Operace k vyžádání informování o změnách stavu
Submit	Obdoba operace Update, nemusí během ní dojít ke změně stavu
Update	Operace, která informuje o změně stavu služby
Upsert	Operace vyvolá operaci Update pokud existuje stavová informace, jinak vyvolá operaci Insert

Tabulka 1: Typy operací služeb

```
public class MoveOrder : Submit<MoveOrderRequest, PortSet<DefaultSubmitResponseType,
    Fault>>
{
        public MoveOrder() { }

        public MoveOrder(MoveOrderRequest body)
            : base(body)
        { }
}

[DataContract]
[DataMemberConstructor]
public class MoveOrderRequest
{
        [DataMember, DataMemberConstructor]
        public double Power;

        [DataMember, DataMemberConstructor]
        public double Distance;

        public MoveOrderRequest() { }
}
```

Výpis 14: Příklad definice služby. RobotExploration.cs

Aby služba mohla obsloužit nově nadefinovanou operaci, musíme jí přidat do definice hlavního operačního portu.

```
[ServicePort]
public class RobotExplorationOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get,
    MoveOrder>
{
}
```

Výpis 15: Přidání operace `MoveOrder` do hlavního operačního portu

Pro takto nadefinovanou operaci už můžeme v hlavním souboru služby vytvořit samotnou obsluhu a tělo operace.

2.4.4.2 Obsloužení operace . V předchozí části jsme si nadefinovali rozhraní služby a přídáním operace do hlavního operačního portu jsme umožnili příjem zpráv typu `MoveOrder`. K takto nadefinované metodě potřebujeme vytvořit jěště její tělo, které bude umět obsloužit požadavky na tuto operaci. Opět existuje několik možností jak takovouto obsluhu operace vytvořit. Nejjednodušší cesta je vytvořit virtuální metodu s anotací, která říká, že se jedná o obsluhu operace.

```
[ServiceHandler]
public virtual IEnumerator<ITask> MoveOrderHandler(MoveOrder moveOrder)
{
    /// Obsluha operace

    moveOrder.ResponsePort.Post(DefaultSubmitResponseType.Instance);

    yield break;
}
```

Výpis 16: Obsluha služby pomocí anotace

2.4.4.3 Chování obsluhy operace . Pokud potřebujeme ovlivnit chování obsluhy operace, můžeme to udělat pomocí zařazení operace do některé z kategorií uvedených v tabulce 2.

Pokud používáme obsluhu služby vytvořenou pomocí anotace, stačí pouze nadefinovat chování operace pomocí atributu anotace `ServiceHandler` takto:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> MoveOrderHandler(MoveOrder moveOrder)
```

Výpis 17: Definice chování operace

2.4.5 Partneři služby

Pokud služba komunikuje s dalšími službami, nazýváme tyto služby partnery. Aby s nimi služba mohla komunikovat, potřebujeme nadefinovat seznam partnerů služby. Můžeme

Concurrent	Obsluha může běžet soucasně s dalšími operacemi takto označenými popř. může být sama spuštěna několikrát. Takto by se měly označovat obsluhy, které neměnní stav služby
Default	MRDS vybere jaký typ je nejvhodnější. Vybírá z `Concurrent`, `Exclusive` a `TearDown`
Eclusive	Obsluha běží samostatně a během jějího průběhu nemůže být spuštěna jiná operace označená `Concurrent` nebo `Exclusive`.
Indepedent	Obsluha běží nezávisle na ostatnitních obsluhách
TearDown	Obsluha operace, která je spuštěna během ukončování služby

Tabulka 2: Parametry chování operace

to udělat pomocí tagu `PartnerList` nebo jednotlivé partnery nadefinovat přímo v hlavní třídě služby.

```
[Partner("drive", Contract = drive.Contract.Identifier,
        CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
drive.DriveOperations _drivePort = new drive.DriveOperations();
```

Výpis 18: Partner drive

Jak můžeme vidět ve výpisu 18 definice partnera obsahuje název partnera, jeho kontrakt a politiku pro vytvoření partnera. Jednotlivé politiky pro vytváření jsou v tabulce 3. Důležitou součásti definice partnera je nadefinování portu, přes který bude probíhat komunikace s danou službou.

CreateAlways	Vytvoří novou instanci partnerské služby
UseExisting	Použije existující instanci služby z adresáře služby
UseExistingOrCreate	Použije nebo vytvoří instanci partnerské služby
UsePartnerListEntry	Použije existující instanci z adresáře služby. Tato instance je vytvořena pomocí definice v manifestu

Tabulka 3: Politiky pro vytváření partnera služby

2.4.6 Nová služba

Novou službu můžeme vytvořit několika způsoby. První možnost je vygenerovat službu pomocí utility `DssNewService`, která je součástí MRDS. Další možností je vytvořit službu prostřednictvím Visual Studia, nebo také pomocí VPL, pokud používáme k vytváření aplikace grafický programovací jazyk, který je součástí MRDS. V tabulce 4 vidíme základní parametry utility `DssNewService`.

Název	Popis
/service:	Definuje název služby. Povinný parametr
/namespace:	Definuje namespace služby
/org:	Prefix služby, který je použit pro vytvoření kontraktu.
/year:	Rok vytvoření služby
/month:	Měsíc vytvoření služby
/alt:	Alternativní kontrakt, který služba implementuje

Tabulka 4: Základní parametry pro `DssNewService.exe`

Službu s kontraktem `http://test.com/service/2010/04/testservice.html` vygenerujeme pomocí utility `DssNewService` takto:

```
dssnewservice /service:TestService /org:test.com/servive /year:2010 /month:04
```

Výpis 19: Ukázka vygenerování nové služby

2.5 DSS

V předchozí kapitole jsme si představili služby jako základní prvek aplikací ovládajících roboty. V této kapitole se podíváme na knihovnu, kterou si můžeme představit jako kontejner pro běh vytváření a běh služeb. Tato knihovna se jmenuje DSS - Decentralized Software Services. Součástí této kapitoly by měly být i služby, ale pro větší přehlednost jsem službám vyčlenil samostatnou kapitolu.

Stejně jako služby je DSS postaveno na základě knihovny CCR. Pro všechny služby existuje společný předchůdce, kterým je třída `DsspServiceBase`. DSS je zodpovědné za spouštění a ukončovaní služeb, jednou z dalších funkcí DSS je rozesílaní zpráv mezi službami. Ve své podstatě DSS je pouze uskupení několika služeb, které spolu komunikují. Jednotlivé služby se starají o načítání konfigurací služeb, řízení bezpečnosti, udržování adresáře běžících služeb atd.

2.5.1 DSSP

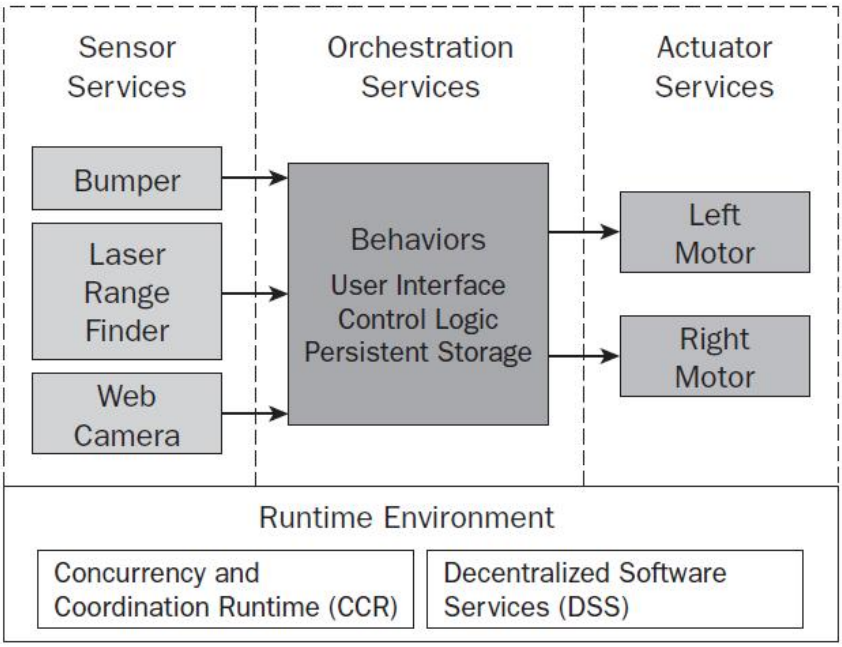
Komunikačním protokolem pro DSS je příznačně pojmenovaný DSSP, jehož specifikaci můžeme volně používat pod licencí Microsoft Open Software Promise. Celý protokol je založen na architektuře REST (Representational State Transfer). Protokol DSSP je obdobný protokolu SOAP (Simple Object Access Protocol), který používají standardní webové služby. Avšak DSSP přináší některé změny, jako například oddělení stavu odchování nebo všechny operace jsou reprezentovány jako stavové operace. Dalším rozdílem je viditelnost stavových informací, u webových služeb jsou neviditelné, kdežto u služeb MRDS je základem, že stavové informace služby jsou viditelné.

Klíčovou vlastností obou modelů, SOAP i DSSP je, že události jsou oznamovány asynchronně, což je velice nutná vlastnost pro programování robotických aplikací.

2.5.2 Spojování služeb

Stejně jako je základním stavebním blokem aplikací služba, tak spojování služeb je základním principem vytváření aplikací. Služby, které spolu komunikují nazýváme partnery a celý proces spojování se nazývá orchestrace služeb. Tímto principem jsme schopni vytvořit aplikaci z jednotlivých služeb od nejnižší po nejvyšší úroveň.

Služby můžeme rozdělit do tří základních kategorií. První tvoří služby obsluhující senzory služeb, tyto předávají informace výš do služeb, které nazýváme orchestrační služby. V těchto službách dochází ke zpracování dat, ke zpracování logiky, k obsluze uživatelského rozhraní. Poslední skupinou služeb jsou aktuátory, které se starají o vykonávání akcí na základě řídící logiky. Do této kategorie služeb spadají nejčastěji služby obsluhující pohon robotů, popř. jiné funkční jednotky. Příklad takto komunikujících služeb můžeme vidět na obrázku 3.



Obrázek 3: Orchestrace služeb. Zdroj: [1]

2.5.3 DssHost

Ke spouštění aplikací vytvořených v prostředí MRDS využíváme program `DssHost`. Spuštěním tohoto programu vytvoříme tzv. `DSS uzel`, který poskytuje potřebné běhové prostředí pro spouštění aplikací a služeb potřebných k jejich běhu.

`DssHost` je v podstatě implementace webového serveru, který umožňuje zobrazování běžících služeb a jiných informací ve webovém prohlížeči. Tato vlastnost nám umožňuje jednoduše komunikovat se službami bez nutnosti mít speciálního klienta. Společně se

startem uzlu se nastartují i některé základní služby, které se starají o fungování aplikace. Dále se instalační adresář označí jako přípojný bod, respektive je označen jako kořenový. Ke všem souborům a složkám v kořenovém adresáři je možno přistupovat jako k lokálním. Navíc všechny služby spuštěné v daném uzlu jsou namapovány právě do kořenového adresáře.

Ke spuštění aplikace `DssHost` potřebujeme dva porty, avšak tyto porty nemají nic společného se třídou `Port` z knihovny `CCR`. Ale jedná se porty určené ke komunikaci. První port je určen pro komunikaci skrz HTTP protokol s hodnotou 50000. Druhým komunikačním kanálem je port s hodnotou 50001, který je určen ke komunikaci skrz DSSP ovládající služby. Dalším podstatným parametrem při spouštění `DSS uzlu` je určení manifestů, které mají být spuštěny. K samotnému spuštění aplikace nám stačí nadefinovat jako parametr jediný sestavující manifest. Pokud však používáme reálného robota připojujeme ke spuštění ještě manifest popisující použitý hardware. Výpis 20 ukazuje jak spustit službu.

```
DssHost /port:50000 /tcpport:50001 /manifest:"RobotExploration/RobotExploration.manifest.xml"
    /manifest:"samples/explorer/Explorer.manifest.xml"
```

Výpis 20: Příklad spuštění DSS uzlu

2.6 Manifesty

Ke konfiguraci a popisu objektů v rámci MRDS se používají soubory zvané manifesty. Jsou to textové soubory založené na standardu XML. Nejčastěji se manifesty používají ke konfiguraci. Základní sadu manifestů, které jsou dostupné po instalaci MRDS, je možné nalézt v adresáři `samples\config`. K těmto základním si můžeme vytvořit další pro potřeby našich služeb. Manifesty můžeme rozdělit do několika skupin.

2.6.1 Sestavující manifesty

Pro první kategorii jsem vybral množinu manifestů, které jsem nazval sestavující. Jsou to soubory které mají koncovku `.manifest.xml`. Pomocí těchto souborů se dají nadefinovat služby do jednoho celku. Další je možnost nadefinovat si hardware, který bude reálně využíván. Takovýto manifest musí obsahovat každá služba jelikož pomocí něj je spuštěna. Tyto soubory jsou procházeny od shora dolů, to znamená, že služby se startují v pořadí v jakém jsou nadefinované v manifestu. Jak vypadá základní manifest po vygenerování je možné vidět ve výpisu 21. Takovýto manifest spustí pouze službu `testservice`.

```
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    >
    <CreateServiceList>
        <ServiceRecordType>
            <dssp:Contract>http://test.com/service/2010/04/testservice.html</dssp:Contract>
```

```
        </ServiceRecordType>
    </CreateServiceList>
</Manifest>
```

Výpis 21: Ukázka základního manifestu

Pokud před spuštěním aplikace známe všechny služby, které budeme používat a mají být spuštěny současně s hlavní službou, je nejvhodnější použít právě manifesty. Nepotřebujeme totiž přidávat do služeb kód, který by se staral o spouštění služeb a získávání jejich kontextu.

Pokud bude naše služba komunikovat s reálným robotem můžeme si pro něj připravit manifest. Nejprve musíme mít připraven manifest s definicí služeb, které budou obsluhovat hardwarovou jednotku. A teprve po té můžeme pomocí proxy tříd k nim přistoupit. Vše bude ukázáno v kapitole věnující se vytváření a úpravě manifestů.

2.6.2 Konfigurační manifest

Dalším typem manifestu jsou soubory s koncovkou `.config.xml`, které slouží k ukládání informací o službě. Používají se k ukládání stavových informací služby. Běžně je můžeme vidět u služeb, které potřebují mít nastavenou počáteční konfiguraci. Nejčastěji jsou vidět u služeb, které pracují s reálným hardwarem, jelikož obsahují informace potřebné k připojení nebo jiné parametry, které jsou potřebné pro nastavení. Jméno souboru je vždy odvozeno od názvu služby, pro kterou obsahuje nastavení. Např. pro službu `legonxtbrickv2`, která se stará o přípojení k jednotce LEGO NXT, je konfigurační soubor pojmenován `legonxtbrickv2.config.xml`. Ve výpisu 22 můžeme vidět jak vypadá konfigurace pro službu `legonxtbrickv2`.

```
<NxtBrickState xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:wsa="http://schemas.
    xmlsoap.org/ws/2004/08/addressing" xmlns:d="http://schemas.microsoft.com/xw/2004/10/dssp.
    html" xmlns="http://schemas.microsoft.com/robotics/2007/07/lego/nxt/brick.html">
  <Configuration>
    <SerialPort>16</SerialPort>
    <BaudRate>115200</BaudRate>
    <ConnectionType>Bluetooth</ConnectionType>
    <ShowInBrowser>true</ShowInBrowser>
  </Configuration>
</NxtBrickState>
```

Výpis 22: Ukázka konfigurace pro službu `legonxtbrickv2`

2.6.3 Popis simulační scény

Posledním manifestem je soubor s koncovkou `.SimulationEngineState.xml`, ve kterém je popsána scéna ze simulátoru. V podstatě se jedná o popis všech objektů, které jsou součástí simulace. Stěny, podlahy, gravitace atd., pro to vše můžeme najít záznam v takovémto typu manifestu. Samozřejmě nechybí popis vlastností obsažených objektů. Takovéto soubory jsou velice obsáhlé, proto je není jednoduché upravovat ručně, obsahují

několik tisíc řádek. Jedna z možností, jak takový manifest získat, je spustit prostředí VSE a scénu si vytvořit a po té uložit pomocí příslušných funkcí.

2.6.4 Tvorba a editace manifestů

Jelikož manifesty jsou obyčejné soubory založené na standartu `XML`, můžeme takové soubory vytvářet a editovat pomocí obyčejného textového editoru. V takovém případě potřebujeme však znát všechny možnosti nastavení jednotlivých služeb, jejich kontrakty atd. Pokud bychom chtěli vytvářet scény pro simulaci, tak nás čeká napsat několik tisíc řádek. Proto společnost Microsoft přináší v rámci prostředí MRDS možnosti, jak takové manifesty vytvořit.

Jedna z možností, jak vytvořit manifest, je vygenerovat službu pomocí utility `DssNewService` nebo pomocí MS Visual Studia, pokud k vývoji používáme programovací jazyk C# nebo jiný podporovaný ve VS. Takto však získáme pouze základní manifest, který pak musíme upravovat sami ručně. Což by pro konfiguraci vlastních služeb nemělo být obtížné.

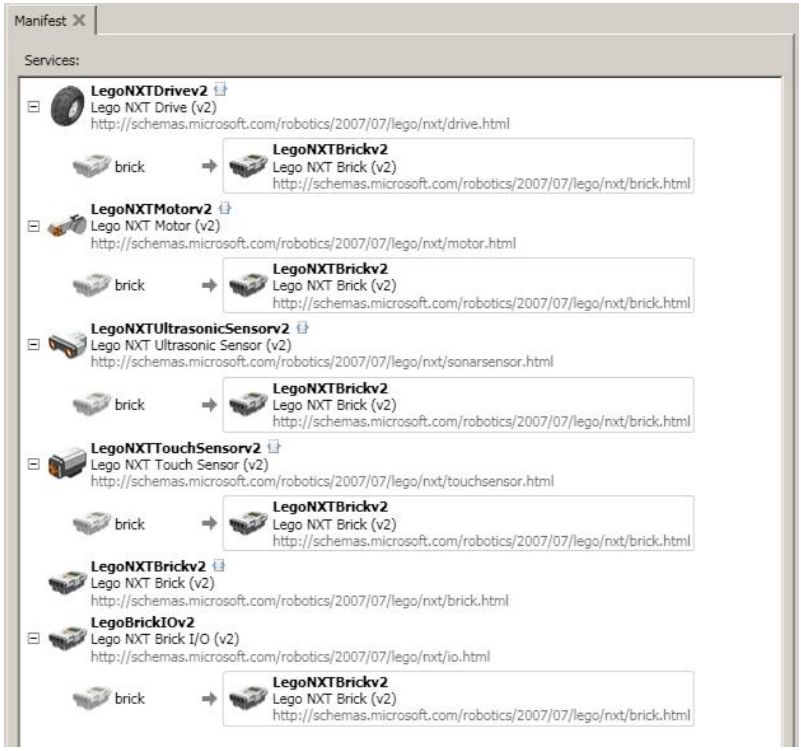
Nejlepší nástroj pro vytváření a úpravu manifestů je nástroj DSSME, který je součástí instalace platformy MRDS. Umožňuje vytvářet a editovat sestavovací a konfigurační manifesty služeb. Při vytváření nových manifestů jsme omezení na využití již existujících služeb, ze kterých můžeme vybudovat nový manifest. U služeb, které jsou konfigurovatelné, umožňuje provést nastavení jednotlivých parametrů. Jediný problém, na který jsem narazil u tohoto nástroje byl, že není možné přenášet manifesty mezi jednotlivými verzemi MRDS.

Jako příklad využiji sestavovacího manifestu, vytvořeného v editoru DSSME pro účely konfigurace reálného robota sestaveného ze stavebnice LEGO NXT. Na obrázku 4 můžeme vidět grafické zobrazení manifestu, který obsahuje několik služeb zastupující obsluhu skutečného hardwaru.

Když máme připraven manifest, můžeme začít používat takto nakonfigurovaného robota ve svých službách, jelikož jednotlivé prvky jsou prezentovány jako služby, které mají kontrakty. Pokud tedy zajistíme, že jednotlivé služby tohoto manifestu budou spuštěny, můžeme se k nim připojit jako k partnerům.

2.7 VPL - Grafické programování robotů

Alternativní možností, jak vytvářet programy v prostředí MRDS, a zároveň nejjednodušší je využít nástroje VPL, který je součástí jeho instalace. Jedná se grafický programovací nástroj, který využívá všechny možnosti platformy MRDS. Programování probíhá jednoduše metodou drag and drop, kdy si uživatel vybírá ze služeb, které jsou k dispozici v MRDS. Tyto služby musí být zkompilovány v podobě knihovny `DLL` v instalačním adresáři `MRDS\bin`. Tento nástroj je určen pro lidi, kteří neumí programovat v žádném programovacím jazyce, ale chtějí vyvíjet základní aplikace pro roboty. Ale stejně tak dobrý může být i pro profesionály, kteří v něm mohou rychle vytvářet jednoduché služby, jelikož výsledné služby je možné exportovat do jazyka C#.



Obrázek 4: Sestavovací manifest pro hw robota

Aplikace, respektive služby, můžeme skládat z různých bloků, které se obecně nazývají aktivity. At' už to jsou bloky, které představují služby jako na obrázku 5 nebo bloky, pomocí kterých můžeme rozšířit logiku služeb. Bloky služeb mohou představovat služby od nejnižší úrovně (služby obsluhující hardware), až po služby s komplexní logikou. Jednotlivé bloky jsou propojeny pomocí "linek", reprezentujících přenos zpráv mezi bloky.

Obrázek 5: Ukázka bloku, představujícího službu

Na obrázku 5 je vidět blok služby starající se o pohonnou jednotku, na její levé straně je šipečka mířící do služby, která symbolizuje příchozí požadavky na obsloužení danou službou. Naopak na její druhé straně jsou dva symboly, které znamenají výstup ze služby. První je šipka směřující ven, která symbolizuje odpověď na volanou operaci. Druhý symbol reprezentuje oznámení, nejčastěji o změně stavu služby nebo dokončení požadované operace. Vlastnosti jednotlivých aktivit můžeme nastavit přímo v editoru VPL, pomocí kliknutí na blok aktivity a nastavení jejich parametrů. Všechny konfigurace včetně všech ostatních údajů, jsou uloženy v podobě manifestů.

Stejně jako služby, můžeme i jednotlivé diagramy vytvořené pomocí VPL, skládat do sebe. To znamená, že blok aktivity může skrývat jiný diagram. Toto je vhodné využívat, pokud vytváříme složitější služby, které mohou obsahovat velké množství aktivit a snadno bychom se v nich ztratili.

2.8 VSE - Simulované prostředí

Poslední důležitou částí, kterou získáme nainstalováním prostředí MRDS, je VSE - Visual Simulation Environment. Jedná se o simulátor 3D prostředí se zachováním fyzikálních zákonů. Ve VSE je možné vytvářet simulované scény jak pro vnitřní tak i pro venkovní prostory. Hned po instalaci můžeme využít několika předpřipravených scén.

V základní instalaci se nachází několik základních simulovaných robotických jednotek, které mohou být doplněny o naše vlastní. Jak sestavit vlastního robota je popsáno v kapitole **??** věnující se implementaci. Základními jednotkami jsou LEGO NXT, iRobot Create a MobileRobots Pioneer 3DX.

Simulátor najde své uplatnění v mnoha případech. Můžeme tím ušetřit náklady na nákup robota nebo můžeme zabránit jeho zničení nesprávně napsaným programem. Může simulovat nové algoritmy, které chceme přidat k ovládání robota, respektive simulátor může fungovat jako testovací prostředí pro vývoj robotických aplikací. Navíc můžeme

vyvíjet aplikace pro roboty, kteří ještě např. nejsou vyrobeni. Samozřejmě nemůže simulátor nahradit skutečné prostředí, jelikož ne vše je v reálném světě tak ideální, jako v tom simulovaném.

3 LEGO NXT Mindstorms

3.1 Popis

LEGO NXT je programovatelná stavebnice společnosti LEGO, pomocí které je možné sestavit mnoho druhů robotů. NXT verze nahrazuje svého předchůdce s označením Robotics Inveton Systems nebo také RCX. NXT je vydáváno v aktuální verzi 2.0. Pro tuto práci byla použita verze 1.0, která obsahuje řídící kostku, 4 senzory a 3 servo motory.

3.2 NXT Kostka

Základní komponentou je řídící kostka, která je mozkem celé stavebnice. Ve své podstatě je to malý počítač, který může mít připojené 4 senzory a tři motory, pomocí kabelů RJ12. Kostka komunikuje s uživatelem skrz monochromatický displej o velikosti 100 x 64 pixelů a 4 navigačních tlačítek. Kostka je zároveň schopná přehrávat zvuky. K napájení slouží šest baterek AA nebo je možno dokoupit dobíjecí Li-Ion baterii.

Se svým okolím kostka komunikuje prostřednictvím portu USB, který se nejčastěji používá k nahrávání nového firmwaru. Důležitějším kanálem je integrované rozhraní Bluetooth v2.0, pomocí kterého kostka může komunikovat až se třemi zařízeními současně.

Z technického hlediska je kostka poháněna 32-bitovým procesorem ARM7 pracujícím na frekvenci 48 MHz s 256 KB Flash pamětí a 64 KB RAM. Pro programy je možno využít necelých 100 KB paměti.

Obrázek 6: NXT Kostka. Zdroj: [4]

3.3 NXT a komunikace pomocí Bluetooth

Tento text je založen na článku [2], který jsem nalezl při hledání řešení problému s komunikací mezi programem běžícím na PC a robotem, který bude popsán v kapitole popisující řešení.

Robot respektive řídící kostka je schopna komunikovat se třemi dalšími zařízeními najednou. Nás bude ovšem zajímat pouze komunikace NXT a PC. V komunikaci je vždy jeden prvek Master a druhý Slave. Ten kdo vytváří komunikaci je vždy Master a stará se o řízení komunikace mezi komunikujícími stranami. Stará se o vyzvedávání zpráv od podřízené strany.

Zprávy mezi jednotlivými prvky komunikace jsou posílány skrz poštovní schránky, kterých NXT obsahuje 10 (V NXT-G číslovány 1 - 10). Každá schránka může obsahovat 5 zpráv, pokud dojde k tomu, že schránka je plná, nejstarší zpráva je přepsána novější což může a při některých úlohách způsobuje velké problémy.

Existují dvě možnosti komunikace. Jedna kdy, jsou skrz Bluetooth posílány přímo příkazy, které mají být vykonány kostkou, ať už to spuštění je spuštění pohybu nebo čtení ze senzoru. Druhou možností je zasílání jen dat mezi dvěma programy. Vzhledem k tomu, že kostka musí přepínat mezi odesíláním a přijímáním, přičemž přepínaní se provádí velice pomalu, podle dokumentace [3] toto přepnutí trvá kolem 30 ms. Z čehož v kombinaci s omezením zmíněném v předchozím odstavci vyplývá, že není možné provozovat velice intenzivní komunikaci a je nutné hledat techniky, které zamezí přetečení schránky, pokud potřebujeme znát obsah všech zpráv.

3.4 NXT Sensory

LEGO NXT obsahuje v balení několik základních senzorů, které můžeme použít při stavbě robota. Jsou to dotykový, ultrazvukový, světelný a zvukový. K těmto základním existuje řada dalších senzorů, které je možné dokoupit a tak rozšířit možnosti NXT. Důležité je poznamenat, že pro většinu doplňujících senzorů existují manifesty pro použití v prostředí MRDS.

3.4.1 Dotykový senzor

Jedná se o obyčejné čidlo, které reaguje na dotek, a to jak na zmáčknutí tak na uvolnění. Je to základní senzor, který je používán k rozpoznávání kolizí robota s nějakou překážkou.

3.4.2 Ultrazvukový senzor

Senzor založený na principu ultrazvuku, který pomocí odrazu zvuku zkoumá vzdálenost od předmětu. Snímač vrací hodnoty v rozsahu 0 - 255 cm. Problém je, že hodnota 255 může znamenat i vzdálenosti, které přesahují tuto hodnotu. Další nevýhodou tohoto snímače je, že některé povrchy nebo materiály rozptylují či pohlcují signál, a tedy v některých případech je nepoužitelný.

3.4.3 Další senzory

Dalšími senzory, které jsou v základním balení jsou zvukový a světelný senzor, který reaguje na zvuky respektive sleduje změny odstínu v odrazech svého světelného zdroje. Další možností, jak rozšířit schopnosti robota, je dokoupit další senzory z nabídky. Kromě společnosti LEGO dodávají senzory pro tyto roboty ještě firmy HiTechnic, MindSensors, Vernier a další. Tyto firmy nabízejí celou řadu senzorů, mezi nejzajímavější patří kompas, akcelerometr, infračervený snímač ad. Existují i některé speciální snímače, jako teploměr nebo senzor pro měření pH.

3.5 NXT-G

Společně s robotem dostáváme v krabici i možnost nainstalovat si programovací nástroj LEGO Mindstorms NXT-G, který umožňuje jednoduše vytvářet aplikace pro roboty. Programy vytvořené pomocí tohoto nástroje jsou po zkompilování nahrány přímo do kostky robota, jejich výhoda vychází z toho, že programy jsou malé v řádu několika kB a řídí robota přímo na nejnižší úrovni. Na rozdíl od MRDS, které řídí robota na bázi neustálé komunikace mezi kostkou a řídícím programem běžícím na jiném stroji.

Programování probíhá pomocí sestavování bloků, které představují jednotlivé části robota, logické prvky, smyčky, Bluetooth komunikaci a další bloky které můžeme použít v programech. Jak vypadá blok je možné vidět na obrázku 16. V základní instalaci se nachází několik desítek bloků. Na internetu je možné stáhnout další desítky až stovky bloků, které nám mohou usnadnit vytváření programů. Je možné taktéž stáhnout SDK pro vytváření vlastních bloků, vytvořených pomocí základních instrukcí robota. Další možností vytvoření vlastního bloku je připravit si jej pomocí nástroje LEGO Mindstorms NXT-G, avšak nevýhodou je, že tyto bloky bývají neúměrně veliké oproti přímo naprogramovaným blokům. A také není vždy možné vytvořit blok potřebných vlastností, např. převod z textu na číslo. Takto vytvořené bloky se spíše hodí pro nahrazování opakujících se částí, jelikož jsme schopni pomoci nich ušetřit nějaké to místo v paměti.

Ve spodní části bloku je výsuvná část, která nám umožňuje komunikovat s ostatními bloky předáváním zpráv. Jedině tak můžou bloky komunikovat, jelikož pokud pouze sestavíme bloky do řady, dochází pouze k jejich postupnému spouštění. Bloky se propojují pomocí linek, které představují posílaní zpráv. Každá zpráva je nějakého typu, pokud propojíme nekompatibilní typy, jsme na to upozorněni při kompilaci. Informace k jednotlivým základním blokům najdeme v nápovědě, s vysvětlením, k čemu je dobrý jaký vstup a výstup, jakého jsou typu a hlavně jak vstupy ovlivňují chování bloku. Hlavní částí, jak ovlivnit chování bloku získáme po kliknutí na samotný blok v levé spodní části obrazovky, kde se zobrazí vlastnosti komponenty.

Velice dobrou možností jak tento nástroj využít je naprogramovat pomocí něj některé části řízení, které je jinak složité vytvořit v MRDS nebo nesprávně fungují. Komunikace probíhá pomocí posílaní zpráv nejčastěji skrz Bluetooth. Jelikož prodlevy mezi spouštěním jednotlivých programů v kostce jsou relativně velké 1 - 3 vteřiny, proto je dobré vytvořit pouze jeden řídící program, který přijímá zprávy a podle obsahu na ně reaguje.

Obrázek 7: Blok v programu LEGO Mindstorms NXT-G

Obrázek 8: Vlastnosti bloku

Problémem však je, že jediný typ zprávy, který je možné z MRDS do NXT-G a zpět zaslat, je `String`. Ostatní typy nejsou kompatibilní.

Dříve než bude možné prográmky napsané v NXT-G využít v aplikacích prostředí MRDS, je nutné učinit několik kroků. Prvním krokem je vytvořit manifest, ve kterém je nadefinována služba pro komunikaci s kostkou NXT a taky služba s názvem IO. Ta se stará o komunikaci mezi kostkou a řídící aplikací. Ještě před prvním využitím aplikace uložené v kostce je nutno tuto aplikaci spustit, jak je ukázáno ve výpisu 23.

```
io.StartLegoProgramRequest request = new io.StartLegoProgramRequest();
request.Program = rp.Body.Filename;

Activate(
            Arbiter.Choice(_ioPort.StartProgram(request),
            delegate(DefaultSubmitResponseType response){
            LogInfo("Start program successful.");
    },
            delegate(Fault fault)
    {
            LogError("Start program error: " + fault.Reason[0].Value);
    }
));
```

Výpis 23: Start programu

Ve chvíli kdy nám program běží, můžeme začít komunikovat s kostkou pomocí zasílání a přijímáním zpráv, jak je ukázáno ve výpisu 24.

```
// Prijem zpravy z mailboxu 1
io.ReceiveBluetoothMessageRequest request = new io.ReceiveBluetoothMessageRequest();
request.Mailbox = 1;
Activate(
      Arbiter.Choice(_ioPort.ReceiveBluetoothMessage(request),
      ReceiveMessage,
      delegate(Fault fault) { Console.WriteLine("Error receveiving " + fault.Reason[0].Value); }
));

//Odeslani zpravy do mailboxu 2
io.BluetoothMessage message = new io.BluetoothMessage();
message.Message = sar.Body.State.ToString();
message.Mailbox = 2;

_ioPort.SendBluetoothMessage(message);
```

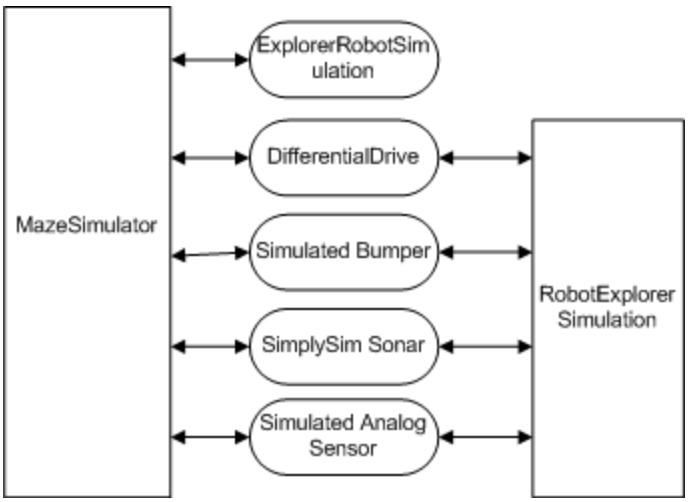
Výpis 24: Příjem a posílaní zpráv

4 Konfigurace robotů

V následujících třech kapitolách bych se chtěl zabývat praktickými poznatky, které jsem nasbíral během vytváření jednotlivých úloh této práce. Jednalo se o úkoly různé obtížnosti a nebylo vždy jednoduché najít řešení. Než se pokusím vysvětlit řešení této práce, chtěl bych se zabývat popisem simulovaného a reálného robota.

4.1 Konfigurace simulovaného robota

Na základě rozšiřitelnosti prostředí VSE byl pro potřeby této práce naprogramován nový model robota. Jako vzor pro tvorbu tohoto robota posloužil reálný robot ze stavebnice LEGO, který je popsán později. Jelikož žádný z předpřipravených modelů nevyhovoval, bylo nutné vytvořit celý model od počátku. Celou tvorbu se budu snažit popsat v následujících kapitolách. Obrázek 9 ukazuje, jak spolu komunikují jednotlivé služby tvořící simulovaného robota.



Obrázek 9: Komunikace služeb simulovaného robota

4.1.1 Simulační služba

Na samotném počátku potřebujeme službu, která se nám postará o vytvoření simulačního prostředí, aby bylo možné samotného robota někam umístit. Jedná se o službu, která zajišťuje komunikaci se službou `SimulationEngine`, taková služba po té může přidávat, nahrazovat nebo mazat entity v simulačním prostředí prostřednictvím portu `SimulationEnginePort`. O vytvoření samotné scény používané při implementaci této práce si povíme o něco později.

Jen pro pochopení jak takovou službu vytvořit, uvedeme malý příklad, na kterém by to mělo být jasnější. Ve své podstatě potřebujeme pro vytvoření základní simulační scény pár

entit jako obloha, podložka, kameru a osvětlení. Samozřejmostí je vytvoření partnerství se službou `SimulationEngine`, abychom mohli vůbec nějaké entity přidávat.

```
// Vytvoreni partnerstvi se SimulationEngine
[Partner("Engine", Contract = engineproxy.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
private SimulationEnginePort _engine = new SimulationEnginePort();

// Metoda Start s pridanim zakladnich entit do sceny
protected override void Start()
{
    SetupSimulationEnvironment();
    AddSky();
    AddLight();
    AddGround();
    base.Start();
}

// Pridani umisteni kamery
private void SetupSimulationEnvironment()
{
    CameraView view = new CameraView();
     view.EyePosition = new Vector3(-1.65f, 0.5f, -0.29f);
     view.LookAtPoint = new Vector3(0, 0, 0);
     SimulationEngine.GlobalInstancePort.Update(view);
}

// Pridani oblohy s texturou skydome.dds a sky_diff.dds
private void AddSky()
{
     SkyDomeEntity sky = new SkyDomeEntity("skydome.dds", "sky_diff.dds");
     SimulationEngine.GlobalInstancePort.Insert(sky);
}

// Pridani osvetleni sceny
private void AddLight() {
     LightSourceEntity sun = new LightSourceEntity();
     sun.State.Name = "Sun";
     sun.Type = LightSourceEntityType.Directional;
     sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
     sun.Direction = new Vector3(0.5f, -0.75f, 0.5f);
     SimulationEngine.GlobalInstancePort.Insert(sun);
}

// Pridani podlahy
private void AddGround()
{
     HeightFieldEntity ground = new HeightFieldEntity("Ground", "Gravel.dds",
     new MaterialProperties("ground", 0.2f, 0.5f, 0.5f)
             );
     SimulationEngine.GlobalInstancePort.Insert(ground);
}
```

Výpis 25: Vytvoření simulační scény

Takto vytvořené simulační prostředí můžeme začít využívat ve službách, které využívají ke svému běhu simulaci. K takovým službám například bude patřit služba vytvářející simulovaného robota.

Ještě bych rád uvedl jeden příklad na vytvoření fyzických entit, které nám budou vytvářet stěny, překážky, části a robota atd. Jedná se o základní stavební prvek všech objektů uvnitř scény, a je proto nutné znát, jak se takovýto prvek vytvoří.

```
BoxShape PlatformBase = new BoxShape(
        new BoxShapeProperties(
        "PlatformBase", // Unikatni nazev entity, unikatnost je dulezita
        1.0f, // hmotnost entity v kilogramech
        new Pose(
            new Vector3(0, 0.06f, -0.09f)), // Umisteni entity, prazdny konstruktor znamena
                relativni umoisteni
        new Vector3(0.13f, 0.09f, 0.005f))); // Rozmery entity
```

Výpis 26: Základní prvek

4.1.2 Podvozek a pohon

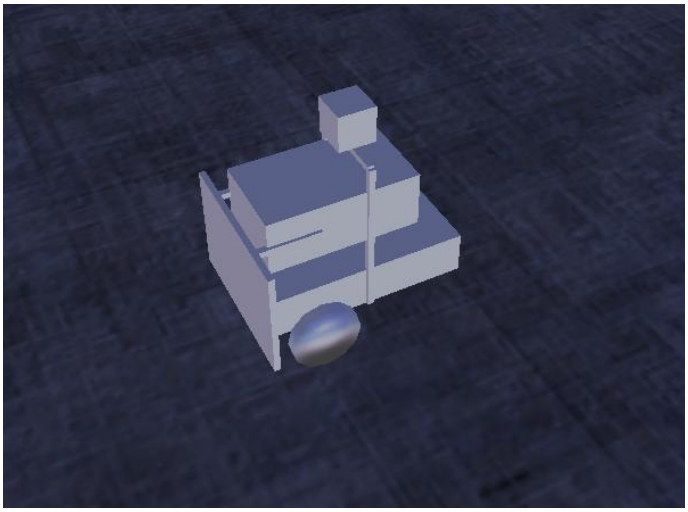
Celé vytváření nového modelu robota začneme od nejdůležitější části, a tou je služba vytvářející podvozek s jednotkou obstarávající pohon. Stejně, jako u reálného modelu, je tato část nejtěžší a tvoří ji část zvaná šasi, dvě přední kola, která se starají o pohyb a otáčení robota a zadní malé otočné kolečko zvané `caster`, které pomáhá při otáčení robota a v simulátoru se modeluje pomocí koule.

Jako nejlepší a nejjednodušší prostředek pro vytvoření podvozku a kostry robota se jeví třída `DifferentialDriveEntity`, která přesně odpovídá našim požadavkům na podvozek. Stačí na specifikovat rozměry jednotlivých součástí, které tvoří požadovaný celek a samozřejmě vytvořit jednotlivé objekty, které se mají zobrazit. Tím, že třída je potomek zmíněné služby, můžeme využít ve svých programech operaci `DriveDistance`, sloužící k pohybu na určenou vzdálenost, a `RotateDegrees`, která zajišťuje otočení robota o požadované stupně. Důležité je poznamenat, že tato třída implementuje kontrakt služby `simulateddifferentialdrive`, která říká , že se jedná o obecnou jednotku pro pohyb a můžeme využít stejnou třídu jako k obsluze reálného robota.

V této práci je navíc tato služba využita k vytvoření celé konstrukce robota a externě se k ní přidávají dva senzory potřebné pro implementaci úloh souvisejících se zadáním. Ukázky pro tuto kapitolu jsou docela rozsáhlé a proto jsou k nalezení v příloze. Pro představu se můžeme na obrázku 10, podívat jak takový nově vytvořený robot může vypadat. Tento model je pouze základní. Je možné jej pokrýt texturami jednotlivých částí, popř. přidat další vlastnosti některých částí.

4.1.3 Simulovaný dotykový senzor

Jeden ze senzorů, které mohou být použity v simulačním prostředí je dotykový senzor. Pro tyto účely je možné využít instanci třídy `BumperArrayEntity`, která je součástí VSE. U simulovaného bumperu je pár věcí, na které je nutné si dát pozor při jeho vytváření.



Obrázek 10: Model simulovaného robota

První problém je, že i když zadáváme konstruktoru simulovaného bumperu rozměry, nevytvoří fyzický objekt představující dotykový senzor. Proto je nutné objekt, který fyzicky představuje bumper vytvořit jinde, v této práci jej vytváří služba starající se o vytváření podvozku. Třída `BumperArrayEntity` vytvoří pouze průhledný objekt, který se chová jako dotykové čidlo a při kontaktu s překážkou či stěnou oznamuje tuto událost. Služba vytvářející tento simulovaný senzor komunikuje se službou, která kontroluje, jestli nedošlo v simulovaném prostředí ke kontaktu s jiným objektem a pokud k tomu dojde, vyvolá oznámení této události. Dále je dobré poznamenat, že simulovaný bumper implementuje kontrakt obecného bumperu, proto může používat stejnou službu pro obsluhu dotykového čidla simulovaného a reálného.

Druhý problem je rozdíl v množství oznámení o kontaktu, reálný bumper vydá pouze jedno oznámení při kontaktu a druhé při uvolnění naproti tomu simulovaný bumper vydává oznámení v krátkých časových intervalech pořád, dokud nedojde k uvolnění kontaktu. Proto je nutné tento rozdíl brát v úvahu a při návrhu služeb tuto skutečnost neopomenout.

Výpis 27 vytváří průhlednou jednotku, která kontroluje, jestli došlo k dotyku nebo ne. Dále je ve výpisu ukázáno, jak vytvořit službu, která se stará o předávání oznámení o změnách stavu dotykového senzoru. Ve výpisu není zahrnutu vytvoření fyzického objektu, znázorňujícího simulovaný dotykový senzor, ale hodnoty použité pro vytvoření průhledné entity je možné použít k jeho vytvoření. Tento objekt je v implementaci této práce vytvářen ve službě starající se o vytváření podvozku a v podstatě celého fyzického modelu robota.

```
BoxShape frontBumper = new BoxShape(
        new BoxShapeProperties(
            "front", 0.001f, //mass
            new Pose(new Vector3(0, 0.06f, -0.09f)), //position
            new Vector3(0.13f, 0.09f, 0.005f)));
frontBumper.State.EnableContactNotifications = true;
BumperArrayEntity bumperArray = new BumperArrayEntity(frontBumper);
bumperArray.State.Name = "LegoNXTBUmpers";

bumperArray.ServiceContract = bumperProxy.Contract.Identifier;

bumperProxy.Contract.CreateService(
        ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
        "http :// localhost/" + bumperArray.State.Name));
```

Výpis 27: Vytvoření simulovaného bumperu

4.1.4 Simulovaný ultrazvukový senzor

Na rozdíl od simulovaného dotykového senzoru, který je součástí VSE, nebyla situace s tímto typem simulovaného senzoru tak jednoduchá. Po dlouhém zkoumání platformy MRDS jsem objevil, že aktuální verze 2008 R2 obsahuje simulovanou entitu pro ultrazvukový senzor. Bohužel po dlouhém boji, kdy se mi podařilo simulovanou entitu správně spustit, jsem zjistil že instance třídy `SonarEntity` není tak úplně simulovaný ekvivalent reálného ultrazvukového čidla, ale dědí ze třídy `LaserEntity`, která se stará o činnost simulovaného laserového senzoru. Hlavním problémem takovéhoto dědění byly odlišné vlastnosti čidel, jelikož laserový senzor má rozsah 180 stupňů místo několika stupňů, kterých využívá ultrazvukový senzor. Navíc tento senzor je velice náročný na systémové prostředky, proto jsem se rozhodl tento standardní simulovaný senzor nevyužít a zkusit hledat alternativní řešení.

Po dlouhém hledání na internetu a po úvahách, jak napsat takový senzor jsem objevil knihovny francouzské společnosti SimplySim, která tvrdila ve svých propagačních materiálech, že jejich senzor je přesnou implementací pro simulovaný LEGO NXT ultrazvukový senzor. Proto jsem si stáhl ze stránek společnosti knihovny pro verzi MRDS 2008 R2, a začal zkoumat, jak jejich senzor přidat ke svému robotu. Po nalezení nápovědy, schované na jejich stránce, jsem vytvořil instanci senzoru. Proto jsem do svých služeb přidal partnerství se službou pro ultrazvukový senzor stavebnice LEGO. Po spuštění služeb to vypadalo, že vše běží v pořádku avšak nepřicházely žádná oznámení o vzdálenostech naměřených senzorem. Bylo to způsobeno špatně navázaným partnerstvím. Informaci, jak propojit simulovaný ultrazvukový senzor společnosti SimplySim jsem získal přímo od tvůrců, kteří velice rychle a ochotně odpovídali na mé dotazy zasílané prostřednictvím elektronické pošty.

Celý problém spočíval v tom, že simulovaný senzor předává hodnoty prostřednictvím analogového senzoru, a proto i v mém programu jsem musel nahradit ultrazvukový

senzor za analogový a získávat data takto. Jedinou nevýhodou analogového senzoru je, že veškeré informace oznamuje pomocí operace typu `Replace`, která je generována v krátkých intervalech, což způsobuje, že i když se robot nepohybuje resp. nedochází ke změně hodnoty, neustále se mění stav senzoru. Toto celé jsem vyřešil obalením příchozích zpráv do vlastní metody typu `Update`, a tím pádem v programu dostávám pouze informace o změnách vzdáleností.

Ještě bych rád upozornil, že výše zmíněný simulovaný senzor využívá standardně dvě služby, jedna představuje výpočet ultrazvukového senzoru a druhá je pro analogový senzor, pomocí které senzor předává informace dále. Toto vše je dobře vidět ve výpisu 28. Ještě poznámka, při vytváření simulovaného ultrazvukového senzoru je vytvořen i fyzický objekt, reprezentující ho ve VSE.

```
// Metoda pro vytvoreni Ultrazvukove Entity
private ultrasonic.UltrasonicEntity CreateSonarSS() {
        ultrasonic.UltrasonicEntity sonarSS;
        sonarSS = new ultrasonic.UltrasonicEntity();
        sonarSS.Box = new BoxShape( // Vytvoreni objektu simulovaneho senzoru
            new BoxShapeProperties(
                    0.1f,
                    new Pose(new Vector3(
                        0f,
                        0.165f,
                        0f)),
                    new Vector3(0.03f, 0.03f, 0.03f)));
      sonarSS.MeasureCone = 10f;  // Rozsah senzoru ve stupnich
      sonarSS.MaxDistance = 2f; // Maximalni merena vzdalenost v metrech
      sonarSS.Scale = 100f; // Meritko
      sonarSS.State.Name = "LegoNxtSonar"; // Unikatni nazev sluzby
      return sonarSS;
}

// Metoda spoustejici sluzbu simulovaneho ultrazvukoveho senzoru
private void RegisterSonarSS() {
      DsspResponsePort<CreateResponse> responsePort =
                ultrasonic.Proxy.Contract.CreateService(
                    ConstructorPort,
                    Microsoft.Robotics.Simulation.Partners.CreateEntityPartner("http://localhost/"
                        +
                    sonar.State.Name));  // Spusti sluzbu
       // Po spusteni a pridani sluzby do simulatoru spusti metodu SubscribeToSonarSS
       Activate(
                Arbiter.Choice(responsePort, SubscribeToSonarSS, LogError));
}

// Metoda spoustejici analogovy senzor a vytvoreni propojeni mezi ultrazvukovym senzorem a
// analogovym senzorem
private void SubscribeToSonarSS(CreateResponse response)
{
            analog.AnalogSensorOperations analogPort =
                ServiceForwarder<analog.AnalogSensorOperations>(response.Service +
                "/analogsensor");
```

```
            analog.AnalogSensorOperations analogNotify =
                new analog.AnalogSensorOperations();
            Arbiter.Receive<analog.Replace>(true, analogNotify,
                delegate(analog.Replace replace) {
            });
            analogPort.Subscribe(analogNotify, typeof(analog.Replace));
}
```

Výpis 28: Vytvoření simulovaného ultrazvukového senzoru

4.1.5 Startovací služba

V tuto chvíli bychom měli mít připraveny všechny potřebné části k vytvoření modelu robota pro simulátor VSE. Jen pro shrnutí bych rád připomněl, že robot se skládá ze základní služby, která vytváří podvozek a potažmo v mém případě celou kostru robota. Dále máme vytvořen dotykový a ultrazvukový senzor. A nakonec je tady ještě služba, která to dá celé dohromady a připraví model robota k používání jako celku.

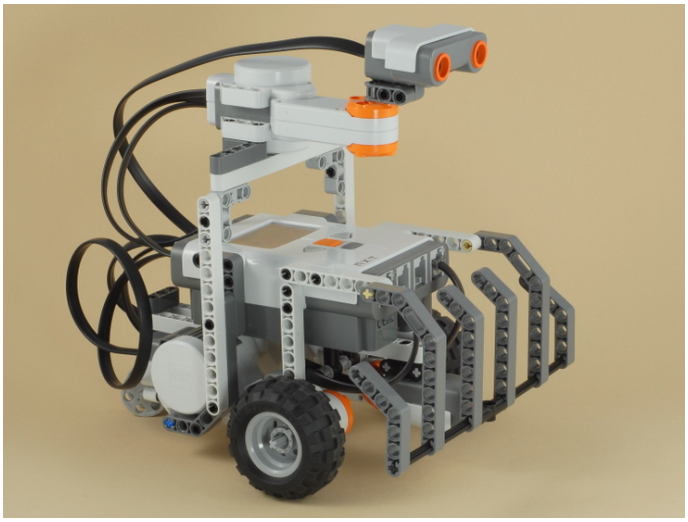
```
protected override void Start() {
            bumper = AddBumper(); // Vytvoreni dotykoveho senzoru
            sonar = CreateSonarSS(); // Vytvoreni ultrazvukoveho senzoru
             ExplorerRobotWithDrive robot = new ExplorerRobotWithDrive("ExplorerRobotWithDrive
                 ", new Vector3(0, 0, 0)); // VYtvoreni modelu robota s umistenim v prostredi na
                 pozici 0,0,0
             robot.InsertEntity(bumper); // Pridani bupmeru k robotovi
             robot.InsertEntity(sonar); // Pridani sonaru k robotovi
             SimulationEngine.GlobalInstancePort.Insert(robot); // Pridani robota do simulatoru
             RegisterSonarSS(); // Nastartovani ultrazvukoveho senzoru
             base.Start();
}
```

Výpis 29: Vytvoření celku robota

4.2 Konfigurace reálného robota

Pro implementaci úloh v této diplomové práci byla pro stavbu robota využita stavebnice LEGO NXT. Podvozek robota je sestaven ze dvou předních kol, které se starají společně s motory o pohyb a otáčení robota. V zadní části je ještě malé pomocné kolečko, které je využíváno především k otáčení.

Ke zkoumání okolí využívá robot dvou senzorů, které byly vybrány, a to ultrazvukový a dotykový. Tyto dva snímače se ukázaly jako nejlepší pro řešení dané úlohy, jelikož pomocí nich můžeme získat velice snadno obraz místnosti, ve které se robot pohybuje. Ještě bych dodal, že ultrazvukový senzor je umístěn na motoru, čímž je umožněno jeho otáčení, bez nutnosti otáčet celého robota. Samozřejmou součástí je řídící kostka, která dělá prostředníka mezi prvky robota a počítačem, který vydává řídící příkazy a přijímá údaje se senzorů. Výsledného robota můžete vidět na obrázku 11, návod na jeho sestavení



Obrázek 11: Model reálného robota. Zdroj: www.nxtprograms.com

se nachází na přiloženém DVD. Model byl převzat ze stránek `www.nxtprograms.com`, na které je možné nalézt mnoho návodů na sestavení robota.

5 Návrh

Během prvních úvah o návrhu aplikace splňující zadaní jsem myslel, že budu mít pouze jeden společný program pro reálného a simulovaného robota. Jenže to se během implementace ukázalo jako nesplnitelné, proto jsem musel začít návrh od znova. Návrh rozpadl se na dvě části, jedna obsluhuje robota v prostředí VSE a druhá robota sestaveného z Lega.

Obecně lze říci, že se architektura skládá ze služeb logiky robota, ovládajících robota, mapování prostředí a dalších. Oba modely mají společné části, které budou popsány společně.

5.1 Strategie průzkumu obvodu místnosti

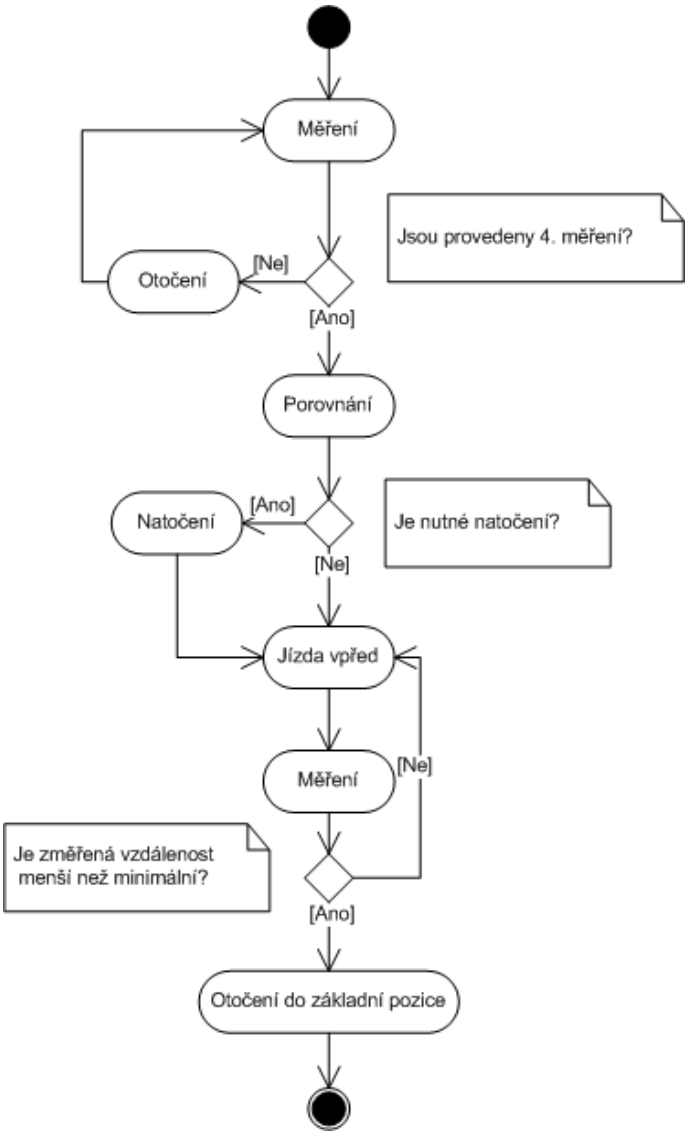
Strategie není samostatnou službou, ale je zahrnuta v centrálním řídícím prvku, který je popsán v další podkapitole. V této části bych se chtěl pokusit o popis, jak vůbec se robot v této práci snaží prozkoumat prostředí, aby se v něm následně mohl pohybovat. Díky zvoleným senzorům se robot dokáže pohybovat po místnosti bez kolize již během první jízdy.

Po nastartování služby se robot pokouší zjistit, kde se nalézá nejbližší stěna, od které by mohl začít průzkum. Nejprve změří vzdálenost ke stěně, na kterou je natočen po spuštění, potom se na svém místě pootočí třikrát o 90 stupňů a při každém pootočení provede měření. Následně vyhodnotí, která vzdálenost je nejkratší, pokud si jsou všechny vzdálenosti rovny, robot jede dopředu V momentě, kdy nalezne nejbližší stěnu, začíná samotné zkoumání prostředí. Tento "startovací" algoritmus můžeme vidět na obrázku 12 v podobě aktivitního diagramu.

Průzkum tedy startuje, když je robot natočen o 90 stupňů odprava od stěny, to znamená že stěnu má po levém boku. Takovou polohu nazývám normální nebo taky základní polohu. Toto je základní pozice, další povolenou pozicí je poloha při natočení o 45 stupňů od své normální pozice. Ještě bych rád poznamenal, že robot by neměl být ke stěně blíže než 20 cm a více než 50 cm. První vzdálenost je velikost základního kroku, o který se robot posouvá dopředu, resp. dozadu. Ale vraťme se k popisu algoritmu, který robot využívá ke své činnosti. Když v popisu používám minusové hodnoty pro vyjádření stupňů mám na mysli stupně proti hodinových ručiček a opačně.

Popis začneme na jednoduché čtvercové místnosti. V tomto případě by se měl robot nacházet v pozici, kdy je natočen o 90 stupňů od stěny, pak tedy provede celkem tři měření. Nejprve změří vzdálenost před sebou, pak vzdálenost o -90 stupňů od své polohy, čili vzdálenost směrem ke stěně a nakonec vzdálenost o -45 stupňů. Následně porovná získané hodnoty a vybere nejmenší z nich, podle vybrané nejmenší vzdálenosti proveden natočení robota pokud je to nutné a posune se o 20 cm vpřed.

Další možností, které jsou do řešených úloh zahrnuty, je možnost, že se stěny lomí o 45 stupňů vůči své původní poloze. Pokud nastane tato možnost, robot se natočí o 45 stupňů po směru hodinových ručiček a pokračuje ve své činnosti, pouze s tím rozdílem, že měření provádí pouze o -45 a 45 stupňů od své polohy, snaží se tedy dostat do své normální polohy.



Obrázek 12: Aktivitní diagram pro startovací algoritmus

Dalším možným krokem je otáčení doprava, pokud narazí robot na roh místnosti. Jelikož ji objíždí vždy po směru hodinových ručiček potřebuje pouze určit, jestli se má natočit o 45 neb 90 stupňů doprava. Poslední možností je případ kdy robot potřebuje pro pokračování couvání. Nejprve se vždy snaží podívat doprava, jestli by se nemohl pootočit tam, pokud to není možné musí robot couvnout o standardní vzdálenost zpátky.

Toto je tedy základní popis funkčnosti robota, tak jak jsem ho navrhl pro tuto práci. V implementaci dochází k rozdílnostem mezi jednotlivými prostředími, jelikož funkčnost simulovaného a reálného robota se liší. Pomocí výše popsaného algoritmu je schopen robot prozkoumat obvod místnosti. K průzkumu vnitřních prostorů místnosti používá jiný algoritmus, který je více spjatý se záznamem místnosti do mapy. Ještě je dobré poznamenat, že robot objíždí místnost dokud nenarazí na počáteční bod, rozeznat počátek mu pomáhá mapovací služba, která je popsána jinde.

5.2 Centrální řídící prvek

Jako centrální řídící prvek označujeme službu, která je primárně spouštěna a zároveň se stará o spouštění všech částí potřebných k provozu aplikace. Tato služba obsahuje v případě této práce i logiku řízení robota, kterou by bylo možné přesunout do samostatné služby a tuto základní službu degradovat pouze na obyčejnou startovací službu.

Po dlouhé úvaze, jak nejlépe vytvořit hlavní službu pro řízení robota v asynchronně komunikujícím prostředí, jsem zvolil jedno ze základních informatických řešení, a to konečný stavový automat. Dalo by se říct, že automaty existují v podstatě tři. První se stará o připraveni robota na průzkum místnosti, druhý se stará o průzkum obvodu místnosti. A třetí se stará průzkum obsahu místnosti.

5.3 Mapovací služba

Služba, která se stará o mapování pohybu robota je další služba, která je společná pro obě prostředí a asi jediná, která je použita naprosto stejně v obou prostředích. Její základní funkce je jednoduchá. Přijímá informace z řídící služby a vytváří si v paměti obraz prostoru, který robot prozkoumává. Další její funkcí je, že hlídá robota během průzkumu obvodu místnosti, jestli nedorazil na počátek. Ve chvíli, kdy je průzkum obvodu místnosti hotov, přepíná se do módu, kdy pomáhá dohledat řídící službě prázdná místa uvnitř místnosti a snaží se je zaplnit. Jak tedy taková služba funguje se budu snažit osvětlit v následujících odstavcích, jelikož tato služba hraje i velkou roli v průzkumu obsahu místnosti.

Základem je opět stavový automat, který se stará o určování natočení robota oproti počátku. Jako počátek se bere základní stav robota, tedy kdy je natočen o 90 stupňů doprava vůči stěně. Další důležitý objekt, který si tato služba musí udržovat v paměti je matice, ve které jsou uloženy hodnoty o stavu průzkumu místnosti. V tabulce 5 je možné vidět jakých hodnot mohou jednotlivá políčka nabývat.

Mapovací služba není nutná během "startovací" fáze proto je postačující, když je aktivována až po dokončení této fáze. Během fáze průzkumu obvodu přijímá informace z řídící služby, která ji posílá výsledky z měření a směr, kterým by se robot měl dále

Hodnota	Popis
UNKNOWN	Označení pro neprozkoumané políčko, počáteční hodnota pro všechna políčka
ENVIRONMENT	Hodnota označující volné políčko, na kterém se neobjevuje žádná překážka a robot ho může využívat
WALL	Označení pro zeď místnosti, použito při průzkumu obvodu místnosti
INSIDE	Hodnota označující překážku uvnitř místnosti

Tabulka 5: Hodnoty políček matice

pohybovat. Hodnoty popisující vzdálenost uloží do matice s popisem typu políčka a podle směru se snaží určit kam se robot bude pohybovat a tedy které další políčko se bude zapisovat. Další podstatnou funkcí, kterou tato služba vykonává je zastavení průzkumu obvodu, kdy se jí řídící služba pokaždé ptá jestli už robot neobjel celou místnost a neměl by skončit.

Před další fázi, bychom si měli nadefinovat několik pojmů, které již byly nebo budou využity v tomto textu. Jsou to pojmy políčko a matice. Pro záznam údajů potřebných k popisu místnosti je využito dynamické matice, která se umí zvětšovat na všechny strany podle potřeby. Jednotlivé prvky matice tvoří políčka, která nám poskytují informace o daném místě v místnosti. Každé políčko matice reprezentuje část místnosti o velikosti 20 x 20 cm, jelikož to je velikost, kterou robot urazí v každém kroku.

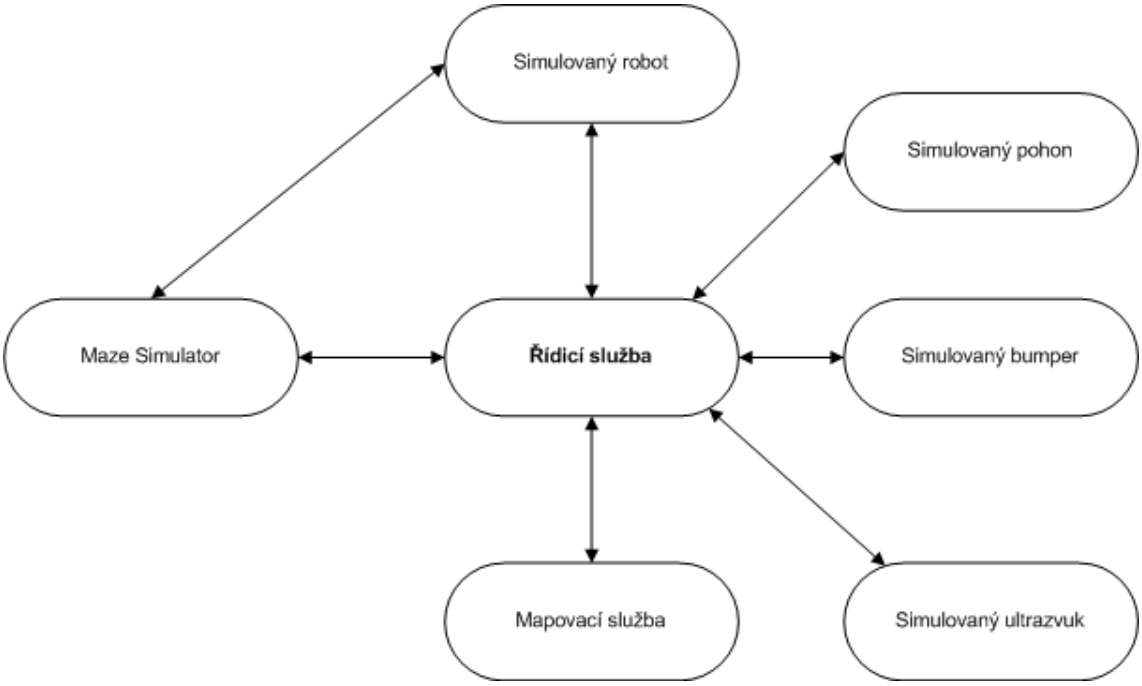
V další fázi, která se snaží zmapovat vnitřní prostory místnosti, hraje tato služba mnohem větší roli. Nejprve se ze získaných dat, která má uložené v matici, snaží získat všechna neprozkoumaná políčka uvnitř místnosti. Z takto získaného seznamu se vždy snaží získat nejbližší neprozkoumané políčko od toho aktuálního a navést robota na toto místo. Takto pokračuje až do doby, než je celý obsah místnosti pokryt. Pro získání nejbližšího políčka je využit jednoduchý výpočet vzdálenosti, který nezohledňuje skutečnou délku nejkratší cesty. Proto se může stát, že dvě nejbližší políčka dělí zeď a robot je tedy navigován špatně. Pokud v tomto případě robot narazí na zeď zachová se stejně, jako v případě, že během zkoumání obsahu narazí na překážku v cestě. Jednoduše najde opět nejbližší volné políčko od toho aktuálního, kde se nachází, a nasměřuje robota k němu.

Poslední funkcí této služby je, že výsledný obraz místnosti v podobě matice umí uložit do souboru. Toto je praktické pro mapování reálných místností, méně už pro simulovaného robota. Získaný obraz místnosti můžeme využít ve službě `MazeSimulator`, která bude popsána v části věnující se implementaci. Tato služba umí z obrázku vytvořit v simulátoru scénu v něm uloženou, pokud jsou dodržena některá pravidla. O nich se zmíním v kapitolce věnované této službě.

5.4 Simulované prostředí

Základní myšlenkou MRDS je rozdělit ovládání do několika samostatných služeb, které spolu asynchronně komunikují. Také návrh pro řízení robota je rozdělen do několika

služeb, které se starají o jednotlivé části řízení robota. Na obrázku 13 je možné vidět celkový koncept aplikace určené pro prostředí VSE.

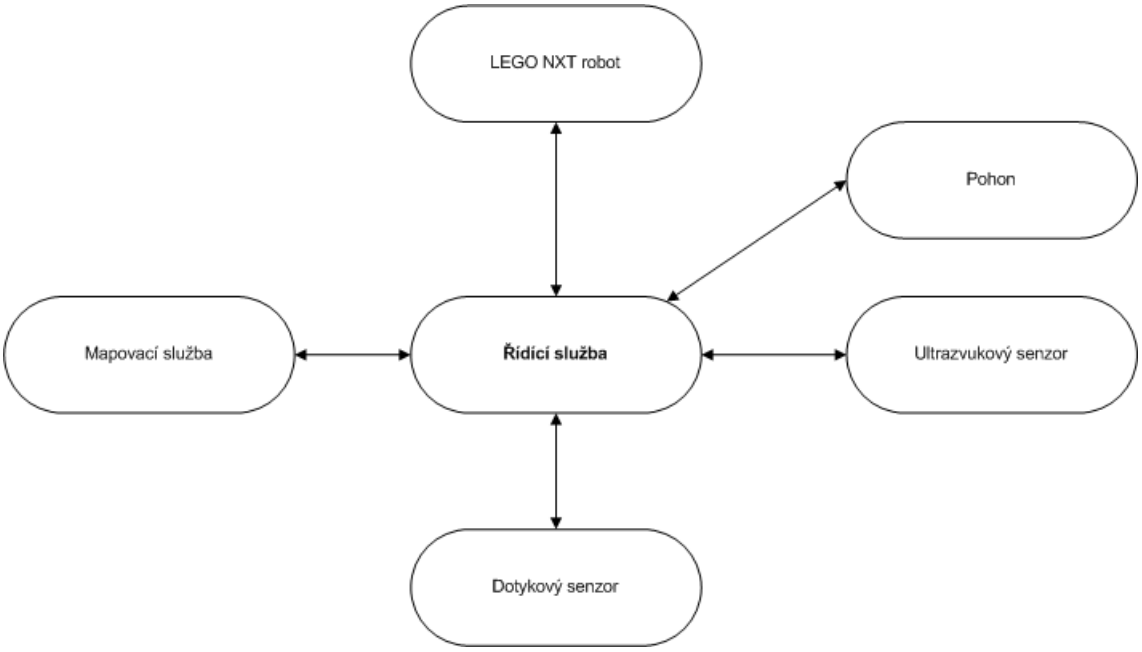


Obrázek 13: Propojení simulovaných služeb

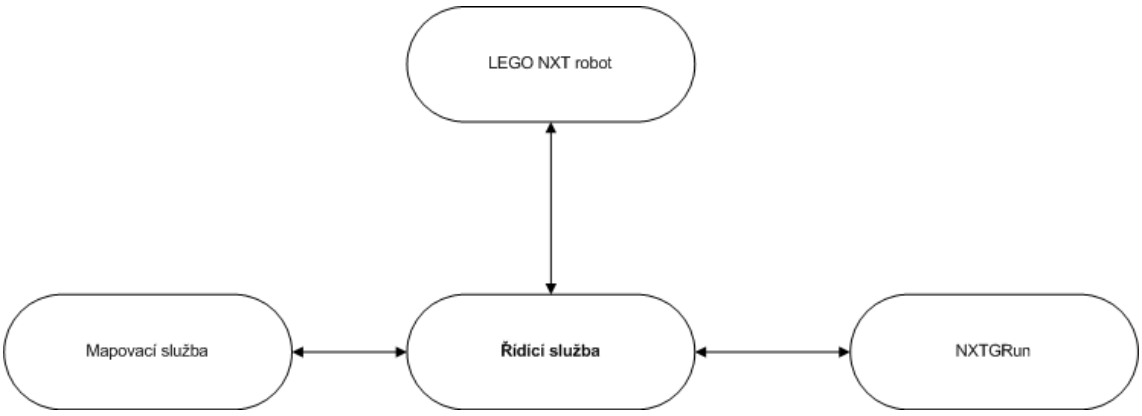
Kompletní aplikace pro VSE je navržena tak, že simulované prostředí vytváří z předdefinovaného obrázku služba `MazeSimulator`. V obrázku zmíněná služba Simulovaný robot představuje implementaci služby, reprezentující model robota. Samozřejmostí jsou služby jako centrální řídící prvek nebo mapovací služba.

5.5 Reálný robot

Stejně jako návrh aplikace pro simulované prostředí, je i program pro reálného robota sestaven z několika služeb, které obsluhují jednotlivé části. První návrh aplikace byl určen zcela pro implementaci v prostředí MRDS, což se však ukázalo jako nefungující řešení, které muselo být nahrazeno alternativou. Problém pravděpodobně pramenil z komunikace mezi počítačem a řídící kostkou pomocí Bluetooth, jak je popsáno v kapitole věnující se LEGO NXT. Proto se finální návrh liší od návrhu původního viz. obrázky 14 a 15.



Obrázek 14: Původní návrh sestavení služeb



Obrázek 15: Finální návrh sestavení služeb

6 Implementace

Poslední část mé práce se věnuje implementaci jednotlivých služeb. Z důvodu rozdílnosti mezi simulovaným a skutečným prostředím uvádím dvě různé implementace, které je možné nalézt na přiloženém DVD. Budu se věnovat pouze popisu implementace respektive funkčnosti služby MazeSimulator, po té se podíváme jak jsou utvořeny řídící struktury pro oba typy robotů a nakonec si představíme implementaci mapovací služby.

6.1 MazeSimulator

MazeSimulator je aplikace respektive služba převzata z přílohy ke knize [1], která se jevila velice výhodná k využití v mé aplikaci. Jedná se o službu vytvářející prostředí simulované scény a to z předdefinovaného obrázku. Já používám ve své práci obrázky typu JPG.

Použití této služby je velice jednoduché v konfiguračním manifestu je nutné pouze nastavit cestu a jméno obrázku jak je vidět ve výpisu 30 a po té je možné službu spustit.

```
<Maze>mistnost.bmp</Maze>
```

Výpis 30: Konfigurace vstupního obrázku

Původně je služba určena pro dva typy robotů. Těmito roboty jsou Pioneer 3DX a LEGO NXT, ovšem pro potřeby této práce jsem potřeboval přidat model svého robota proto jsem upravil patřičnou metodu, která se stará o přidání robota do prostoru. Navíc jsem zavedl svého robota v sestavovacím manifestu této služby a tím mám zaručeno, že dojde ke spuštění i služby starající se o mého robota, včetně všech podpůrných. Pomocí této je robot tedy přidán do simulační scény, kterou nám služba vytvořila. Dále se tato služba stará i komunikaci mezi simulovaným světem a řídícími prvky aplikace prostřednictvím `SimulationEnginePort`.

Aby bylo možné službu MazeSimulator využít je nutné dodržet několik základních pravidel, na které se nyní podíváme. Pokud neuvádíme v tagu `Maze` cestu k souboru tak je nutné mít takovýto soubor zkopírován ve složce `MRDS\store\media\maze_textures`. Tento obrázek může obsahovat pouze 16 barev, které je služba schopna reprezentovat jako překážky. Seznam barev je zobrazen na obrázku `colorpalette16.bmp` ve složce služby. Obrázek může být v několika formátech, nejvhodnější je obyčejný bmp soubor, který můžeme vytvořit i v programu malovaní ve Windows. Další možné formáty jsou GIF a JPG. Avšak JPG formát má nevýhodu v tom, že je ztrátový kompresní formát a může tudíž dojít ke ztrátě informací. Objekty a jejich výška v simulátoru je označena barvou, která je na obrázku. Barva, která se nachází levém horním rohu je použita pro podlahu a proto by neměla být použita pro žádný jiný objekt ve scéně. Další barvy jsou využity, pro vytvoření dalších objektů, je vždy dobré vybrat bravu, která bude představovat stěnu a tou ohraničit celý obrázek, abychom zajistili konzistenci prostředí a taky to, že robot nevyjede mimo vytyčený prostor.

6.2 Knihovna `RobotControl`

Než se dostanu k popisu jednotlivých řídících služeb, chtěl bych se zmínit o knihovně `RobotControl`, která převzala roli pomocné knihovny. Jejím obsahem je několik služeb, které jsem vytvořil tak, aby poskytovali co největší komfort pro použití v ostatních službách. Ve většině případů tyto služby pouze překrývají funkčnost původních služeb komunikujících s jednotlivými senzory nebo motory.

Tato knihovna vznikala ještě v době, kdy jsem si myslel, že bude stačit pouze jedna řídící služba pro obě prostředí, což se později ukázalo jako nereálné nebo jsem nenašel řešení, které by to umožňovalo. Proto obsahuje i služby, které se starají o obsluhu služeb pro LEGO NXT, i když ve výsledné práci nejsou plně využity. V následujících podkapitolách bych rád představil všechny služby, hlavně se budu věnovat těm, které se se podílejí na výsledných aplikacích.

6.2.1 Služba RobotControl.Drive

Klíčová a nejproblematičtější služba kvůli, které došlo k rozdělení na dvě řídící aplikace. Problém není v tom, že by služba byla špatně napsaná, ale v tom, že dochází ke ztrátě potvrzovacích zpráv z reálného robota. Tomuto problému se budu více věnovat v kapitole 6.3.

Tato služba je tedy použita hlavně pro simulovaný model, u kterého funguje bez problému způsobujících zastavení programu robota. U této služby jsem si nadefinoval dvě základní metody, které se starají o provoz robota. Jsou to metody pro pojíždění robota `SetMove` a pro otáčení `SetRotation`. Obě metody jsou ve svém základě podobné, i když vykonávají jinou činnost. Obě operace jsou typu `Submit` a obě pracují v módu `Exclusive`, navíc aby se tyto metody ukončili musí pohyb resp. rotace proběhnout celá v pořádku. Teprve po té je dokončena hlavní úloha operace, je vydané oznámení typu `Update`, že služba Drive dokončila svou činnost.

K této službě si dovolím jednu malou ukázku, která je s malými změnami použitelná pro obě operace. Jedná se o metodu, která hlídá dokončení pohybu resp. rotace robota. U pohybu se sleduje vlastnost `DriveDistanceStage`, u rotace `RotateDegreesStage`. Tato metoda je klíčová pro ukončení dvou výše popsaných operací služby a ve výpisu 31 je metoda, která na dokončení pohybu.

```
private IEnumerator<ITask> DriveDistanceHandler(drive.DriveDistance distance) {
if (distance.Body.DriveDistanceStage == drive.DriveStage.Canceled) {
    completitionPort.Post(distance.Body.DriveDistanceStage);
} else if (distance.Body.DriveDistanceStage == drive.DriveStage.Completed) {
    completitionPort.Post(distance.Body.DriveDistanceStage);
}
yield break;
}
```

Výpis 31: Metoda čekající na dokončení nebo zrušení pohybu

6.2.2 Služba RobotControl.SimulatedSonar

Jak je z názvu patrné jedná se u službu, která se stará o obsluhu simulovaného ultrazvukového čidla. Jak již jsem se zmínil v kapitole věnující se vytvoření simulovaného robota, je tento senzor a oznámení z něj přicházející tvořen analogovým senzorem. Služba analogového senzoru však poskytuje pravidelně vyvolávanou operaci typu `Replace`, i v případě, že nedochází ke změně vzdálenosti od zdi či překážky.

Neustále vyvolávaní operace Replace je nežádoucí a proto jsem vytvořil tuto službu, která přijímá oznámení z analogového senzoru a testuje jestli se změnila vzdálenost od posledního oznámení. Pokud dojde ke změně oznámí tuto událost ostatním službám pomoci operace `Update` jak můžeme vidět ve vypísu 32.

```
private void AnalogReplace(analog.Replace replace) {
    double currentMeasurement = Math.Round(replace.Body.RawMeasurement, MidpointRounding.
        AwayFromZero);
    if (lastMeasurement != currentMeasurement)
    {
        _state.Distance = currentMeasurement;
        _mainPort.Post(new Update(_state));
    }
    lastMeasurement = currentMeasurement;
}
```

Výpis 32: Kontrola změn vzdálenosti simulovaného ultrazvukového čidla

6.2.3 Služba RobotControl.NXTGRun

Toto je poslední služba, které se budeme věnovat při popisu služeb na implementovaných v knihovně `RobotControl`. Služba se věnuje komunikaci mezi řídící službou a kostkou robota LEGO NXT, z čehož vyplývá, že je používaná v aplikaci pro fyzického robota. Tato služba vznikla po všech neúspěšných pokusech zprovoznit komunikaci čistě pomoci MRDS a jazyka C#. Jejím úkolem je zjednodušit komunikaci pro spuštění ovládací služby na robotovi a hlavně zjednodušit zasílaní a příjem zpráv mezi robotem a řídící službou, která rozhoduje o dalších krocích.

Služba obsahuje dvě operace, které se starají o dříve zmíněné operace. První operace jen spouští řídící program a je typu `Submit`. Druhá služba je již o něco zajímavější, je také typu `Submit` avšak přijímá jako vstupní parametr počet zpráv, které by měla operace provedená na robotovi vrátit. Ještě doplním, že tato metoda se stará jak o zasílaní, tak i o přijímaní zpráv, tím že nastartuje příslušné metody. První se stará o čekání na přijetí požadovaného počtu zpráv a druhá o jejich samotné přijetí, kdy zkouší přijmout zprávy z robota ve zvoleném intervalu. V mém případě to zkouší každých 100 ms, tato hodnota se ukázala výhodná z důvodu, že nedochází ke zbytečnému čtení prázdných zpráv mezi příchodem jednotlivých zpráv. Obě tyto metody můžeme vidět ve výpisu 33. Rád bych ještě podotkl, že výpisy ke spuštění programu v kostce a výměny zpráv jsou v dřívější kapitole ve výpisu 23 resp. 24

```
private IEnumerator<ITask> ReceiveMessages() {
    receive = true;
    while (receive)
    {
        Thread.Sleep(_state.Poll);
        io.ReceiveBluetoothMessageRequest request = new io.ReceiveBluetoothMessageRequest();
        request.Mailbox = 1;
        Activate(
            Arbiter.Choice(_ioPort.ReceiveBluetoothMessage(request),
            ReceiveMessage,
            delegate(Fault fault) {Console.WriteLine("Error receveiving " + fault.Reason[0].Value); }
        ));
    }

    yield break;
}

private void ReceiveMessage(io.BluetoothMessage message) {
    int distance = int.Parse(message.Message);
    distancePort.Post(distance);
}

private void DistanceHandler(int[] array) {
    receive = false;
    _state.messages = array;
    _mainPort.Post(new NXTUpdate(_state));
}
```

Výpis 33: Metody obsluhující přijem zpráv z kostky

6.3 Řídící služba reálného robota

Nyní konečně vysvětlil problém, který vyvolal řadu úprav a změnil implementaci pro reálného robota od základu. Problém vyvstal v podstatě jediný, ale zásadní, a to v tom, že se čas od času stávalo, že robot přestal reagovat na příkazy k pohybu. Nebo pohyb vůbec nedokončil, popřípadě když pohyb dokončil, tak o tom neinformoval řídící službu. To způsobovalo zastavení běhu programu, jelikož aplikace vyžaduje znalost přesně ujetých úseků k řízení běhu programu a záznamu v mapovací službě. I když jsem zkoušel mnoho způsobů, jak vyřešit tento problém, tak se mi to nezdařilo, a proto bylo nutné sáhnout k alternativnímu řešení, které znamenalo rozdělit řešení na dvě.

Jako alternativu jsem zvolil kombinaci implementace řízení pomocí MRDS v jazyce C# a vykonávaní obsluhy robota pomocí programovacího jazyka NXT-G. Tyto dvě části fungují na principu výměny zpráv, kdy řídicí služba zasílá zprávy, které obsahují informace o tom, jakou činnost má robot vykonat a pokud je to nutné, čeká na příjem zpráv. Program běžící v kostce funguje jako nekonečná smyčka, která vždy po ukončení své hlavní činnosti čeká na zaslání informace, jestli bude běh programu pokračovat nebo ne. Pokud ano, vrací se na začátek a opět čeká na příjem zprávy, která určí další činnost robota.

Během vykonávání hlavní činnosti jsou postupně z robota odesílány zprávy, nejčastěji s naměřenými hodnotami vzdálenosti robota od překážky. Příjem a zasílání používají oddělené schránky k výměně zpráv, i když je toto pouze bezpečnostní opatření před ztrátou zpráv. V reálné situaci by nemělo nastat, že se schránka přeplní, jelikož jsou odesílány nejvýše tři zprávy z robota do řídící služby a jedna zpráva je přijímána.

Program v řídící kostce je rozdělen na několik větví, které se starají o jednotlivé úkoly. O tom, který se provede je rozhodnuto na základě přijaté hodnoty z počítače. Tato hodnota je číselného charakteru, ale jak jsem zmínil v kapitole věnující se NXT-G, mezi MRDS a mezi kostkou NXT je možné zasílat pouze textové zprávy, proto jsem musel v NXT-G začít přijímat typ řetězec a po té ho převést na číslo. Tuto operaci však standardní NXT-G neumí a proto jsem musel doinstalovat nejprve update pro programovací prostředí a po té doinstalovat nový blok, který jsem na šel na internetu. V tabulce 6 je možné vidět jednotlivé číselné kódy s významy, použité pro komunikaci.

Hodnota	**Popis**
-3	Robot pouze pokračuje v jízdě pod úhlem
-2	Robot se před pokračováním otočí o 45 stupňů doleva
-1	Robot se před pokračováním otočí o 45 stupňů doprava
0	Robot před pokračováním nemusí vykonat žádnou dodatečnou činnost
1	Robot se před pokračováním otočí o 90 stupňů doprava
2	Robot se před pokračováním otočí o 90 stupňů doleva
3	Hodnota použita k oznámení, že robot se vrátil do normální polohy otočením doleva o 45 stupňů
4	Podobné hodnotě 3 s tím rozdílem, že se robot otáčí o 45 stupňů doprava
10	Hodnota způsobí couvání robota o jednu rotaci zpět
10	Hodnota využívána k měření vzdáleností vpravo od robota bez rotace
99	Hodnota, kterou musí kostka přijmout před pokračováním smyčky

Tabulka 6: Kódy pro komunikaci mezi robotem a NXTG

V tabulce 6 se zmiňuji o pokračování, tím myslím činnosti navazující na první úlohy programu v kostce. Rád bych poznamenal, že tyto číselné kódy se používají k určení, jestli je nutné natočit robota a o kolik. Další věcí, kterou číselné hodnoty určují, je co se bude dít po dokončení rotace. Pro hodnoty menší než 10 platí, že po dokončení s nimi souvisejících akcí se posune robot o jednu rotaci předních kol vpřed. Tato hodnota je klíčová i při zjišťování vzdáleností robota od překážky, hodnoty menší než nula používají jiný princip než hodnoty větší než nulou včetně nuly. Pokud robot příjme hodnotu větší než 10, znamená to, že se bude provádět jiná sekvence kroků. Touto sekvencí může být

zjišťování vzdáleností vpravo od robota, použité před otočením robota doprava nebo jí může být případ, kdy se robot dostane do slepé uličky a potřebujeme robota posunout zpět.

Nyní zmíníme jak probíhá určování toho jak se má postupovat dál v řízení. Jak vyplývá z návrhu, je vnitřní struktura služby navržena jako stavový automat, kdy si pamatujeme poslední stav a podle reakce od ostatních služeb přechází služba do jiného stavu. Popis vnitřní funkcionality začneme ve chvíli, kdy řídící služba má vykonat nový pohyb robota. Požadavek na pohyb robota je učiněn změnou stavu služby se vstupním parametrem určujícím, co se má vykonat. Po té robot změní poslední stav robota, vyvolá zaslání zprávy robotu s požadavkem na vykonání činnosti. V tuto chvíli se řídící program uspí a čeká na vyřízení svého požadavku. Ve chvíli kdy obdrží všechny zprávy, které požadoval, může změnit stav, který vyvolá zpracování dat.

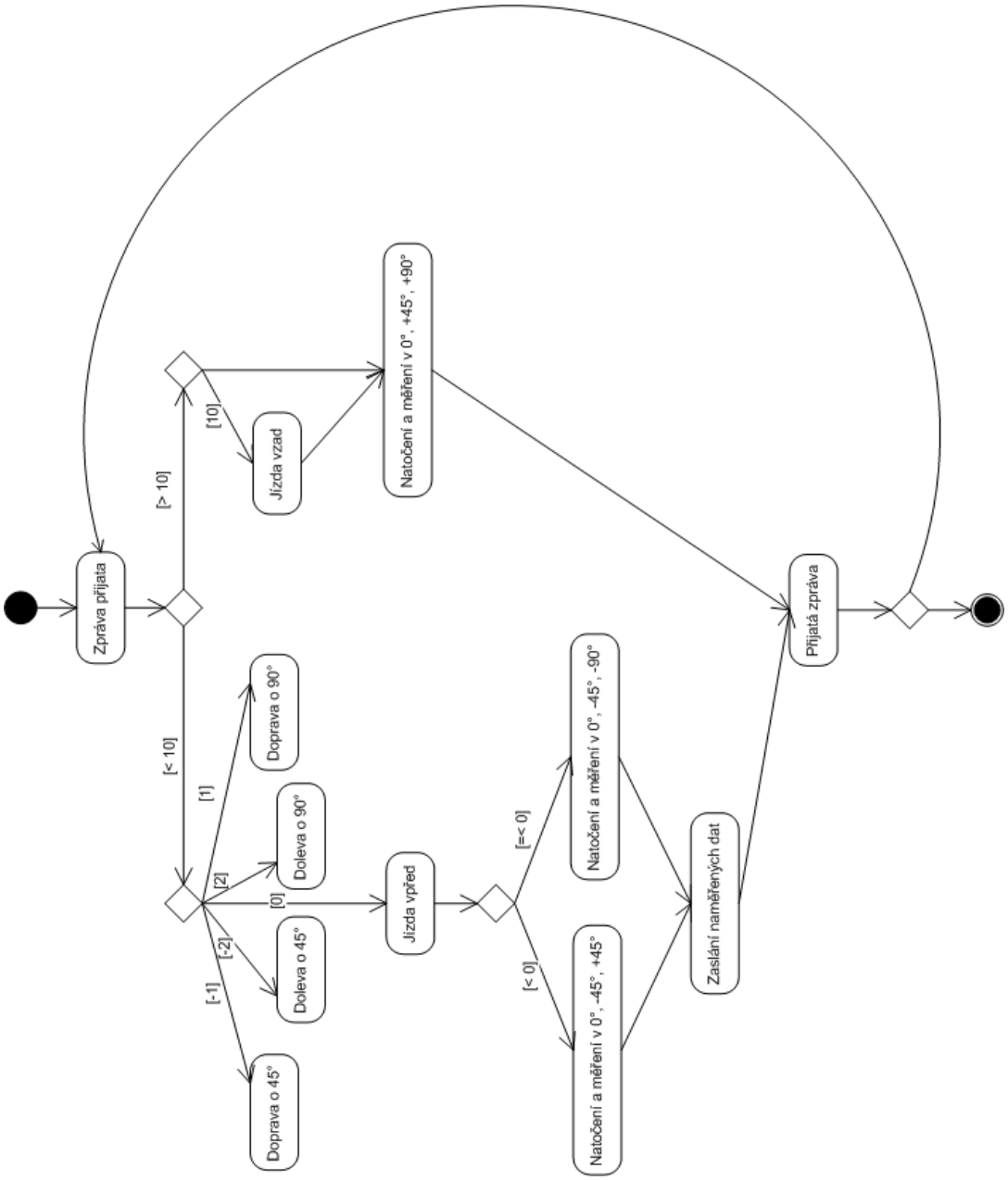
Zpracování přijatých údajů je rozdílné v závislosti na právě skončené činnosti, a dělí se na tři části. Obecně se dá říci, že v tomto místě pouze řídící program rozhodne o dalším kroku robota. Dříve než začne připravovat další krok, pošle aktuálně přijaté hodnoty do mapovací služby, která se postará o jejich zápis. Poté požádá mapovací službu o posun kurzoru pro zápis ve směru dalšího kroku. Předtím než však změní stav pro vykonání dalšího kroku, musí řídící služba se dotázat mapovací služby, zda-li může ještě pokračovat.

Do této chvíle jsem nastínil implementaci složitější části, která se stará o zjištění obvodu místnosti, pokud robot dojede do místa ze kterého začal je první část u konce. Poté je na mapovací službě, jestli rozhodne, zda-li se má nebo nemá pokračovat. Pokud je určeno pomocí jejich výpočtů, že místnost má obsahuje nějaká neprozkoumaná políčka, je nutné je navštívit a zaznamenat je do mapy. Tato činnost je mnohem jednodušší a používá se k ní druhý program nahraný v kostce, který na základě přijatých pokynů natočí robota do požadované polohy, posune ho o jedno políčko vpřed a vrací řídící službě výsledek měření vzdálenosti před robotem. Toto měření je využito v případě, že robot stojí před políčkem, které mu vybrala mapovací služba. Právě ona je zodpovědná za výběr políčka, které má být navštíveno a zjištěno, co se na něm nachází. Ve chvíli, kdy mapovací služba rozhodne, že jsou všechna políčka navštívena, prohledávaní končí.

Ještě bych rád poznamenal, že na přiloženém DVD je možné nalézt i původní implementaci určenou k řízení robota čistě pomocí MRDS a jazyka C#. Ale jak již jsem se zmínil, tato implementace není příliš funkční, pravděpodobně z důvodu problémů v množství komunikace mezi robotem a počítačem, jestli jsem správně pochopil funkčnost služeb řešících tyto záležitosti, posílají informaci při každém otočení kol o 1 stupeň, k určení o kolik se mají ještě pootočit. Toto je pravděpodobně největší problém. Pro případné zájemce či pokračovatele je proto součástí DVD i původní řešení.

6.4 Řídící služba simulovaného robota

To co se nepodařilo vyřešit u fyzického modelu robota, jsem vyřešil u toho simulovaného. Ten je napsán celý pomocí programovacího jazyka C# a simulačního prostředí VSE. Ikdyž obsahuje jednu odlišnost v provádění své úlohy oproti reálnému robotovi. Zatímco u reálného robota se při měření vzdáleností od překážky kolem robota otačí pouze ultra-



Obrázek 16: Logika programu v kostce robota

zvukový senzor, u toho simulovaného v důsledku neexistence simulovaného motoru, se musí otáčet celý robot. To ovšem způsobuje malé nedostatky v pohybu, jelikož otáčení robota není úplně přesné. Proto občas robot provádí korekce a zbytečné natočení o 45 stupňů spojené s pohybem vpřed. Tyto zbytečné pohyby jsem omezil zvětšením minimální vzdálenosti od stěny pro natočení o 45 stupňů. Proto většinou robot vykonává pohyby, kdy jede vodorovně kolem stěny nebo mění směr o 90 stupňů.

Základem pro implementaci jsou služby MazeSimulator, simulovaný model robota a mapovací služba a taktéž služby komunikující se senzory a pohonem robota. Všechny tyto služby jsou dirigovány pomoc řídící služby, která je pojmenovaná RobotExplorer-Simulation. Stejně jako u reálného robota je ovládání této řídící služby postaveno na základě stavového automatu popsaného v kapitolách věnujících se návrhu. Nejprve se služba snaží o prozkoumání obvodu virtuální místnosti, ve které se nachází. Postupuje po jednotlivých krocích, o velikosti 20 cm, po kterém následuje měření. Na základě měření se rozhodne co se bude dít dále a před vykonáním následujícího kroku se dotáže mapovací služby, jestli může pokračovat nebo jíž dojel do původního bodu, od kterého začínal měření.

6.5 Mapovací služba

Jedinou společnou službou pro obě implementace je mapovací služba. Jedná se o poměrně rozsáhlou službu, která se stará v první fázi pouze o záznam obvodu místnosti. V druhé fázi prozkoumává obsah místnosti. A nakonec ukládá vytvořenou mapu souboru.

K ukládání map do paměti se využívá objekt zvaný `Matrix`. Je to objekt představující matici, ve které každý prvek obsahuje nějakou hodnotu výčtového typu `MATRIX_ITEM_TYPE`. Jeho jednotlivé položky jsou popsány v tabulce 5. Každé políčko matice tedy představuje popis prostředí, ve kterém se robot pohybuje.

Celá matice je z pohledu jazyka C# naimplementovaná jako seznam seznamů obsahujících informace o jednotlivých prvcích matice. Každý seznam má pevně nastavenou velikost. Na počátku má matice velikost 10 x 10, v případě potřeby se dynamicky zvětší. Samotná matice nabízí několik veřejných metod pomocí, kterých je možno s ní komunikovat. A mnohem větší množství privátních metod pro řešení své logiky. Základ věřejných metod určených ke komunikaci s okolím je metoda na přidání nového prvku, metoda na zjištění jestli robot dosáhl startovní pozice a metoda k navrácení nejbližšího políčka. Z privátních metod bych jmenoval metody pro přidání řádku, sloupce, pro výpočet neprozkoumaných políček a nejbližšího volného políčka. Matice si také navíc pamatuje aktuální pozici kurzoru, který určuje místo zápisu. Důležitým údajem je informace o počátečním políčku, podle kterého se určuje ukončení určování obvodu. Toto políčko je na začátku je nastaveno na hodnotu X = 1 a Y = 1. Ve chvíli kdy dochází k přidávání sloupečku před něj nebo řádku nad něj, je tato hodnota měněna společně se změnou v počtu řádek či sloupců.

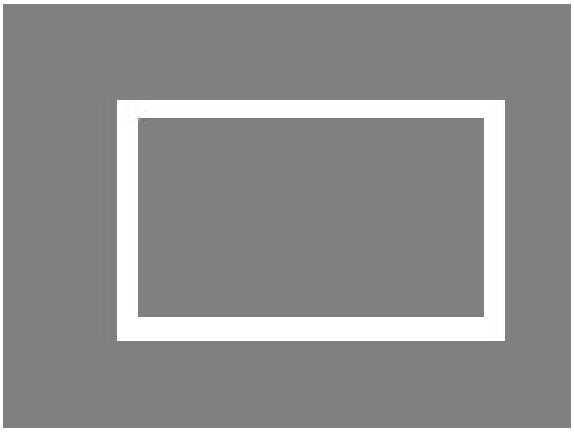
Další důležitou informací, kterou potřebuje mapovací služba k práci je informace o stavu natočení robota. Robot může být natočen do 8 směrů, natočení je relativní a je vždy určeno vzhledem k počátečním podmínkám. Na začátku je tedy robot natočen o 0 stupňů z pohledu mapovací služby začíná pohyb zleva doprava.

Jak jsem již několikrát zmínil, prohledávání místnosti začíná určením jejího obvodu. Samotné mapovaní začíná ve chvíli kdy je robot v blízkosti stěny, to je pro případ, že je nutné vykonat "startovací" sekvenci, pomocí které se robot dostane z prostoru na okraj místnosti. Robot začíná mapovaní v bodě X = 1 a Y = 1. Je to dáno tím, že robot při prohledávání obvodu v podstatě prozkoumává vždy dvě políčka, jedno po kterém projíždí a druhé je políčko tvořící stěnu, proto je matice nastavena o jedno políčko od kraje již od počátku. Po té pokračuje podle instrukcí z řídící služby. S řídící služby tedy vždy obdrží informaci o tom, kterým směrem se chce vydat robot, podle této hodnoty vypočte posun kurzoru pro zápis v matici. Ve chvíli kdy robot objede celý obvod místnosti končí první fáze této služby.

Ve druhé fází probíhá na začátku výpočet volných políček uvnitř místnosti. Pokud existují uvnitř místnosti nějaká volná políčka začíná jejich průzkum. Jelikož robot dorazil do počátečního bodu začíná výpočet odtud. Z vytvořeného seznamu volných políček najde políčko, které je nejblíže tomu aktuálnímu. V tuto chvílí posílá řídící službě informaci, jak má natočit robota a kterým směrem se má vydat. Po každém kroku se zkontroluje jestli robot nedosáhl požadovaného políčka nebo jestli na políčku, které měl navštívit není překážka. Pokud by se stalo, že mezi aktuálním políčkem a nejbližším volným políčkem leží zeď, mapovací služba se snaží najít další volné políčko. Prohledávání končí ve chvíli, kdy nezbývá žádné volné políčko.

Poslední částí této služby tvoří zápis finální pozice. Ve chvíli kdy máme prozkoumanou celou místnost, můžeme požadovat aby tyto výsledky byly nějak zaznamenány. V případě této služby se výsledná matice ukládá do obrázku, tak aby mohl být využit např. ve službě MazeSimulator. Jelikož pro označení políček máme pouze čtyři hodnoty použijeme tedy i čtyři barvy na označení všech objektů v matici. Ještě je nutné poznamenat, že před procesem překreslení z matice do obrázku je nutné matici zvětšit o jeden políčko na každé straně a označit ho hodnotou určenou pro označení volného prostředí. Toto opatření je nutné kvůli službě MazeSimulator, která na základě levého horního pixelu určuje barvu pro podlahu.

Na obrázku 17 můžeme vidět ukázkový výstup z mapovací služby, který zároveň může sloužit jako vstup pro službu MazeSimulator. Tento obrázek vznikl během testování na uměle vytvořené místnosti z krabic.



Obrázek 17: Ukázka možného výstupu z mapovací služby

7 Závěr

Jedním z cílů této práce bylo nastudovat prostředí MRDS, což se ukázalo jako mimořádně složitý úkol, jelikož se jedná o velice rozsáhlé prostředí, které skrývá spoustu možností a na nováčky líčí velké množství pastí. První pastí je nutnost uvědomit si, že při vytváření aplikací pro roboty je nutno přemýšlet odlišně než při psaní klasických desktopových nebo webových aplikací. Robotika je prostředí, které se chová asynchronně a je nutné tomu přizpůsobit i své myšlení. Celé prostředí přináší mnoho zajímavých konceptů, které se snaží práci ulehčit.

Knihovna CCR se snaží pomoci uživatelům této platformy ulehčit práci s vlákny. At' už se jedná o jejich vytváření, nebo o to, že nemusí myslet ani na zámky u proměnných společných. Stačí si pouze zapamatovat pár základních konstrukcí, pomocí kterých se dají vytvářet velice komplexní aplikace a celé pozadí tvořené vlákny je vytvořeno za nás. U služeb vytvořených pomocí další části zvané DSS, je vidět jak lze skloubit i programování zdánlivě neslučitelných prostředí jako je robotika a webové služby. Využití architektury REST přináší lehkost do vytváření služeb i do komunikace služeb a jejich skládání do jednoho celku.

Pro začínající a programovací jazyky neznalé lidi nabízí platforma MRDS grafické prostředí VPL, pomocí kterého mohou vytvářet plnohodnotné aplikace. Navíc je možné programy vytvořené pomocí tohoto nástroje i převést do programovacího jazyka C#. Poslední užitečnou součástí MRDS je simulační prostředí VSE, ve kterém můžeme začít s programováním robotů okamžitě aniž bychom vlastnili fyzicky nějakého robota. Můžeme si v něm vytvořit libovolné prostředí se zachováním fyzikálních vlastností.

Ale abych jenom nechválil prostředí MRDS, musím podotknout, že jsem musel řešit i velice složité problémy, které ne vždy bylo možné řešit pomocí prostředků tohoto prostředí. Nejpatrnější to bylo při komunikaci mezi řídícím programem, běžícím na počítači a robotickou jednotkou sestavenou z Lega NXT. Problémy vyústily v rozdělení řešení řízení pro reálného a simulovaného robota. Dalším problémem, který jsem řešil byla implementace simulovaného ultrazvukového senzoru. Tento senzor by měl být v současné verzi VSE obsažený, bohužel je postaven na základě laserového senzoru a proto byl zcela nevhodný pro použití ve výsledné aplikaci. Toto vyústilo v nalezení alternativního řešení, kterým se stal senzor společnosti SimplySim.

Pro realizaci úkolu této práce měl být použit robot postavený ze stavebnice LEGO NXT, který je cenově dostupný, je jednoduše rozšiřitelný o množství přídavných senzorů. Navíc se jedná o klasickou stavebnici ze série LEGO Technic, a můžeme si tedy hrát se sestavením finální podoby robota podle naší libosti nebo můžeme využít některý z návodů zveřejněných na internetu. Po prozkoumání možností, jsem si vybral pro implementaci úloh této práce dva senzory. A to dotykový senzor, který by měl sloužit pro odhalování nízkých překážek před robotem a druhým zvoleným byl ultrazvukový senzor, který nám umožňuje měřit vzdálenosti robota od překážky. A ještě jednu věc bylo nutno prozkoumat a to grafický programovací jazyk NXT-G, který je součástí distribuce.

Po prostudování a pochopení všech potřebných znalostí se bylo možné vrhnout na řešení úlohy určené v zadání. Vše začalo výběrem modelu fyzického robota, po jeho sestavení bylo možné provádět všechny pokusy o sestavení výsledné aplikace. Z počátku

vše vypadalo dobře, po vytvoření první verze řídící aplikace robot jezdil bez problémů. Ale to bylo jenom zdání. Ve chvíli, kdy se přidalo čekání na ujetí přesné vzdálenosti k obyčejnému ježdění objevil se problém popsaný výše. Proto bylo nutné část ovládací logiky přesunout z jazyka C# do NXT-G a ulehčit tak komunikaci mezi počítačem a robotem, která probíhala pomocí Bluetooth. Ve chvíli, kdy byly některé části převedeny na kód v NXT-G, bylo možné pokračovat, a to přidáním mapovací služby, která se stará o záznam pohybu robota v místnosti a snaží se jí zmapovat pro další použití a následně i uložit do výsledného obrázku.

Jedním z těchto použití je možnost využít obrázek s mapou místnosti jako základ pro vytvoření simulovaného prostředí. K tomu jsem použil službu s názvem MazeSimulator, kterou jsem si vypůjčil z knihy [1]. Ve chvíli, kdy existovalo simulované prostředí, vznikl další úkol, a to vytvořit simulovaný model robota, který by se podobal tomu reálnému. Nejprve bylo nutné pochopit jak takový model vůbec vytvořit. V podstatě jediným problém při jeho tvorbě byla neexistence simulovaného ultrazvukového senzoru ve standardní instalaci MRDS jak je popsáno výše.

Během studia i řešení úkolů a problémů jsem narazil na několik možností jak práci rozšířit, popř. co by bylo dobré prostudovat v oblasti funkčnosti některých prostředků MRDS. Jako první možnost rozšíření bych viděl vylepšení mapovací služby, která by lépe dokázala posílat robota k nejbližšímu volnému políčku v místnosti. V současné verzi pouze počítá vzdálenosti bez zjišťování , jestli mezi políčky neleží např. zeď. Další možností jak tuto práci vylepšit by bylo vytvořit samostatnou jednotku pro strategii a logiku řízení, její součástí by mohlo být i řešení k nalezení nejbližšího volného políčka. Zajímavým řešením by bylo propojit obě služby do jedné. To znamená, že například při průzkumu místnosti reálným robotem by se v simulátoru zobrazoval jeho pohyb i s vykreslováním mapy místnosti.

Asi nejzajímavější úkol bych viděl ve vytvoření vlastních služeb "ovladačů", kterými by se nahradily původní dodané v MRDS. Mohl by se tím dát vyřešit problém s pohybem robota. Toto je možné díky otevřenosti komunikačního protokolu stavebnice LEGO NXT, který je ke stažení na stránkách společnosti LEGO. Pokud by se podařilo vytvořit fungující službu, která by nezahlcovala komunikaci mezi robotem a počítačem, dalo by se uvažovat o sjednocení řídicích prvků pro reálného a simulovaného robota do jedné aplikace.

Než jsem začal tvořit tuto práci, nevěděl jsem o robotice ani o programování v MRDS vůbec nic. Po dokončení mohu směle tvrdit, že jsem se naučil spoustu nových věcí, které mě obohatily. Myslím si, že velkým přínosem bylo poznat asynchronní prostředí. Celá tato práce byla veliká výzva, jelikož dodnes neexistuje kompletní dokumentace k MRDS, která by popisovala kompletní prostředí. Jediná použitelná je kniha [1], pomocí které se dají pochopit začátky. Z tohoto důvodu se dá každý člověk, který se dá do programování robotů pomocí MRDS považovat za průkopníka, jelikož každý může přijít na něco nového.

8 Literatura

[1] Kyle, Johns, Travor, Taylor, *Professional Microsoft Robotics Developer Studio*, Indiana:Wrox, 2008

[2] Toledo, Sivan, *Analysis of the NXT Bluetooth-Communication Protocol*, http://www.tau.ac.il/ stoledo/lego/btperformance.html [cit. 2010-05-01]

[3] *Bluetooth Developer Kit, http://mindstorms.lego.com/en-us/support/files/default.aspx [cit. 2010-05-01]*

[4] *LEGO NXT, http://en.wikipedia.org/wiki/Lego_Mindstorm_NXT [cit. 2010-05-01]*

A Obsah DVD

Popis obsahu jednotlivých adresářů:

- `Text` - Obsahuje elektronickou verzi tohoto textu v podobě pdf
- `Knihovny` - Obsahuje potřebné knihovny
- `Impelementace` - Obsahuje jednotlivé implementace služeb
- `Implementace\MazeSimulator` - Obsahuje vše potřebné ke spuštění služby MazeSimulator
- `Implementace\Simulace` - Obsahuje služby potřebné ke spuštění simulace
- `Implementace\NXT` - Obsahuje prográmky potřebné pro obsluhu robota, vytvořené v prostředí LEGO Mindstorms NXT
- `Implementace\Concept` - Původní implementace služeb obsluhy robota
- `Implementace\Final` - Koncová verze služeb pro obsluhu reálného robota
- `Distribuce` - zde jsou uloženy všechny zkompilované služby včetně manifestů
- `Navod` - Návod na sestavení robota
- `Ostatni` - Obsahuje další informace k programování LEGO NXT v prostředí MRDS, obrázky, diagramy, dokumenty

B Příprava prostředí a spuštění služeb

B.1 Požadavky na nainstalovaný software

- Microsoft Visual Studio 2008
- Microsoft Robotics Developer Studio 2008 R2 Academic edition
- Lego Mindstorms NXT

B.2 Postup instalace služeb

V momentě kdy máme nainstalovány obě výše zmíněné aplikace, můžeme pokračovat následujícími kroky:

1. Pro případ, že bychom chtěli služby upravovat, popř. si je spustit ze zdrojových kódů, potom musíme zkopírovat z adresáře Implementace na přiloženém DVD do instalačního adresář MRDS
2. Pro případ, že chceme služby pouze spustit je nutné všechny služby zkopírovat z přiloženého DVD ze složky Distribuce do `MRDS\bin`.
3. Pokud budeme spouštět pouze simulační část této práce, můžeme v tuto chvíli přejít k části věnující se spouštění. Pokud ne, potřebujeme sestavit robota podle přiloženého návodu.
4. Ve chvíli, kdy máme sestaveného robota, musíme do něj nahrát řídicí software.
5. Nejprve je však nutné nahrát update pro prostředí LEGO Mindstorms NXT, který najdeme na přiloženém DVD v adresáři `Knihovny\Dynamic Block Update`. Tento update se bohužel již nenachází na oficiálních stránkách, proto je přiložen na DVD.
6. Po aktualizaci vývojového prostředí Lego Mindstorms NXT, nalezneme nabídku Tools ->Block Import and Export Wizard. Pomocí ní můžeme naimportovat nový programový blok potřebný pro provoz výsledné aplikace. Blok se nachází v adresáři `Knihovny\TextBlocks` a ve skutečnosti obsahuje více bloků, nás ovšem zajímá pouze ten starající se o konverzi řetězce na číslo. Tento blok je stažen ze stránek: http://www.nxtasy.org/repository/nxt-g-blocks/ [cit. 2010-05-01]
7. Ještě zbývá poslední krok ke zprovoznění robota. Musíme do něj nahrát řídící software, se kterým komunikuje řídící služba vytvořená v MRDS. Nejprve je nutné mezi robotem a Lego Mindstorms NXT vytvořit spojení, buď pomocí Bluetooth nebo USB. Poté otevřít ze složky na DVD `Implementace\NXT` oba programy a nahrát je do kostky.
8. Tímto je příprava před spuštěním hotova.

B.3 Spuštění služeb

V tuto chvíli bychom měli mít již vše potřebné nainstalované a zkopírované a zbývá již poslední krok k tomu, aby se virtuální či reálný robot rozjel. Tady je popis kroků potřebných ke spuštění.

1. Spustit příkazovou řádku pro MRDS
2. Potom se přesunout pomocí příkladu `cd` do složky s požadovanou službou
3. Pokud jsme ve složce služby stačí použít příkaz `dsshost` nebo využít připravených spouštěcích souborů s koncovkou .cmd (Pouze dvě složky obsahují smysluplné služby ke spuštění a to NXTExplorationRobot a RobotExplorerSimulation)
4. Pokud jsme dospěli do tohoto bodu bez chyb, měla by se právě startovat služba pro ovládání robota

C Vytvoření podvozku simulovaného robota

```
public class ExplorerRobotWithDrive : ExplorerRobot {
        public ExplorerRobotWithDrive()
        {       }

        public ExplorerRobotWithDrive(string name, Vector3 initPos)
            : base(name, initPos)
        {       }

        public override void Initialize (Microsoft.Xna.Framework.Graphics.GraphicsDevice device
            , PhysicsEngine physicsEngine)
        {
            base.ServiceContract = drive.Contract.Identifier;
            base.Initialize(device, physicsEngine);
        }
}

public class ExplorerRobot : DifferentialDriveEntity    {
        public ExplorerRobot()
        {       }

public ExplorerRobot(string name, Vector3 initialPos) {
            InitValue(name, initialPos);
            InitPlatform();
            InitWheels();
            InitCaster();
}

private void InitValue(string name, Vector3 initialPos)
{
    //Robot parameters
    MASS = 1.5f;
    CHASSIS_DIMENSIONS = new Vector3(0.07f, 0.035f, 0.16f);
    CHASSIS_CLEARANCE = 0.015f;
    FRONT_WHEEL_RADIUS = 0.0275f; // Front wheel diameter = 5.5 cm
    FRONT_WHEEL_WIDTH = 0.025f;
    CASTER_WHEEL_RADIUS = 0.0125f; // Caster (back wheel) diameter = 2.5 cm
    FRONT_AXLE_DEPTH_OFFSET = -0.05f;
    FRONT_WHEEL_MASS = 0.1f;
    MotorTorqueScaling = 20;
    //Robot settings
    base.State.Name = name;
    base.State.Pose.Position = initialPos;
    base.State.MassDensity.Mass = MASS;
    //  Chassis positions
    BoxShapeProperties motorBaseDesc = new BoxShapeProperties("chassis", MASS,
        new Pose(new Vector3(
        0,
        CHASSIS_CLEARANCE + CHASSIS_DIMENSIONS.Y / 2,
        0)),
        CHASSIS_DIMENSIONS);
    motorBaseDesc.Material = new MaterialProperties("chassisMaterial", 0.0f, 1.0f, 1.0f);
```

```
    motorBaseDesc.Name = "ChassisShape";
    ChassisShape = new BoxShape(motorBaseDesc);
    //Caster Wheel Position
    CASTER_WHEEL_POSITION = new Vector3(
        0,
        CASTER_WHEEL_RADIUS,
        CHASSIS_DIMENSIONS.Z / 2 - CASTER_WHEEL_RADIUS);
    //RIGHT Front Wheel
    RIGHT_FRONT_WHEEL_POSITION = new Vector3(
    CHASSIS_DIMENSIONS.X / 2 + FRONT_WHEEL_WIDTH,
    FRONT_WHEEL_RADIUS,
    FRONT_AXLE_DEPTH_OFFSET);
    //Left front wheel
    LEFT_FRONT_WHEEL_POSITION = new Vector3(
        -CHASSIS_DIMENSIONS.X / 2 - FRONT_WHEEL_WIDTH,
        FRONT_WHEEL_RADIUS,
        FRONT_AXLE_DEPTH_OFFSET);
}

private void InitWheels() {
    // RightWheel
    WheelShapeProperties wheelFRProp = new WheelShapeProperties(
        "FrontRightWheel", FRONT_WHEEL_MASS, FRONT_WHEEL_RADIUS);
    wheelFRProp.Flags |= WheelShapeBehavior.OverrideAxleSpeed;
    wheelFRProp.InnerRadius = 0.7f * FRONT_WHEEL_RADIUS;
    wheelFRProp.LocalPose = new Pose(RIGHT_FRONT_WHEEL_POSITION);
    RightWheel = new WheelEntity(wheelFRProp);
    RightWheel.State.Name = base.State.Name + "Right Wheel";
    RightWheel.State.Assets.Mesh = WheelMesh;
    RightWheel.Parent = this;
    //Left Wheel
    WheelShapeProperties wheelFLProp = new WheelShapeProperties(
        "FrontLeftWheel", FRONT_WHEEL_MASS, FRONT_WHEEL_RADIUS);
    wheelFLProp.Flags |= WheelShapeBehavior.OverrideAxleSpeed;
    wheelFLProp.InnerRadius = 0.7f * FRONT_WHEEL_RADIUS;
    wheelFLProp.LocalPose = new Pose(LEFT_FRONT_WHEEL_POSITION);
    LeftWheel = new WheelEntity(wheelFLProp);
    LeftWheel.State.Name = base.State.Name + "Left Wheel";
    LeftWheel.State.Assets.Mesh = WheelMesh;
    LeftWheel.Parent = this;
}

private void InitCaster() {
    CasterWheelShape = new SphereShape(
        new SphereShapeProperties("rear wheel", 0.001f,
        new Pose(CASTER_WHEEL_POSITION),
        CASTER_WHEEL_RADIUS));
    CasterWheelShape.State.Name = "Caster Wheel";
    CasterWheelShape.State.Material = new MaterialProperties("small friction with anisotropy",
            0.5f, 0.5f, 1);
    CasterWheelShape.State.Material.Advanced = new MaterialAdvancedProperties();
    CasterWheelShape.State.Material.Advanced.AnisotropicDynamicFriction = 0.3f;
    CasterWheelShape.State.Material.Advanced.AnisotropicStaticFriction = 0.4f;
    CasterWheelShape.State.Material.Advanced.AnisotropyDirection = new Vector3(0, 0, 1);
```

```
}

private void InitPlatform () {
    //base.State.PhysicsPrimitives.Add(box);
    Vector3 RVConsole_dimensions = new Vector3(0.005f, 0.13f, 0.005f);
    Vector3 LVConsole_dimensions = new Vector3(0.005f, 0.13f, 0.005f);
    Vector3 PlatformBase_Dimension = new Vector3(0.07f, 0.045f, 0.12f);
    Vector3 RHConsole_dimensions = new Vector3(0.005f, 0.005f, 0.05f);
    Vector3 LHConsole_dimensions = new Vector3(0.005f, 0.005f, 0.05f);
    Vector3 BumperPlatform_dimensions = new Vector3(0.13f, 0.09f, 0.005f);
    Vector3 UpperConsole_dimensions = new Vector3(0.07f, 0.005f, 0.005f);
    // Right Vertical console
    BoxShape RVconsole = new BoxShape(
        new BoxShapeProperties(
        "RVConsole", 0.001f,
        new Pose(new Vector3(
            CHASSIS_DIMENSIONS.X / 2,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y / 2,
            0)) ,
        RVConsole_dimensions));
    base.State.PhysicsPrimitives.Add(RVconsole);
    BoxShape LVConsole = new BoxShape(
        new BoxShapeProperties(
        "LVConsole", 0.05f,
        new Pose(new Vector3(
            -CHASSIS_DIMENSIONS.X / 2,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y / 2,
            0)) ,
        LVConsole_dimensions));
    base.State.PhysicsPrimitives.Add(LVConsole);
    BoxShape PlatformBase = new BoxShape(
        new BoxShapeProperties(
        "PlatformBase", 1.0f,
        new Pose(
            new Vector3(
            0,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y / 2 + PlatformBase_Dimension.Y / 2,
            -0.02f)),
        PlatformBase_Dimension));
    base.State.PhysicsPrimitives.Add(PlatformBase);
    BoxShape RHConsole = new BoxShape(
        new BoxShapeProperties(
        "RHConsole", 0.001f,
        new Pose(
            new Vector3(
            CHASSIS_DIMENSIONS.X / 2,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y / 2 + PlatformBase_Dimension.Y / 2,
            -CHASSIS_DIMENSIONS.Z / 2 + RHConsole_dimensions.Z / 3)),
        RHConsole_dimensions));
    base.State.PhysicsPrimitives.Add(RHConsole);
    BoxShape LHConsole = new BoxShape(
        new BoxShapeProperties(
        "LHConsole", 0.001f,
        new Pose(
```

```
        new Vector3(
            −CHASSIS_DIMENSIONS.X / 2,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y / 2 + PlatformBase_Dimension.Y / 2,
            −CHASSIS_DIMENSIONS.Z / 2 + LHConsole_dimensions.Z / 3)),
        LHConsole_dimensions));
    base.State.PhysicsPrimitives.Add(LHConsole);
    BoxShape BumperPlatform = new BoxShape(
        new BoxShapeProperties(
            "BumperPlatform", 0.05f,
            new Pose(
            new Vector3(
                0,
                CHASSIS_CLEARANCE + BumperPlatform_dimensions.Y / 2,
                −CHASSIS_DIMENSIONS.Z / 2 − 0.01f)),
        BumperPlatform_dimensions));

    base.State.PhysicsPrimitives.Add(BumperPlatform);
    BoxShape UpperConsole = new BoxShape(
        new BoxShapeProperties(
        "UpperConsole", 0.001f,
        new Pose(
        new Vector3(
            0f,
            CHASSIS_CLEARANCE + RVConsole_dimensions.Y,
            0f))),
        UpperConsole_dimensions));
     base.State.PhysicsPrimitives.Add(UpperConsole);
          }
     }
```